Notes on Programming Linguistics

John M. Wozencraft

and

Arthur Evans, Jr.

Department of Electrical Engineering

Massachusetts Institute of Technology

Cambridge, Massachusetts

February 1971

Preface

These notes represent the intellectual content of the subject 6.231, Programming Linguistics, taught in the Electrical Engineering Department at Massachusetts Institute of Technology to undergraduates who contemplate a serious professional interest in computer science. An important part of the subject material has to do with PAL, a computer programming language designed to be an integral part of the educational experience, and it is intended that students perform, on a computer, a set of homework exercises in PAL. The details of the PAL language are not covered in these notes but instead in a separate publication, referred to in these notes as the PAL Manual. It is assumed that readers of these notes have access to the PAL Manual.

Acknowledgements

The thinking of the authors in preparing this material has been strongly influenced by both Christopher Strachey and Peter Landin, each of whom visited MIT for a semester or more and presented a series of lectures. We are pleased to acknowledge our debt to each of them -- their influence on our thinking has been immense. In addition, signifigant contributions have been made by Martin Richards and James H. Morris, Jr., as well as by Franklin L. DeRemer, D. Austin Henderson and Robert H. Thomas. Judith L. Piggins assisted with the PAL programs at the ends of Chapters 3, 4 and 5.

We are grateful to the Electrical Engineering Department of the Massachusetts Institute of Technology for encouraging over a period of years the curriculum development of which the present material is a part, and to the Ford Foundation, for its support of curriculum development at M.I.T.

The typing and editing of the present version has been capably done by Mrs. June Pinella. The computer system used was CTSS, Compatible Time Sharing System, operated by the Information Processing Center at M.I.T.

This work was supported (in part) by Project MAC, an M.I.T. research program sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract Number Nonr-4102(01), and in part by the National Aeronautics and Space Adminstration, grant NGR-22-009-393.

Finally, particular thanks go to many generations of students and teaching staff who have helped debug earlier versions of these notes.

Table of Contents

Table of Contents

Table of Contents

Chapter 1

INTRODUCTION

1.1 Perspective

The title of these notes includes the phrase "programming linguistics", and since this phrase is not in current usage it behooves us to define it. But doing so is easy: inasmuch as "linguistics" is the science of language, it follows that "programming linguistics" is the science of programming languages -- those languages whose purpose is communication with computers. It would be premature to claim that this science is fully understood, or even that it is a science. Nonetheless, the importance of delineating such a science is manifest in the context of today's computer technology and its history.

Historical Sketch

Although most major developments in digital computers have occurred within the past twenty years, the underlying concepts were anticipated by an Englishman, Charles Babbage, in the 1840's. Babbage had built a small mechanical device that calculated numerical tables digitally by a method of polynomial approximations. The insight gained through this development led him to propose a more ambitious machine that would have incorporated the central features of today's computers: a digital device with flexible internal programs and decision-making capabilities.

Unfortunately the technology of that day could not support the actual construction of the machine (the proposal involved long trains of wooden gears) and Babbage's ideas lay dormant until embodied in the Z3 and the Mark I, developed in Germany in 1941 and at Harvard in 1943, respectively. Shortly thereafter, John Von Neumann proposed the Princeton machine, which introduced the idea of a program which could modify itself. This computer, using vacuum tube logic and electrostatic storage, was operating by 1949. Together with a few other contemporary machines constructed independently in this country and in England, the Princeton machine marked the launching of modern digital computer technology.

Of course, Babbage's ideas are not the only early ones to which we are indebted. For example, punched cards for data storage were introduced in the 1890's by H. Hollerith while he was director of the U. S. census. (The card size in use today is identical to that of the dollar bill of that period.)

Since 1950 computer technology has advanced at a furious pace. Magnetic core memories (invented by J. W. Forrester and installed in the MIT Whirlwind computer in 1951) have almost entirely replaced electrostatic devices for rapid-access data storage, and vacuum tubes have given way to transistors. Current developments in thin-film memories and integrated circuits are having an impact, and more flexible input/output devices are encroaching upon the domain of the punched card. These advances can be counted on to provide levels of computer size, speed and reliability which over the foreseeable future will continually enlarge the complexity of problems to which computers can be addressed.

Indeed, it seems fair to say that developments in computer hardware have outstripped developments in our understanding of how to harness them effectively. Important research in pattern recognition, artificial intelligence, information retrieval, natural-language translation, and man-machine interaction proceeds with vigor; but no scientist believes that the central problems in any of these fields have been fully understood, much less resolved.

The crucial issues in such research are intellectual, not linguistic. Nevertheless, the difficulty man experiences in communicating with machines often detracts markedly from the progress that is made. First, in bringing their intellect to bear upon these problems, men experience a need for a method of expression which is simultaneously well-matched both to their own patterns of thought and to the computer's ability to comprehend. (For example, large-scale work in artificial intelligence could not begin until suitable languages such as LISP and IPL had been developed.) Second, many of the issues are sufficiently profound that they seem unlikely to yield to the

inspiration of a single researcher. There is need for communication not only with computers but between men -- often between groups of men working independently on disparate problems.

The evolution of computer language appears more responsive to the first of these two communication needs than to the second. Starting with the acceptance of Fortran in 1956 the development of special-purpose languages has mushroomed. In part this may be attributable to mistaken hope that a tough intellectual problem will become amenable once a language suitable for dealing with it is available. More substantive, however, is the economic aspect: A good programmer working on tough problems can be expected to produce about 100 lines of debugged program per month. Including overhead, each line of working program costs about $30, regardless of the language used. Since one line written in a language well-adapted to a problem is equivalent to from 10 to 100 lines in an ill-adapted language, the economic gain is manifest.

As a result, somewhere between 50 and 75 major high-level programming languages, and over a thousand dialects, now exist. Moreover, the inadequacy of documentation for most dialects -- even for most languages -- is appalling; often the precise effect of a phrase can be determined only either by direct experimentation with a specific implementation or by intimate knowledge of the compiler. Historically, a consequence has been that substantive communication between different workers in a computer science has been inhibited.

Computer Art and Computer Science

Of course, not all lack of adequate intercommunication is attributable to proliferation of computer languages. Perhaps the heart of the problem is the enormous current ratio of computer art to computer science. The number of theorems relevant to computation is distressingly small, and the insight provided by the theorems that do exist is even smaller. In this regard our state of knowledge of computation is analogous to that prevailing in the electrical communications field before the 1940's. Over the preceding 40 years it had become possible to communicate with great effectiveness, but there were no theoretical underpinnings to tell us what communications was all about. Not until the work of Wiener, Kotel'nikov and Shannon did it become possible to evaluate the performance of an actual or proposed communications system in terms of absolutes.

A similar situation now obtains in the field of computation. There is no metric against which to measure the value of a proposal or a point of view. In the absence of broad professional agreement as to what is important, each little group tends to go its own way, hoping that its approach will prove fertile. Usually the effort required to understand someone else's program is incommensurate with the insight to be gained through doing so, and identical problems get solved over and over again by different people in different installations. The ability to prove the equivalence of programs is almost as elusive a goal now as when first enunciated in 1957 by McCarthy as a crucial step towards a viable theory of programming.

Standing in the way of the development of such a theory are two problems. The first (less fundamental) one concerns the dependence of the outcome of a program on the details of the computer on which it is run. For example, a program may overflow the memory of a small computer, but not of a larger one; or round-off-error accumulation may cause a program to abort on a machine with a smaller word-length, but not with a larger one. Presumably the effects of machine dependence can be eliminated, at least in principle, by considering program equivalence in relation to some canonic machine, perhaps with infinite memory and word-length.

A more basic impediment is the problem of variables. In conventional mathematics a "variable", say x in the equation

$$x^2 - 1 = 0$$

actually denotes a constant value (or set of allowable values) which may or may not be known explicitly; but even if implicit, at least the denotation is invariant with respect to time. By contrast, in the course of an iterative computation the value denoted by x may well be different during successive epochs. The only mathematical technique we have for treating this situation involves the concept of "state", wherein the evaluation process is viewed as a sequence of transformations on data stored within the machine and each transformation in turn depends upon the data configuration produced by its predecessor. From an analytical point of view the deficiency in

the approach is that it reinvolves us in the same laborious detail we use computers to avoid.

When the computer is simple enough, of course, the specification of its state is also simple and the state transformation method becomes very powerful as well as very fundamental. Indeed, by following this approach Turing (in 1936) was able to prove two theorems central to mathematical logic: that a certain extraordinarily simple machine can compute anything that a machine of arbitrary complexity can compute; and that certain questions are undecidable, i.e., that no machine can compute an answer to them. (Using a different approach we show later that whether or not an arbitrary program will loop indefinitely is an example of an undecidable question.)

Strictly speaking, the foregoing statement of Turing's results is incorrect; we should have said "can compute anything that any machine in a broad class of arbitrarily complex machines can compute." But other logicians have proved the same results starting from entirely different premises. For example, Post has considered analogous problems via the manipulation of strings of symbols, Kleene via recursive functions, and Church and Curry via algebraic manipulations. In each case the class of problems to which answers are computable is the same. Accordingly, there is a wide-spread belief -- called "Church's Thesis" -- that Turing's result is inevitable.

The Study of Computation at M.I.T.

Although of great importance, the results of mathematical logic to date unfortunately do not in themselves constitute a viable theory of computation, so that any organized course of study in computer science reflects to a much larger degree than is desirable the bias and viewpoint of its organizers. We take the point of view that it is useful to concentrate upon three mutually complementary categories of subject matter.

The first category, to which this text is addressed, concerns principles and concepts underlying programming languages and the abstract specification of algorithms. To a large extent we presume the existence of an idealized evaluating mechanism that is free of important practical constraints such as finiteness of memory. The second category concerns the structural organization of real machines for carrying out algorithms, and treats issues that arise when the goal is to make practical computers appear to users as if they were ideal. The third category concerns methodology for dealing with very complex systems; the rationale here is that the principal technological impact of computers lies in the possibility of dealing effectively with systems vastly more complicated than was possible before. Successor subjects 6.232, Computation Structures, and 6.233, Information Systems, treat categories two and three respectively.

It is clear that a great many important topics are not contained within any of these categories. The hope, however, is that the three subjects, in conjunction with appropriate subjects in mathematics, provide appropriate common background for continued studies of a more specialized nature.

The danger of bias extends also into the internal content of each category discussed above. As mentioned, "linguistics" is the science of language, and we have already disclaimed common agreement that much science of computation exists. On the other hand, rather than to study the details of one or two particular languages out of 50 or 75, it seems preferable, both educationally and intellectually, to establish a coordinate system in terms of which many programming languages can be described. The delineation of such a coordinate system is one of two major objectives in our study of programming linguistics.

The pedagogical approach that we follow uses a particular language, PAL, as an educational vehicle. (The acronym is for Pedagogic Algorithmic Language.) We begin our study of PAL by gaining operational familiarity with the language by seeing in Chapter 2 many examples of its use. That chapter contains the mathematical underpinnings on which the formal definition of PAL's semantics is based. Chapter 3 is concerned with specification of a gedanken evaluator for the "applicative" subset of PAL, which is the subset directly related to the λ-calculus. (Gedanken is a German word that translates best into the English phrase "thought of". This gedanken evaluator is motivated strongly by the work of Landin.) In Chapters 4 and 5 we extend the gedanken evaluator to accommodate the "imperative" subset of PAL: linguistic constructs such as the assignment and goto commands. Taken together, these three chapters formalize PAL semantics. In essence, the

aggregate of computational concepts developed along the way represents the coordinate system we seek to develop. The applicability thereof to other languages is indicated enroute, but both time and cohesiveness of presentation militate against thorough treatment of this topic.

A second major objective is to inculcate skill and style in programming. To achieve this end there is a continuing series of home problems to be programmed in PAL.

1.2 Underlying Concepts

Before proceeding to the details of language definition, it seems advisable to discuss in a preliminary way the conceptual base on which the language is founded. In general, the point of view throughout these notes is that computation is concerned with transformations among abstract objects. The key word here is abstract.

To gain insight into this viewpoint, consider part (a) of Figure 1.2-1, in which the three objects represented by x's are distinguishable by their relative positions. A transformation mapping unordered pairs of these objects into single objects is represented in part (b) of the figure. Since both members of the pair may be the same, there are 6 cases to be considered. In the figure each case corresponds to an arrowhead having two tails; the interpretation is that the transformation maps the objects on which the tails terminate into the object on which the arrowhead terminates.

An alternate way to represent a transformation is illustrated by Table 1.2-1:

	0	1	2
0	0	1	2
1	1	2	0
2	2	0	1

Table 1.2-1

Here we distinguish three objects by means of the names 0, 1, 2 and specify the transformation by placing the result at the intersection of the row and column corresponding to each choice of argument pair. The symmetry of the table about its principal diagonal implies that order within the argument pair is immaterial. It is clear from the table that the transformation can be described as "integer addition modulo-3".

There are three important aspects to these examples. First, the reader may notice that there is something in common between the system defined by the transformation and objects of Figure 1.1 and the system defined by the transformation and objects of Table 1-1. Indeed, the two systems are equivalent in the sense that

(1) there is a one-to-one correspondence between the entities of the two systems, and
(2) this correspondence is preserved under the transformation of the two systems.

Two such systems are called isomorphic. We may think of the figure and table as different representations of a single abstract system that underlies them both. By "abstract" we mean independent of representation, so that the only questions we can ask about objects in the abstract system concern how they behave under the transformation. The situation is analogous to that in physics: it is not meaningful to ask what an electron is, but only how it behaves. An abstract object is no more or less than a bundle of properties.

The second important aspect of the example is that it is convenient (though inessential) to be able to refer to objects by name -- the tabular representation is more transparent to the human mind than is the figure, especially since we chose the names {0, 1, 2} in accordance with established convention. But the choice of names is clearly arbitrary; an equally valid representation of our abstract system is that of Table 1.2-2, in which we use the symbol III because the Romans had no symbol for zero.

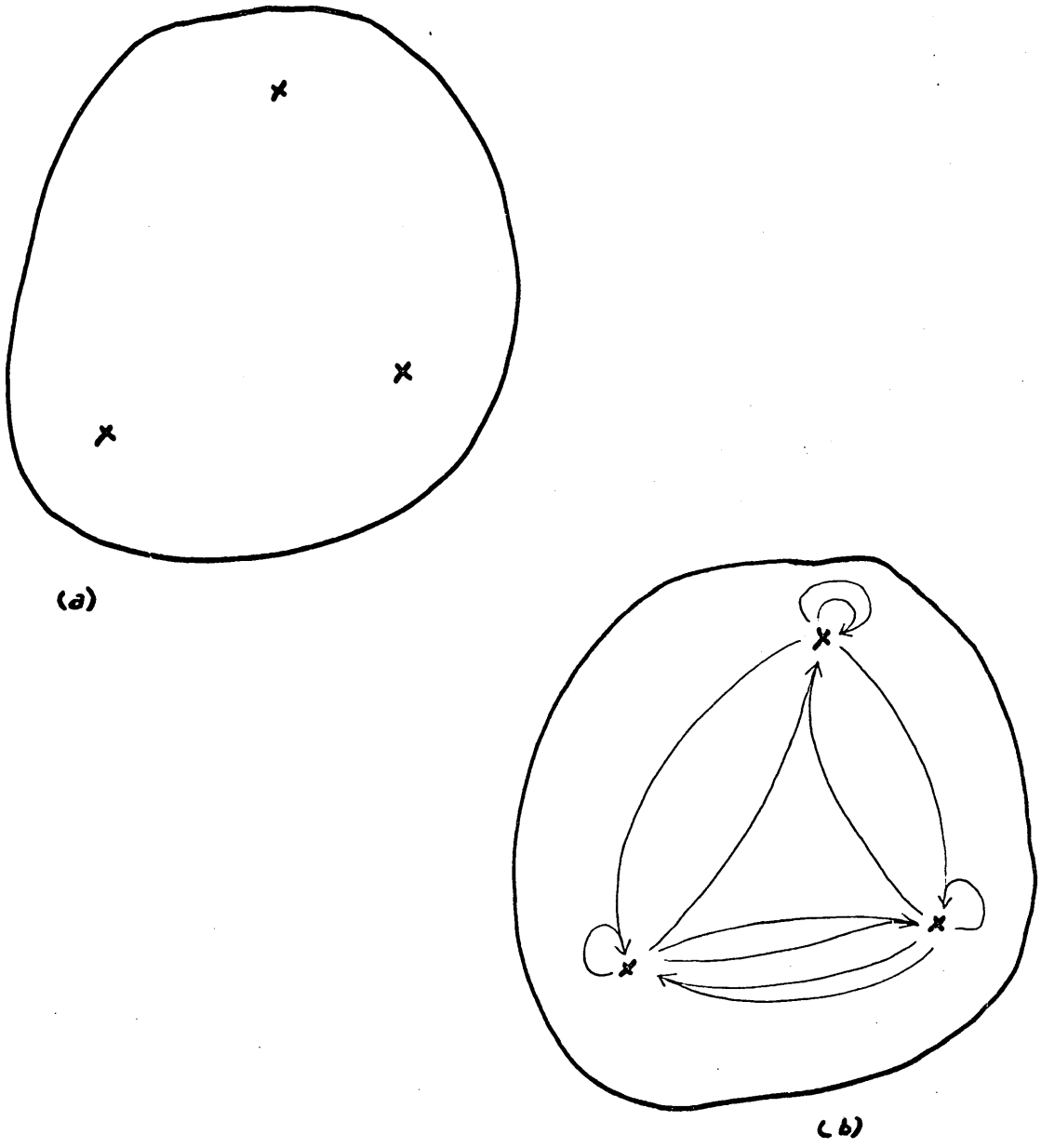


Figure 1.2-1: (a) A space of three objects.
(b) A binary transformation on them.

⊕	III	I	II
III	III	I	II
I	I	II	III
II	II	III	I

Table 1.2-2

Moreover, it is permissible -- often desirable -- to economize on the use of names by providing them for only some, not all, of the objects in an abstract system. For instance, if we adopt ⊕ as the name of the transformation of Table 1.2-2, each of the functional expressions

$$\oplus\ (\mathrm{I},\ \mathrm{II}) \tag{1.2-1a}$$

$$\oplus\ (\mathrm{II},\ \oplus\ (\mathrm{II},\ \mathrm{II})\ ) \tag{1.2-1b}$$

and the name III designate the same abstract object; i.e., the same object in abstract space. Indeed, it is evident from (1.2-1b) that just the names ⊕ and II suffice to permit the designation of all three objects. (Alternatively, ⊕ and I would also suffice.)

The fact that "transformation" is just another word for "function" is the third important aspect of our example. Both words imply neither more nor less than a mapping from one set of objects called the domain of the function onto a set (perhaps the same) of objects called the range of the function. Functions that map numbers onto numbers are the most familiar ones in elementary mathematics, but in computation we are often concerned with more general mappings. In these more general situations the word transformation may at first seem more natural.

There are two common ways of specifying a function. The direct method is to enumerate all of the result-argument pairs, as in Tables 1.2-1 and 1.2-2. The indirect method is to express the function in terms of other functions that have already been specified. For example, we might define a function whose domain is the set of single objects from our triad, in terms of the function ⊕ whose domain is the set of pairs of these objects, by writing

$$f(x) = \oplus\ (\mathrm{II},\ x) \tag{1.2-2}$$

The concepts introduced above are basic to our point of view and are elaborated further throughout these notes. For instance, we think of a program as the specification of a (perhaps complicated) function in terms of simpler basic functions; and we think of running a program as determining that object in an abstract space which results from application of the function to particular arguments (data) chosen from the same abstract space.

It should be reemphasized that this is not the only possible point of view that could be adopted. In particular, one alternative would be to back off from abstract objects and relate programs directly to transformations on the concrete representation of data in an actual or idealized computer. One of our purposes, however, is to break away wherever possible from machine dependence and to separate the intellectual problems of algorithm specification from the equally important but distinct problems of machine design. In these terms one aspect of machine design is producing a hardware implementation of basic functions and data such that the resulting system is a concrete representation of a desired abstract system. By contrast, a central linguistic problem is producing an exact specification of the transformational properties that define the abstract system itself.

1.3 An Algorithm for Symbolic Differentiation

The relation between the abstract point of view introduced above and the activity of programming may be demonstrated in terms of an algorithm for symbolic differentiation of simple algebraic expressions. First we generate the algorithm, and then formalize it as a PAL program.

The immediate question that arises when we are confronted with an expression such as

$$z*x - x(3.6 + x/y) \tag{1.3-1}$$

concerns what it is that the expression denotes. Clearly, different interpretations are possible, and we are not able to adjudicate among them solely on the basis of the expression itself. For example, (1.3-1) might be taken as an arithmetic expression. Then the operators would denote arithmetic functions and the letter symbols would be names of numbers, so that (1.3-1) would denote a number. On the other hand, we are interested here in symbolic differentiation, so that (1.3-1) is to be taken as an algebraic expression and must be given a non-numerical interpretation. In order to proceed, we must specify what this interpretation is to be; in other words, we must specify what transformational properties pertain to the class of things we call "algebraic expressions".

Properties of Algebraic Expressions

In writing (1.3-1) we have made use of several conventions that people frequently find useful:

(1) the convention which allows us to infix the symbols {+, -, *, /};
(2) the precedence convention which allows us to write "3.6 + x/y" in lieu of "3.6 + (x/y)"; and
(3) the juxtaposition convention which allows us to write "x(3.6 + x/y)" in lieu of "x*(3.6 + x/y)".

An alternate representation of (1.3-1), one which makes the structure of the expression more explicit, is the "tree diagram":

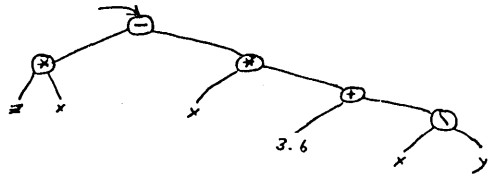


Figure 1.3-1: Tree Form of Equation (1.3-1)

Here each node of the tree is labelled with one of the symbols {+, -, *, /} and each terminal branch of the tree is labelled with an atomic expression: an expression which has no algebraic substructure. Even in this more explicit representation, however, we note that conventions are still important. For instance, the assumption is made that the divisor and the subtrahend are written on the right branch descending from a "/" or "-" node, respectively.

Presumably we are willing to accept as a valid algebraic expression any entity that can be represented by such a tree. We therefore make the following definition:

An algebraic expression (algex) is either
  an atom,
  or it has
    an operator, which is one of {+,-,*,/}
    and a left operand, which is an algex,
    and a right operand, which is an algex.

To a large extent, the succinctness of this definition stems from the fact that it is recursive, in the sense that the class of things called algebraic expressions is defined in terms of itself.

The definition is "abstract" in the sense that it provides vital information about the transformational properties of algebraic expressions, and hence about the properties that must be evidenced by any valid representation thereof. In particular, the definition implies that we can determine of any algebraic expression

a. whether or not it is an atom, and
b. if not which of the symbols {+, -, *, /} is its operator and what two algebraic

expressions are its right and left operands.

In order to complete the characterization of algebraic expressions, of course, we must also establish the properties of atoms. In general -- certainly in the present case, in which we seek the symbolic derivative with respect to some particular atom, say x -- we are interested not only in whether or not an expression is atomic but also (whenever it is) in whether or not it is some special atom. Of the class of entities we choose to call atoms, therefore, we require that:

c. given any two instances of atoms, we can determine whether or not they are instances of the same atom.

Informal Description of the Algorithm

An algorithm for symbolic differentiation of algebraic expressions involves the specification of a transformation that takes two arguments (the expression to be differentiated, and the variable of differentiation) and produces therefrom another expression; i.e. another entity that also has the properties of the class "algebraic expressions". It is convenient to refer to the function and its arguments by names, say D, E, and x, respectively. Then the algorithm is simply a specification of the result, D(E, x), of applying the transformation D to E and x. An informal description that embodies the essential features of the desired transformation is on the next page. There are several notable aspects of this description:

1. It is exceedingly long-winded.
2. It is convenient to introduce new names (Op, L, R, L1, R1) as part of the description itself.
3. It depends heavily upon all the properties of algebraic expressions.
4. It is a circular description, in the sense that in order to determine L1 and R1 we must be able to apply the very transformation that is being described.

In the process of formalizing the algorithm as a PAL program we find that it is possible to be much more succinct, primarily through adoption of a representation for algebraic expressions that is both specific and convenient.

Representing Algebraic Expressions in PAL

Knowing the abstract properties of algebraic expressions permits us to proceed to the problem of choosing a representation of these entities in PAL. There are three classes of abstract objects in PAL which are well adapted to this purpose. The class of strings is useful for representing "atoms"; the class of tuples is useful for representing non-atomic algebraic expressions; and the class of truthvalues is useful for testing hypotheses. The properties of these objects, plus PAL's syntactic conventions for denoting them, are detailed in Chapter 2, as well as in the PAL Manual.

There are many different representations for algebraic expressions possible in PAL -- indeed, it is characteristic of the field of computation that there is seldom a unique solution to any problem, although one solution may be preferable to another in terms of transparency to another reader or economy of implementation.

The solution that we choose to consider here represents algebraic expressions by strings if they are atomic, and otherwise by tuples of order 3. A tuple in PAL is much akin to a vector in many other programming languages, but a vector usually must be homogeneous (in that all of its components must be of the same type, such as integer or rational) while a tuple may be heterogeneous. We exploit this freedom to satisfy the requirement that an operand in an algebraic expression may be a non-atomic algebraic expression. For non-atomic expressions, the requirement that we be able to determine the operator and the two operands is satisfied by adopting the convention that:

(1) the first component of the tuple represents the left operand;
(2) the second component of the tuple is one of the strings ['+', '-', '.', '/'] and represents the operator; and that
(3) the third component of the tuple represents the right operand.

(Note that the string for the infix multiply is '.'. PAL provides special conventions for quoting

```
The derivitive of E with respect to x is

    if E is x then the atomic expression 1
    otherwise the atomic expression 0

otherwise
    let L denote the left operand of E
    and Op denote the operator of E
    and R denote the right operand of E

    next,
    let L1 denote D(L, x)
    and R1 denote D(R, x)

    if Op is the symbol +, then the expression
        whose left operand is L1
        and whose operator is the symbol +
        and whose right operand is R1

    otherwise, if Op is the symbol - then the expression
        whose left operand is L1
        and whose operator is -
        and whose right operand is R1

    otherwise, if Op is the symbol * then the expression
        whose left operand is the expression
            whose left operand is L
            and whose operator is the symbol *
            and whose right operand is the symbol R1
        and whose operator is the symbol +
        and whose right operand is the expression
            whose left operand is L1
            and whose operator is the symbol *
            and whose right operand is R

    otherwise, if Op is the symbol / then the expression
       whose left operand is the expression
                whose left operand is the expression
                    whose left operand is L1
                    and whose operator is the symbol *
                    and whose right operand is R
                and whose operator is -
                and whose right operand is the expression
                    whose left operand is L
                    and whose operator is the symbol *
                    and whose right operand is R1
       and whose operator is /
       and whose right operand is the expression
           whose left operand is R
           and whose operator is the symbol *
           and whose right operand is R

    otherwise, E is improperly formed and does not
               represent an algebraic expression.
```

an asterisk, and we prefer not to involve ourselves with them at this point.) Thus the algebraic expression of (1.3-1), whose tree form is shown in Figure 1.3-1, is represented by the PAL expression

```
z*x - x*(3.6 + x/y)                                    (1.3-2)
```

On page 1.3-7 we listed three properties of any representation of algebraic expressions. We now show that these requirements are met. Whether or not an algebraic expression is atomic may be determined by testing its representation with the predicate "Atom", so that property (a) implied by our definition is satisfied. Moreover, whether or not two atomic expressions are the same may be determined through use of the infix functor "eq", which satisfies property (c). Since each operator is represented by the occurrence of one of the strings '+', '-', '.' or '/' as the second component of a tuple, "eq" also permits determination of the operator of a non-atomic expression, as required by property (b). Finally, the remaining requirements of property (b) are satisfied by knowing that the left and right operands are represented respectively by the first and third components of a tuple.

Program for Symbolic Differentiation

Given the ability to represent algebraic expressions, it is not difficult to convert our informal symbolic differentiation algorithm into a corresponding PAL program. The program written below mirrors the semantic intent of the informal algorithm exactly; referral thereto should make the program relatively transparent even without extensive familiarity with PAL.

```
def rec D(E, x) =
     test isstring E  // Is it an atom?
     ifso (E eq x -> '1' | '0')  // Yes - check for x.
     ifnot  // No, it's not.
       (  let L  = E 1  // The left operand.
          and Op = E 2  // The operator.
          and R  = E 3  // The right operand.
          in
          let L1 = D(L, x)  // Derivitive of left.
          and R1 = D(R, x)  //  ditto right
          in
             Op eq '+'  ->  (L1, '+', R1)
          |  Op eq '-'  ->  (L1, '-', R1)
          |  Op eq '.'
               ->  ( (L, '.', R1), '+', (L1, '.', R) )
          |  Op eq '/'
               ->  ( ( (L1, '.', R), '-', (L, '.', R1) ),
                     '/',
                     (R, '.', R)
                   )
          |  error  // Improper data
       )
```

A comment is in order concerning the format of this program. An effort has been made that the indenting scheme used indicate the parsing. In all programs in this text, the physical layout is such as to facilitate as much as possible the reader's understanding of the programmer's intent. It is reccommended strongly that the student make it a practice to follow such conventions in his programming.

The last line but one uses the non-PAL reserved word "error", with obvious meaning.

1.4 Technique for Language Definition

The major thrust of the rest of these notes is to define both the syntax and semantics of the programming language PAL. Before getting started on the details, it seems appropriate to say a little about what we are about to do and how we are going to go about doing it.

The "what" is fairly straightforward, and was suggested above: We are going to define a language. But the definition is merely a surface manifestion of the important intellectual content. Recall that these notes intend to deal with "programming linguistics": the study of those languages used by people in communicating with computers. Our intent is to illuminate various ideas relevant to such languages, and our approach to doing so is to describe a particular language, PAL, expecting thereby to reveal one manifestation of those ideas. It is not our claim that PAL should be studied intensively for its own sake. To the contrary, PAL is a pedagogic tool, studied in depth solely because of the concepts it reveals. Inasmuch as the original purpose in designing PAL was to use it as a teaching vehicle to illuminate precisely those concepts, it is not surprising that the points we wish to make can be made quite cleanly by citing PAL.

Subsets of PAL: For the sake of expository convenience, we find it useful to isolate three subsets of PAL. The first such subset we consider involves only those aspects of PAL having to do with the application of functions to arguments, and the linguistic facilities needed by the programmer to define functions of his choosing. We call this subset R-PAL, roughly because it includes those constructs that can be used on the right side of assignment statements. (A better justification of the term R-PAL, as well as the term L-PAL mentioned below, must await chapter 4.) As we see, R-PAL is solidly based on the mathematics of the λ-calculus. The differentiation program D given earlier is written entirely in R-PAL.

The second major subset is L-PAL, and includes assignment statements. The presence of assignment raises problems such as the following: Suppose that x is a structure of some sort with three components, a, b and c, and that the assignment statement

```
b := b + 1
```

is executed. Is x changed thereby? The answer has to do with whether or not b and the second component of x occupy the same "memory location". To use the technical term we like, does b share with x's second component? Explication of L-PAL involves understanding of the mechanism of sharing and its ramifications, so that questions such as the above can be answered. (The answer in this case turns out to be "yes".)

PAL is completed by adding to L-PAL the concepts of labels and transfer of control (goto statements), producing what we call J-PAL ("j" for jumping). J-PAL is identical with the PAL of the Manual, and the term is only used on occasion to emphasize the hierarchical sets of languages.

The Plan of Attack

Because the development presented in the rest of these notes is full of details, it is all too easy for the reader to lose sight of the ultimate goal and to fail to see how a particular topic fits into the overall flow. Thus it seems appropriate to provide, at this point, an overview of the entire development.

Before starting on the definition of PAL in chapters three, four and five, we study in chapter two PAL's underlying mathematical foundations. We have two major areas to study: The first is to define carefully (or at least to show how one can so define) all of the objects with which PAL deals. These are the objects such as integers and strings, which were used in the example on differentiation, as well as rational numbers, truthvalues, and others of which we have not yet made use. We take the attitude that it is not adequate to say, for example, that integers are available in PAL, with their "usual" properties. Instead, we feel it necessary to be able to define precisely just what those properties are. While this approach may seem artificial with respect to integers (after all, we all "know" their properties), it pays off when we introduce classes of objects (such as tuples) whose properties are not so familiar.

The second major area of study in chapter two has to do with functions. Since much of what

goes on in programming languages has to do with the application of a function to arguments, we adopt a point of view in language description that emphasizes functional application. We study in depth that mathematical discipline called λ-calculus, a study which yields us excellent tools for dealing with functions. The capstone of the chapter is a mathematical treatment of recursion.

In chapter 3, the conceptual foundations laid in chapter 2 are used upon which to build the R-PAL subset. R-PAL is developed as syntactic sugaring for the λ-calculus. Although λ-notation is adequate for expressing any algorithm, it is not very convenient for use by people to describe complex transformations. Thus our approach in defining R-PAL is to show how various R-PAL constructs are just ways of rewriting λ-expressions in a properly human-engineered form. Having approached R-PAL that way, we can explain the meaning of any R-PAL program in terms of the mathematics developed in chapter two.

It is unfortunately the case that the λ-calculus, although admirably suited for explaining R-PAL, is not well suited for explaining the imperatives that distinguish L-PAL and J-PAL from R-PAL. Thus, having provided an adequate explication of the meaning of any R-PAL program, we undertake to provide a second such explanation. This one is presented in two ways: First, we present it as a conceptual evaluating mechanism called the CSE evaluator. This mechanism accepts as input a PAL program to be evaluated and returns the "value" of that program. Clearly, understanding such a mechanism implies understanding PAL. Second, because the mechanism is fairly complicated, we exhibit a PAL program which represents the algorithm carried out by the machine. That is, we exhibit in PAL a program that explains the meaning of PAL programs. The apparent circularity is illusionary and not real, since we can explain the meaning of that PAL program by appeal to the λ-calculus.

In chapter 4, we develop L-PAL by studying assignment and the related problem of sharing. Now the CSE mechanism developed in chapter 3 shows its value, since it is this mechanism rather than the λ-calculus explanation upon which we build to accomodate assignment. Finally, in chapter 5 we build further on the mechanism to accomodate also labels and goto statements.

Abstract Syntax

There are two aspects of language description that must be considered -- syntax and semantics. In the last few paragraphs we have discussed our plan of attack in explaining PAL's semantics, ignoring all mention of syntax. This is quite appropriate, since we feel that it is with the semantic problems that we should wrestle, the syntactic problems being more tractable. Nonetheless, a few words about our approach to syntax are appropriate here. In a PAL expression such as

$$x + y * z \qquad (1.4\text{-}1a)$$

there are two sorts of issues. From the point of view of semantics, we are concerned with such questions as the values of the three variables used and the meanings of the operators "+" and "*". Syntactically, we are concerned with parsing the expression: To which of

$$x + (y * z) \qquad (1.4\text{-}1b)$$
$$(x + y) * z \qquad (1.4\text{-}1c)$$

is it equivalent? Our common cultural background leads us to hope it is the first of these rather than the second (it is), but the specifiers of a language must make clear in such cases just which parsing is correct.

We take two different approaches to this problem. In the PAL Manual a notation for syntactic description (Backus Naur Form -- BNF) is described and used extensively. BNF certainly provides an adequate tool for answering such questions. In these notes, however, we choose to give little attention to such syntactic issues, instead taking the attitude that the problem arises from the need to communicate in only one dimension. Consider now Figure 1.4-1, which shows 2-dimensional representations of several expressions:

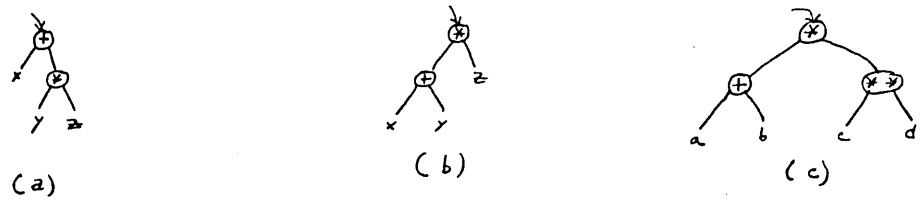


Figure 1.4-1: Tree Form of Some Expressions

It should be quite clear that (a) in the figure corresponds to (1.4-1b) and (b) to (1.4-1c). Indeed, it is not possible to illustrate in the tree form the problem that was raised by (1.4-1a) -- in trees the parsing is explicit, while in (1.4-1a) it is implicit. Thus we regard syntax as providing us with rules for flattening trees into one dimension, with a minimum number of parentheses. For example, Figure 1.4-1(c) can be flattened into

(a + b) * c ** d (1.4-1d)

where the operator "**" stands for exponentiation.

Our point of departure in these notes in explaining PAL is to assume that our input to the evaluating mechanism is a PAL program in the form shown in Figure 1.4-1, rather than in the "flat" form of equations (1.4-1). Thus whereas a BNF description of infix operator expressions such as these requires four or five lines (as in section 3.3 of the PAL Manual), we need only one "tree equation", as in Figure 1.4-2:

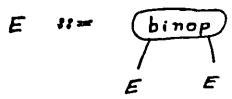


Figure 1.4-2: Tree Syntax Definition

We refer to this form of syntax description as abstract syntax. The intent of the figure is that one form of an "E" (i.e., an expression) is a binop node whose two sons are E's. (We use "binop" to represent the class of infix binary operators.) The BNF equivalent to the figure is

<E> ::= <E> <binop> <E> (1.4-2)

but this is ambiguous, not giving an answer to the question about equation (1.4-1a). (Syntactic ambiguity is discussed in section 1.2 of the PAL Manual.) Thus by dealing with tree syntax, we avoid all questions of parsing and can ignore the process of translating equation (1.4-1a) to the form of Figure 1.4-1a. It is not that these questions are uninteresting or unimportant, but that the focus of these notes is on other problems.

Chapter 2

CONCEPTUAL FOUNDATIONS

In Chapter 1 we viewed a program as the specification of a (perhaps complicated) function in terms of simpler basic functions. A more precise characterization of our view of computation is provided by the following definitions:

1. An algorithm is the specification of a transformation on abstract objects, the specification to be in terms of functions that need no further specification.
2. A programming language is a set of conventions for communicating algorithms.
3. A program is a representation of an algorithm in a programming language.

Note how the definition of algorithm follows from our claim that we regard computation as having to do with abstract objects. The "communication" of definition 2 might be between man and man or between man and computer, or even between two computers.

Insofar as our immediate needs are concerned, these definitions serve primarily to alert us to what information a valid description of a programming language must provide. Specifically, any such description must afford answers to the following questions:

A. What is the universe of discourse of the language? That is, what are the properties of the abstract objects, including various functions, with which the language deals?

B. Which abstract objects and transformations can be referred to directly, and what are the conventions for doing so? That is, which abstract objects have names in the language?

C. What facilities does the language afford for building complicated transformations out of simpler ones?

D. What are the grammatical conventions of the language?

Clearly, modifying the answer to any one of these questions while leaving the other answers invariant modifies the language. Moreover, the questions are "linguistically orthogonal", in the sense that modification of one answer can not be nullified by modification of the others. Thus the questions define at least one set of coordinates suitable for language description.

In this chapter we are be concerned primarily with the first three of these questions. Accordingly, throughout this chapter we focus less upon PAL than upon the system of mathematical logic on which much of PAL is based. This system, the λ-calculus of Church and Curry, may itself be viewed as a language, albeit one with particularly simple syntax and semantics. As we proceed, we indicate how PAL reflects the linguistic constructs of the λ-calculus.

2.1 Universe of Discourse

Historically, programming languages have usually been designed to deal principally (although not exclusively) with some specific class of abstract objects. For example:

FORTRAN and ALGOL programs manipulate numbers.
COMIT and SNOBOL programs manipulate strings.
LISP and IPL programs manipulate lists.

The current trend seems to be towards "universal" languages, such as PL/I, which include in their universe of discourse many types of data.

PAL, like PL/I, deals with many types of abstract objects (hereafter abbreviated "obs"). Specifically, we are concerned with disjoint subsets of obs called truthvalues, strings, numbers, tuples and functions. Numbers, in turn, are partitioned into the classes integer and rational.

Our objective in studying these sets is twofold. First, we seek thorough understanding of the obs themselves, as a prerequisite to writing non-trivial programs. One cannot hope to build up a complicated transformation without detailed knowledge of the basic constituents from which it is to be composed.

Second, we seek understanding of a methodology for defining classes of obs, an understanding which ultimately should afford insight into certain questions of language extensibility. We want to study how a language may be modified by incorporating a new class of obs into its universe of discourse, or by excising a class already included, without deranging the language as a whole.

Plan of Attack: Our objective in this section is to specify the universe of discourse of PAL -- to answer question A on page 2.1-14. In doing so, we find it convenient to answer question B also, in that we give the PAL names for the objects we describe. Note that A suggests that "basic functions" are part of the universe of discourse. A complete specification of the set of integers must include, for example, specification of the basic functions such as addition and multiplication that operate on them.

In succeeding subsections we give the properties of truthvalues, strings, integers, rationals and tuples. In each case we describe both the set of abstract objects and also the relevant basic functions. We give detailed consideration to functions in Sections 2.2 and 2.3.

Mathematical Preliminaries

Since some of the mathematical conventions and notation used in the rest of this chapter may not be familiar to all readers, we present now the ideas needed. Many of the less familiar ideas are used in only one place in these notes. At that point there usually appears a suggestion that the reader study again the relevant part of this section. On first reading of these notes it is probably appropriate just to scan this section, planning to study it again as needed.

Sets: A set is a collection of objects. If the object m is in the set S, we write

$$m \in S$$

while if not we write

$$m \notin S$$

In general in this section, we use upper case letters for sets and lower case letters for members of sets.

We use two conventions to denote explicit sets. For example, we might write

$$\{a, b, c, d\}$$

to denote that set containing the four objects listed. We could write then

$$\{a\}$$

to denote the set consisting of the single element a. Note that this set is different from a. We sometimes write

$$\{x \mid P(x)\}$$

to indicate the set of all objects x such that P(x) is true. For example, the two writings

$$\{x \mid (x \text{ is an integer}) \text{ and } (x > 0) \text{ and } (x < 5)\}$$
$$\{1, 2, 3, 4\}$$

each denote the same set. We write ϕ to denote that set containing no elements at all.

If R and S are sets, then the set $R \cup S$ is the union of R and S: the set of all objects that either are in R or are in S or are in both. That is,

$$R \cup S = \{x \mid (x \in R) \text{ or } (x \in S)\}$$

Also, $R \cap S$ is the intersection of R and S: the set of all objects in both R and S. Thus we have

$$R \cap S = \{x \mid (x \in R) \text{ and } (x \in S)\}$$

For any set S, it is true that

$$\phi = S \cap \phi$$
$$S = S \cup \phi$$

Saying that $R \cap S = \phi$ is equivalent to saying that R and S have no elements in common.

If all members of set R are also members of S, we say that R is a subset of S or that S contains R, and write

$$R \subset S$$

It is true that

$$\phi \subset S$$

for all sets S.

Ordered Pairs: We are on occasion concerned with ordered pairs of objects. We write

$$\{ a, b \}$$

to denote that ordered pair whose first element is a and whose second element is b. The question of whether there is a correspondence between ordered pairs and PAL's 2-tuples is not dealt with at this time. We confine our use of the ordered pair, which is a mathematical idea, to the first part of this chapter before the introduction of tuples.

The set of ordered pairs with first element in set A and second element in B is written

$$A \circledast B$$

so that we have

$$A \circledast B = \{\{a, b\} \mid (a \in A) \text{ and } (b \in B)\}$$

Of course the two sets may be the same set. For example, if N is the set of integers, then $N \circledast N$ denotes the set of ordered pairs of integers.

Relations: A relation on a set A is a set R such that

$$R \subset A \circledast A$$

Thus R is a subset of all possible ordered pairs of objects from A. We then write

$$a \; R \; b$$

to mean that

$$\{a, b\} \in R$$

For example, the relation "less than" on numbers is the set

$$\{\{x, y\} \mid (y-x) \text{ is positive}\}$$

We are interested in three possible properties a relation might have. A relation R on a set A is said to be

reflexive	if $x \; R \; x$;
symmetric	if $x \; R \; y$ implies $y \; R \; x$; and
transitive	if $x \; R \; y$ and $y \; R \; z$ imply $x \; R \; z$;

for all x, y and z in A.

An equivalence relation is a relation that is reflexive, symmetric and transitive. For example, the relation "is congruent to" is an equivalence relation on the set of triangles. The

subset relation is not an equivalence relation, since it is reflexive and transitive but not symmetric.

Functions: If D and C are sets, we say that f is a function from D to C if there is a subset S of D such that, for each $x \in S$, f specifies a unique element $f(x) \in C$. This element is called the value of f at x. The set D is the domain of f, C the codomain of f, and S the domain of definition of f.

Note that a function has three components: a domain, a codomain, and a rule. However, we are willing to say of two functions that they are equal if we can show that they have the same domain of definition and the same rule. That is, if f and g each have domain of definition S, we require that $f(x) = g(x)$ for all $x \in S$. Since they have the same domain of definition, there can be no point at which one is defined and not the other.

The range of a function is the set of values it takes. It is a subset of the codomain. For example, the range of $f(x) = x^2$ on integers is the set

$$\{0, 1, 4, 9, 16, 25, \ldots\}$$

Suppose that f is a function with domain of definition D and range R, and that $S \subset D$. Then by f(S) we mean the set

$$\{y \mid \text{there is an } x \in S \text{ such that } y = f(x)\}$$

This is the set of function values corresponding to elements of S, and it is always a subset of R. For example, for the squaring f above

$$f(\{1, 2, 3\}) = \{1, 4, 9\}$$

We usually specify functions by giving the domain and a rule, trusting that the nature of the rule makes clear the domain of definition. The range is thereby defined, and a codomain follows. For example, consider the function divide with domain pairs of rational numbers. Clearly the domain of definition is the set of pairs with second element non-zero, and the range is the rationals.

A function is said to be total over a domain if that domain is also its domain of definition. A partial function over a domain is one that is not total over that domain. Division is a partial function over pairs of rationals.

Consider the function f defined by

$$f(x) = x^2 - 13x + 8$$

with domain integers. Clearly f is total over that domain. The codomain of f is integers, but specifying the range is awkward. Although it seems more useful to specify the range than the codomain, we frequently find it convenient to specify the codomain. In such cases, it seems desirable to specify the "smallest" convenient codomain.

The null function is that function whose domain of definition is empty.

A constant function, or a function of no argument, is one whose domain of definition is the set $\{\phi\}$ where ϕ is the empty set. (Such a function is not the null function.) We write $f() = 3$ to indicate that constant function f whose value is 3, suggesting by the empty parentheses a function of no arguments.

Functionality: If D and C are sets, we write

$$D \to C$$

to designate the set of all functions with domain D and codomain C. We can then write

$$f \in D \to C$$

or

$$f: D \to C$$

to indicate that the domain of f is D and its codomain is C. Thus one might write

$$f \in \text{integer} \to \text{integer}$$

to indicate that f is a function whose domain and codomain are each the set of integers. Use of the mark "$\in$" is justified by our statement that the arrow notation defines a set of functions. Such a specification of domain and codomain is called the functionality of a function.

We can indicate that f is a constant function by writing

$$f \in \{\phi\} \to S$$

where S contains the value.

A few more examples may help to clarify this concept. Consider the function Addn: $N \otimes N \to N$ defined by

$$\text{Addn } (x, y) = x+y$$

where N is the set of all integers. (Note that we have defined Addn by giving its domain, codomain and rule.) Suppose we have another function, Addr: $R \otimes R \to R$, defined by

$$\text{Addr } (x, y) = x+y$$

where R is the set of rationals. Finally, suppose we want a single function Add which "works" on either pairs of integers or pairs of rationals. What is its functionality? Its domain is clearly

$$(R \otimes R) \cup (N \otimes N)$$

and its codomain

$$R \cup N$$

so it would be correct to write

$$\text{Add} \in (R \otimes R) \cup (N \otimes N) \to (R \cup N)$$

but this is misleading, since what is wanted is to suggest that Add of two integers never yields a rational, or vice versa. We thus write

$$\text{Add} \in (N \otimes N \to N) \wedge (R \otimes R \to R)$$

Specifically, we say

$$f \in (A \to B) \wedge (C \to D)$$

only if $A \cap C = \phi$, and we mean that

$$f \in (A \cup C) \to (B \cup D)$$

and that $f(A) = B$ and $f(C) = D$. Such a function is called a polymorphic function. In most programming languages the usual arithmetic operators are polymorphic. The concept is uncommon in conventional mathematics.

Note two things: The arrow notation defines a set of functions, and it is a set that goes to the right of the arrow. Thus it seems permissible to write

$$f \in N \to (N \to N)$$

Presumably such an f is a function whose domain is integers and whose values are functions from integers to integers. Such a function is called a curried function (after the mathematician H. B. Curry). Although it is awkward to write a curried function using conventional mathematical notation, it is easy to write one in PAL. Curried functions play a crucial role in what follows.

Other Definitions Concerning Functions: We collect here some other definitions which we have use of in what follows.

A predicate is a function whose range is the set {true, false} of truthvalues. For example, the function f defined by

$$f(x) = x > 0$$

on integers is a predicate, returning true when its argument is a positive integer and false for zero or negative integers as argument.

A function is said to be onto its codomain S if S is its range, while otherwise the function is into. For example, the function $f(x) = x^2$ from integers is into the non-negative integers, but not onto them.

A function f from D to C is said to be many to one if there are x, y ∈ D with $x \neq y$ such that $f(x) = f(y)$. That is, f maps several points onto the same value. We say f is one to one if it is not many to one. For such a function, $f(x) = f(y)$ implies that $x = y$.

A function g is a functional extension of f if g is defined at all points at which f is, and if their values are equal at those points. Every function is (vacuously) a functional extension of the null function.

Suppose we have sets F and G and functions $f: F \otimes F \to F$ and $g: G \otimes G \to G$, and that f and g are total over the indicated domains. Consider a function $\theta: F \to G$ with θ total over F. Then θ is said to be an isomorphism from (f, F) to (g, G) if θ is one to one and onto, and if

$$\theta [f(x, y)] = g[\theta(x), \theta(y)]$$

for all x, y ∈ F. Note that an isomorphism requires both the sets F and G and also the functions f and g on them. An example of an isomorphism is given on page 1.2-4.

The closure of a function f over set S is a set defined as follows: Let $S_0 = S$, and define

$$S_{n+1} = S_n \cup f(S_n)$$

for $n = 1, 2, \ldots$. Then an element is in the closure if it is in at least one of the S_n. For example, if

$$f(x) = x+1$$

then the closure of f over the set {0} is the set of non-negative integers. Similarly, the closure of division over pairs of integers is the set of rationals.

Structure Definitions: A structure definition provides a technique to specify in a single bundle many properties of certain kinds of objects. We have already seen one such definition (on page 1.3-7), and we repeat it here, slightly changed to meet our present needs:

An algebraic expression (algex) is a structured object. It is either an
- atom, or it is a
- binop, in which case it has an
  - operator, which is in the set B, and a
  - left-part, which is an algex, and a
  - right-part, which is an algex, or it is a
- unop, in which case it has an
  - operator, which is in the set A, and an
  - operand, which is an algex.

(The indenting scheme used has the obvious significance.) When we write a structure definition such as this one (and we write many of them), we are saying certain specific things about the class of algex's. In particular, we are saying that given an algex we can tell whether it is an atom, a binop or a unop. That is, we are claiming the existence of certain set-membership predicates. Second, if we have an algex which is a binop, we are able to select out any one of its three parts; just as we can select out either of a unop's two parts. That is, we have suitable selectors. Finally, given a member of the set B and two algex's, we can construct an algex which is a binop; or we can build a unop from a member of A and an algex. That is, we have suitable constructors.

The important point is that writing the structure definition implies the existence of the relevant predicates, selectors and constructors. Further, a definition such as that of algex implies that there can be no other kind of algex than the three kinds listed. Suppose that one wanted to prove a theorem about algex's: The definition given makes it clear that no more is needed than to prove that the theorem holds for atoms, binops and unops.

For example, consider the differentiation algorithm given in English on page 1.3-9. The first line implies use of the predicate isAtomic (or some such); the specification of L, Op and R requires use of selectors; and use of a constructor is implied by a writing such as "the expression whose left operand is L1 and whose operator is + and whose right operand is R1".

An important aspect of a structure definition has to do with representation. To be acceptable, any proposed representations of algex's must permit implementation of the needed predicates, selectors and constructors. The reader should satisfy himself that the PAL representation used in section 1.3 does meet these requirements, as does the tree representation suggested in Figure 1.3-1, page 1.3-7.

Meta-language: Throughout these notes we make a sharp distinction between abstract objects and PAL's conventions for denoting them. The reason is that the obs are invariant to any particular choice of naming conventions. For example, the properties of the positive integers would be unaffected by a decision to use Roman instead of Arabic numerals in PAL, and the properties of strings would be unaffected even were we to excise from PAL every linguistic facility for dealing with them.

In order to talk about the obs independently of PAL, we need a meta-language: a language for talking about a language. As our meta-language we use ordinary technical English, supplemented by conventional mathematics. But PAL itself is modelled in large part on conventional mathematical notation, so that some special device is frequently necessary to guarantee the distinguishability of PAL and conventional expressions. We resolve the difficulty by agreeing in the latter case always to refer to obs by "meta-names", names that are not allowable in PAL's syntax. Specifically, meta-names always are boldface (which in typescript, as in the present edition of these notes, is indicated by wavy underlining), but which accord with PAL syntax in other regards. Thus we might write

2

in PAL to denote the ob 2. In text (as opposed to on a separate line) we use double quotes to set off PAL names, as "2". We sometimes use underlines instead for names only one character long: 2.

With these conventions understood, we now proceed to discuss each of the classes truthvalues, strings, integers, rationals and tuples in turn. For each class we give first an intuitive discussion, next a formal definition, and finally details on the relation to PAL.

Truthvalues

The class of truthvalues contains precisely two obs: true and false, with the properties the reader should expect them to have. We are interested in the usual three functions on truthvalues: And, Or and Not.

Although in successive sections we use to advantage the axiomatic method to define properties of sets such as the integers, that method seems inappropriate for the truthvalues: Since the set contains only two members, it is simpler just to list all of the properties we want. The power of the axiomatic method comes into evidence in defining infinite sets, such as the integers.

Formal Definition: Rather than define the three usual functions mentioned earlier, we define the single function Nor. We then define the other three functions in terms of it. We have

Definition: The set of truthvalues contains the two obs true and false.
There is a function Nor: truthvalue → truthvalue such that
Nor (true, true) = false
Nor (true, false) = false

$$\underline{\text{Nor}}\ (\underline{\text{false}},\ \underline{\text{true}}) = \underline{\text{false}}$$
$$\underline{\text{Nor}}\ (\underline{\text{false}},\ \underline{\text{false}}) = \underline{\text{true}}$$

We have specified Nor entirely: Since there are only two distinct truthvalues, there are only four distinct pairs of truthvalues; and we have shown what value Nor has for each possible argument it might get. (This technique is clearly unsuitable for, say, defining addition of integers.) A compact way to provide the same information is in tabular form, as

x	y	Nor (x, y)
f	f	t
f	t	f
t	f	f
t	t	f

(Here we have written t and f for true and false, respectively.) Such a table is called a truth table.

We want now to define And, Or and Not to have the properties one expects. We can do so either by giving a tabular definition or by defining them in terms of Nor, whose properties are known. We do the latter, and then show that the functions do as expected.

Definition: The functions Not, And and Or, with functionalities

$$\begin{aligned} \underline{\text{Not}} &\in \text{truthvalue} \to \text{truthvalue} \\ \underline{\text{And}} &\in \text{truthvalue} \otimes \text{truthvalue} \to \text{truthvalue} \\ \underline{\text{Or}} &\in \text{truthvalue} \otimes \text{truthvalue} \to \text{truthvalue} \end{aligned}$$

are defined as follows:

$$\begin{aligned} \underline{\text{Not}}(x) &= \underline{\text{Nor}}(x, x) \\ \underline{\text{And}}(x, y) &= \underline{\text{Nor}}(\underline{\text{Not}}(x), \underline{\text{Not}}(y)) \\ \underline{\text{Or}}(x, y) &= \underline{\text{Not}}(\underline{\text{Nor}}(x, y)) \end{aligned}$$

As is reasonable, we have used the first definition in the next two. That these three functions have the usual properties is revealed by the following truth table:

x	y	Nor(x, y) [a]	Nor(x, x) [b]	Not(y) [c]	Nor(b, c) [d]	Not(a) [e]
f	f	t	t	t	f	f
f	t	f	t	f	f	t
t	f	f	f	t	f	t
t	t	f	f	f	t	t

Column (a) repeats the definition of Nor. Column (b) is the definition of Not, and it clearly is the expected negation of x. Column (c) is the same for y. Column (d) is the Nor of (b) and (c) as in the definition of And, and reveals the expected transformation; and column (e) does the same for Or.

Relation to PAL: PAL provides identifiers "true" and "false" as the names of true and false, respectively; and the PAL name for Not is "not". Instead of providing functions for And and Or, PAL provides infix operators for them. Thus if E and F are any PAL expressions, then the PAL expression

E & F (2.1-1a)

denotes the same object as does

And (E, F) (2.1-1b)

and

E or F (2.1-2a)

denotes the same ob as does

Or (E, F) (2.1-2b)

where in each case E and F are the obs denoted by E and F, respectively. Since And and Or have been defined only for the case in which the argument is a pair of truthvalues, each of these expressions is undefined unless both E and F denote truthvalues. This point is important and deserves amplification.

Saying of a construct that it is undefined means neither more nor less than that we have not defined it. Thus

Not (3)

is undefined because we have not defined the value returned by Not when its argument is an integer. Another way to look at it is that 3 is not in the domain of definition of Not, so this expression must be undefined. The implication of this is that the meaning of a PAL expression such as

not 3

is undefined. One might hope that execution of a program containing such an expression would result in an appropriate error message, but an implementor of PAL is free to do anything at all and still have met the formal definition. (All existing PAL implementations give a good diagnostic.)

Note that no PAL name has been provided for Nor. Since

°Nor (x, y) = And (Not (x), Not (y))

(a fact that the reader should verify with a truth table), one may write in PAL

def Nor (x, y) = (not x) & (not y) (2.1-3)

Names of variables in PAL consist of letters, digits and the underscore character. Names two or more characters long consisting of all lower-case letters are reserved words. Thus Nor, x and y are variables in this program and not is a reserved word. We use the term functor for a name (such as "&", "or" or "not") which, though not a variable, nonetheless denotes a function. This program defines the function Nor of two variables x and y, in the obvious manner. x and y are dummy variables, or formal parameters.

Strings

The class of obs called strings may be thought of as sequences of zero or more characters chosen from a specified alphabet. The length of a string is the number of characters it contains. For example,

AaB?2 (2.1-4)

represents a string of length 5 with characters chosen from an alphabet containing {A, a, 2, ?, B} as a subset. The meta-name of the string represented by (2.1-4) is 'AaB?2', and its PAL name is

'AaB?2'

Two strings are said to be equal if and only if their representations are the same.

We think of the stem of a string of length k>0 as the leftmost character in its representation, and of the stern of a string of length k>0 as that string of length (k-1) obtained by deleting the stem. Thus the stem of (2.1-4) is the single character "A", and its stern has PAL name

'aB?2'

Finally, we are interested in concatenating two strings, say of length m and n, to produce a string of length (m+n). For example, the string of (2.1-4) may be thought of as the result of concatenating the first and second elements of any of the six following ordered pairs, in which "Λ" denotes the string of length zero.

Λ	, AaB?2
A	, aB?2
Aa	, B?2
AaB	, ?2
AaB?	, 2
AaB?2	, Λ

(2.1-5)

Postulates: The substance of the preceding informal discussion is encapsulated in the following

Definition: A string system $[\mathcal{S}_L, A]$ over a finite set L is composed of a set and a function A, such that the following hold:

1. Each member of L is a member of $\mathcal{S}_L$.
2. There is an element $s_o \in \mathcal{S}_L$ which is not an element of L. The length of s_o is zero. Let $\mathcal{S}_L^\circ$ be the set of all elements of $\mathcal{S}_L$ other than s_o.
3. The function $A: L \otimes \mathcal{S}_L \to \mathcal{S}_L^\circ$ is one-to-one and onto. For every element $x \in L$, it is true that
$$A(x, s_o) = x$$
4. Define the sets S_n as follows:
$$S_o = [s_o]$$
$$S_{n+1} = S_n \cup A(L \otimes S_n)$$
Then the set $\mathcal{S}_L$ is defined as follows: An object x is a member of $\mathcal{S}_L$ if there is some n such that $x \in S_n$.

The effect of these definitions is to define the properties of the set $\mathcal{S}_L$ (the set of strings) and the function A. The set $\mathcal{S}_L$ depends on the set L -- the alphabet of the string system, in that changing L changes the strings defined. Property 1 says that each letter of the alphabet is a string. Property 2 says that there is at least one string which is not a letter: the empty string, or string of length zero.

Property 3 gives the functionality of A and makes the important points that it is one to one and onto. This means that for every $x \in \mathcal{S}_L$, there are unique m and n such that

$$A(m, n) = x$$

That m and n exist follows from onto, and that they are unique follows from one to one.

Property 4 defines $\mathcal{S}_L$ in a manner similar to the definition of closure given earlier, on page 2.1-19. It would have been pleasant to say that $\mathcal{S}_L$ is the closure of A over $(L \cup \{s_o\})$, but doing so would not have been correct since A's domain is pairs and our definition of closure is inappropriate.

We now define two functions, making use of our earlier discussion of Property 3.

Definition: There are functions M and N with

$$M: \mathcal{S}_L^\circ \to L$$
$$N: \mathcal{S}_L^\circ \to \mathcal{S}_L$$

such that, for every $x \in \mathcal{S}_L^\circ$,

$$A[M(x), N(x)] = x$$

That M and N are well-defined (i.e., unique) follows from the fact that there exist unique m and n such that $A(m, n) = x$.

Relation to PAL: The PAL name for a string is called a quotation, and is written by writing the characters of the string between single quote marks. The alphabet L over which strings are defined is given in the PAL Manual, in Section 2.1. Certain characters in the PAL alphabet, such as new line and quote, are provided with special names to facilitate quoting them. (Details are in

Section 2.4 of the Manual.) The PAL name for s_o is '', two adjacent quote marks.

PAL provides "Stem" and "Stern" as the names of M and N, and the function "Conc" defined below. In addition, PAL provides the infix functors "eq" and "ne" for determining if two strings are equal.

It often happens that the primitive functions which are convenient for writing postulates do not coincide with the most convenient set of functions for programming. Consider the problem of determining whether or not two strings are the same. That objects are distinguishable implies the existence of a predicate, say IsEmpty, with functionality:

$$\text{IsEmpty} \in \text{string} \to \text{truthvalue}$$

such that IsEmpty returns true when applied to s_o and returns false when applied to any other string; as well as another predicate, say Eq, such that

$$\text{Eq} \in \text{L} \otimes \text{L} \to \text{truthvalue}$$

and Eq returns true when both arguments are the same character, and false if they are different characters. As it happens, PAL's designers have not provided functions corresponding to IsEmpty and Eq, but have provided instead a more powerful predicate for determining the equality of arbitrary strings. But assume that this were not so, and that "Eq" were the PAL name for the function Eq and "IsEmpty" for IsEmpty. Then the arbitrary equality predicate could be defined by the PAL program

```
def rec Equal (x, y) =
      IsEmpty x -> IsEmpty y
    | IsEmpty y -> false
    | Eq (Stem x, Stem y) -> Equal (Stern x, Stern y)
    | false
```

This program deserves some comment. The PAL conditional expression

```
B -> E | F
```

has the value of E if B is true and of F if B is false. (It is undefined otherwise.) Thus

```
IsEmpty x -> IsEmpty y | F
```

has the value true if both x and y are empty, the value false if x is empty and y if not, and the value of F otherwise. Thus the effect of the first two lines of the body of the function (the part after the "=") is to return true if both x and y are empty, false if only one of them is, and to go to the next line if neither is. Here we ask if the first character of x (its Stem) is the same as the first character of y. If not, we return false, while if so we call "Equal" recursively to compare the Stern of x with the Stern of y. The punctuation "rec" indicates a recursive function -- one that calls itself. The reader should convince himself first that "Equal" returns false if one but not both of its arguments is s_o. Given that neither argument is s_o, it returns false unless the stems of its arguments are equal. Finally, should this test also be passed, the value of "Equal" depends upon equality of the sterns of its arguments. Clearly, the procedure always terminates for finite arguments, and equally clearly Stem and Stern are applied only to non-empty strings.

A second example concerns the function A, which affords only a primitive version of concatenation. Presumably we would prefer a function, say Conc, for concatenating strings of arbitrary length, such as those in (2.1-5). Actually, Conc rather than A is the string concatenation operation provided in PAL. But were the situation reversed and "Adjoin" the PAL name for A, we could remedy the inconvenience by the program

```
def rec Conc (x, y) =
      IsEmpty x -> y
    | Adjoin [Stem x, Conc (Stern x, y)]
```

Rather than provide the function "Equal", PAL provides the infix binary functor "eq". The PAL expression

E eq F

has value true if E and F denote the same string. (We see later that it is also true if E and F denote the same integer or the same rational. That is, "eq" is polymorphic.) The functor "ne", for not-equal, is also provided.

Finally, we note that PAL provides no function at all for determining the length of a string. The definition in PAL of an appropriate function, say "Length", is left as an exercise.

Integers

Two classes of numbers, called integer and rational, are of interest. We consider rationals in the next subsection, considering here the integers and their properties.

While in the previous two sections we said all there was to say about the sets of truthvalues and strings, in this and the next section we leave out many of the details. Our reasons are three-fold:

- A complete axiomatic treatment of the integers or of the rationals is not only lengthy but is also available in many elementary math books. The other classes of interest to us are peculiar to PAL and are thus not elsewhere defined.
- We have already made the intellectual point that axiomatic treatment of a universe of discourse is possible, a point we make again in the treatment of tuples.
- The reader is (presumably) already familiar with integers and rationals, so there seems little educational value to a formal treatment of them.

We therefore elect to give in this section only some of the needed axioms, and in the next subsection none at all.

Postulates: What we want to define is the set of integers, both positive and negative, as well as the operations, addition, subtraction, multiplication, division and exponentiation. Instead we define by postulate only the non-negative integers -- the natural numbers -- and give PAL definitions for addition and multiplication on them.

Definition: The set of natural numbers is a set $\underline{N}$ such that

1. There is a unique element $\underline{n}_0 \in \underline{N}$. Let $\underline{N}_0$ be the set of all elements of $\underline{N}$ other than $\underline{n}_0$.
2. There is a total function Succ: $N \rightarrow N_0$ which is one-to-one and onto the set $\underline{N}_0$.
3. The set $\underline{N}$ is the finite closure of Succ over $\{n_0\}$.

It should be clear that the $\underline{n}_0$ and Succ of this definition correspond respectively to zero and to the function which adds one to an integer.

In property 2, the words "total", "one-to-one" and "onto" are all key. (These ideas are all defined in the early part of this chapter.) "Total" means that every integer has a successor; "one-to-one" means that there are no two distinct integers that have the same successor; and "onto" means that every natural number other than zero is the successor of some natural number. As we are often interested in the predecessor of a natural number, we provide the

Definition: There is a function Pred: $\underline{N}_0 \rightarrow \underline{N}$ such that, if Pred(m) = n, then m = Succ(n).

That Pred is well defined follows from our discussion above of property 2.

The arithmetic transformations on the natural numbers follow from the postulates. For example, if "0", "Succ" and "Pred"were the PAL names for $\underline{0}$, Succ and Pred, respectively; and if "Zero" were the PAL name of a predicate whose value is true when applied to $\underline{0}$ and false when applied to any other integer; then we could define functions for adding and multiplying natural numbers by the PAL programs

```
def rec  Add(x, y)  =  Zero y -> x | Add(Succ x, Pred y)          (2.1-9a)

def rec Mult(x, y)  =  Zero y -> 0 | Add(x, Mult(x, Pred y))      (2.1-9b)
```

The reader should convince himself that these functions work so long as x and y are non-negative integers. (How can you be sure the recursion will not loop forever?)

To complete the formal definition, one should now provide postulates for the negative integers, and define subtraction. Division and exponentiation could then be defined by PAL programs. The interested reader may find axiomatization of the negative integers in any suitable mathematics text, and the PAL functions are left as exercises.

Relation to PAL: PAL provides the following infix binary functors:

+	addition
-	subtraction
*	multiplication
/	division
**	exponentiation
le	less-than-or-equal-to
<	less-than
ge	greater-than-or-equal-to
>	greater-than
eq	equal
ne	not-equal

The first five functors designate functions that have functionality

$$\text{integer} \otimes \text{integer} \rightarrow \text{integer}$$

and the last six may be assumed to designate functions with functionality

$$\text{integer} \otimes \text{integer} \rightarrow \text{truthvalue}$$

(We see later that these functors are polymorphic.) All of the functors work as one might expect. Division by zero is not defined, nor is exponentiation for certain cases such as "0**0". PAL also provides "+" and "-" as prefix unary functors, the latter designating negation. Note that the value of E/F is always an integer. We do not here specify whether truncation or rounding takes place if the division is not exact.

The PAL name for a number is called a numeric, and the PAL name for an integer is called an integer numeric.

The only integers that PAL provides names for are the non-negative integers, and these in the usual way. Further, each non-negative integer has many names, both "2" and "0002" being names of 2. Although the negative integers do not have names, one may easily write an expression whose value is any negative integer desired: "0-5" denotes -5. The decision not to provide names of the negative integers is an arbitrary one, and changing it would not change the expressive power of the language.

Rationals

Mathematically, the set of rationals is the closure of the integers under division. Postulates for the rationals are not provided here, for the reasons already given. The eleven functors listed on this page also work on rationals. The functors "+", "-", "*" and "/" have functionality

$$(N \otimes N \rightarrow N) \wedge (R \otimes R \rightarrow R)$$

(Here and below we use N to abbreviate integer and R to abbreviate rational. The mark $\wedge$ is defined on page 2.1-18.) These functors designate functions which return an integer when applied to a pair of integers and return a rational when applied to a pair of rationals. It is incorrect to apply

one of these functors to a rational and an integer: The PAL expression

```
3 + 4.2
```

is erroneous.

For exponentiation we have the restriction

$$** \in (N \otimes N \rightarrow N) \wedge (R \otimes N \rightarrow R)$$

Thus the exponent must be an integer. We would be admitting a mathematical inconsistency into the language were we to permit rational exponents, since it would then be possible to write expressions denoting obs not in the universe of discourse. For example, the expression "2 ** 0.5" would (presumably) denote the square root of 2, an irrational. Had we used reals instead of rationals, this problem would have been avoided. However, rationals are closer by far to what actually goes on in computers, and introduction of reals leads to many other problems. In some programming languages, the term real or floating point is used for essentially the same class of obs that we call rational.

The six relational functors are polymorphic. Each of "<", "le", ">" and "ge" designates a function with functionality

$$(N \otimes N \rightarrow T) \wedge (R \otimes R \rightarrow T)$$

where T is the set of truthvalues. The two functors "eq" and "ne" are even more polymorphic, designating functions with functionality

$$(T \cup S \cup N \cup R) \otimes (T \cup S \cup N \cup R) \rightarrow T$$

where S is the set of strings. Thus one may write in PAL

```
2 eq 'abc'
```

Such an expression denotes true if both operands belong to the same class (T or S or N or R) and have the same value. Each of the following PAL expressions is defined and denotes false:

```
2 eq 2.0      2 eq '2'      true eq 'true'
```

Names are provided in PAL for only these rationals of the form P/(10**Q), where P is a positive integer and Q is any integer. The name of a rational consists of one or more digits followed by a decimal point followed by one or more digits. Power of ten notation, such as the 13.2E5 of Fortran or PL/I, is not provided; and names such as "1." and ".1" are not acceptable. (Use instead "1.0" or "0.1".) Note that "2" denotes the integer 2 and "2.0" the rational 2.0, and these are two different obs. This point is discussed further below.

The user of PAL should be cautious in his use of rationals, since most implementations do not store rationals exactly. For example, the rational with PAL name "0.2" would probably be approximated in a computer by a binary floating point number, so that the PAL expression

```
(5.0 * 0.2) eq 1.0
```

might denote false. For this reason one implementation of PAL (on the TX-2 computer at Lincoln Laboratory) uses "fuzzy" tests for the relational functors when their arguments are rationals. That is, if a and b are rationals which are not too close to zero, then the PAL expression

```
a eq b
```

is equivalent mathematically to something like

$$| (a-b) / M | < \delta$$

where M is the maximum of |a| and |b|. (The case in which both a and b are close to zero is treated specially.) The constant δ is implementation dependent.

Although rationals are usually only approximated, it is safe to assume that any sensible language implementation provides precise integer arithmetic so long as the magnitude of the operands does not exceed some large (machine-dependent) threshold. On the other hand, one should look askance at programs whose successful operation depends critically upon precision of rational

arithmetic.

It is well to remark that imperfections due to the finiteness of machines are not peculiar to numbers alone. There is also a limit on the length of a string or even of a program. We view such difficulties as implementational rather than linguistic in nature and give them little attention in this document.

Integers and Rationals: From our conceptual viewpoint there is as much distinction between the abstract classes integer and rational as there is between either one of them and the class string. Although it is convenient in PAL to use the same functors -- such as "/" for "division" -- in conjunction with both integers and rationals, the transformation specified by the functor depends upon the type of its arguments. (The "type" of an ob is the class to which it belongs.) For example, in PAL

a/b

denotes the ordinary quotient if a and b denote rationals, whereas it denotes the integer part of the ordinary quotient if a and b denote positive integers. Finally, the expression is undefined if a denotes a rational and b an integer, or vice versa.

In many languages (PL/I, for one) a "transfer function" is invoked automatically for certain types of arguments. For example, in PL/I an integer argument to "+" would be converted to type rational if the other argument were rational. Type transfers in PAL, on the other hand, are never automatic. Instead, the basic transfer functions

"itoR" (for integer to rational)
"Rtoi" (for rational to integer)
"Stoi" (for string to integer)

which PAL provides must be invoked explicitly by the programmer when needed. These functions are described in Section 3.4 of the PAL Manual.

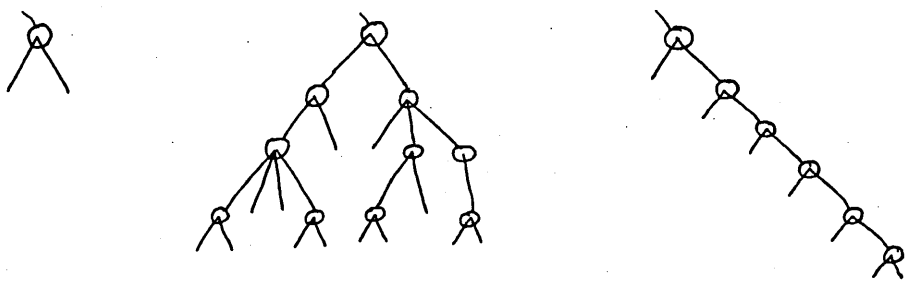
Tuples

As we have already observed in conjunction with the symbolic differentiation program of Chapter 1, the class of tuples is useful in dealing with entities having "structure". Specifically, tuples are important in two ways:

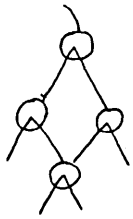
1. A tuple may be used to represent a data structure.
2. A tuple may be used as the argument to "polyadic" function: one that takes several arguments. (This point is explained in Section 2.2 of this chapter.)

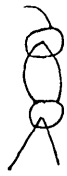
A major part of Chapters 3, 4 and 5 is the formal specification of PAL's semantics. As part of this specification, the PAL program whose semantics are to be explained is represented as a structure. Thus we make extensive use of structured data in these notes, and it is important that the reader understand the ideas.

For the present, we restrict our consideration of structured data to tree-like structures, like this:

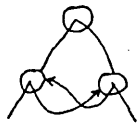


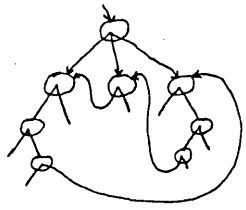
In Chapter 4, we are also interested in objects like





or, even worse, like this





We call the situation in which two arrows point to the same node sharing, and much of our concern in Chapter 4 is the proper explanation of sharing. But sharing becomes relevant only in a world which includes assignment statements -- and there is nothing in the λ-calculus corresponding to assignment. For the present, therefore, we content ourselves with a simplified treatment of tuples in which the concept of sharing does not arise.

PAL's tuples are similar in some ways to vectors in conventional mathematics. Now an n-component vector, say

$$V \equiv V1, V2, \ldots, Vn \qquad (2.1\text{-}10a)$$

may be viewed as a function on the integers such that

$$V(j) = \begin{cases} Vj, & \text{if } 1 \le j \le n \\ \text{undefined}, & \text{otherwise} \end{cases} \qquad (2.1\text{-}10b)$$

Since we are disposed to view computation in functional terms, we ascribe a corresponding property to tuples.

Postulates: We capture the intuitions of the preceding discussion by the following

> Definition: A tuple system [$\mathcal{T}$, Order, Augment] over a terminal set $\mathcal{A}$ is a set $\mathcal{T}$ of tuples together with two functions Order and Augment, such that
>
> 1. No element of the terminal set $\mathcal{A}$ is a tuple. That is, $\mathcal{A} \cap \mathcal{T} = \phi$.
> 2. There is a total function Order: $\mathcal{T} \to$ integers. A tuple t such that Order(t) = k is called a k-tuple, or a tuple of order k.
> 3. Each k-tuple acts as a function over the first k positive integers. That is,
>
> $$t \in \{1, 2, \ldots, \text{Order}(t)\} \to (\mathcal{A} \cup \mathcal{T})$$
>
> 4. There is a unique $\Psi \in \mathcal{T}$ such that Order(Ψ) = Ω. Let $\mathcal{T}_o$ be the set of all tuples other than Ψ.
> 5. There is a function Augment: $\mathcal{T} \otimes (\mathcal{A} \cup \mathcal{T}) \to \mathcal{T}_o$. If t is any element of $\mathcal{T}$ and $x \in \mathcal{A} \cup \mathcal{T}$, then
>   (a) Order Augment(t, x) = 1 + Order(t)

(b) $[\underline{\text{Augment}}(t, x)]\ k = \begin{cases} x & \text{if } k = 1 + \underline{\text{Order}}(t) \\ t(k) & \text{otherwise} \end{cases}$

6. The set $\mathcal{T}$ is defined as follows: Consider a sequence of sets I_n defined

$$T_0 = \mathcal{A}$$
$$T_{n+1} = T_n \cup \underline{\text{Augment}}(T_n \otimes (T_n \cup \mathcal{A})$$

Then $\underline{t} \in \mathcal{T}$ if there exists an $\underline{n}$ such that $\underline{t} \in I_n$.

As in the definition of strings, in which the set $\mathcal{A}_L$ depends on the alphabet $\underline{L}$, the set $\mathcal{T}$ of tuples depends on the terminal set $\mathcal{A}$ chosen. Property 1 requires only that $\mathcal{A}$ not contain any tuples.

We use the term k-tuple for a tuple $\underline{t}$ such that $\underline{\text{Order}}(t) = \underline{k}$. Clearly then Ψ is a 0-tuple, and property 4 implies that it is the only one. Property 3 implies that a k-tuple acts as a function on the first $\underline{k}$ integers. It follows then that Ψ is akin to ϕ, the null function, since each has empty domain of definition. But they are different: $\underline{\text{Order}}(\Psi) = 0$, while $\underline{\text{Order}}(\phi)$ is undefined.

For convenience of expression, we think of a tuple as being "made up" of the elements in its range. Suppose that $\underline{t}$ is a tuple and that $\underline{t}(\underline{k})$ is $\underline{z}$. Then we think of $\underline{z}$ as being the k-th <u>component</u> of $\underline{t}$, or the $\underline{k}$-th <u>element</u> of $\underline{t}$. The impact of property 5 is then that applying <u>Augment</u> to a $\underline{k}$-tuple $\underline{t}$ and any ob $\underline{x}$ yields a $\underline{k+1}$-tuple whose first $\underline{k}$ components are the same as those of $\underline{t}$ and whose $\underline{k+1}$-st component is $\underline{x}$.

It is not hard to show that <u>Augment</u> is one-to-one (by a simple proof using induction) and onto (by property 6) the set $\mathcal{T}_0$. Thus for every tuple $\underline{t}$ other that Ψ, there exist unique $\underline{w}$ and $\underline{x}$ such that $\underline{\text{Augment}}(\underline{w}, \underline{x})$ is $\underline{t}$.

Note that there is a distinction between a 1-tuple and its component. If $\underline{t} = \underline{\text{Augment}}(\Psi, \underline{x})$, it follows (from properties 5b and 4) that $\underline{\text{Order}}(t) = \underline{1}$, so that $\underline{t}$ is a 1-tuple. Further, $\underline{t}(1) = \underline{x}$. But $\underline{t}$ and $\underline{x}$ are different obs entirely, with different properties.

That a component of a tuple may itself be a tuple follows from property 6.

<u>Relation to PAL</u>: The terminal set $\mathcal{A}$ in PAL over which tuples are defined includes

truthvalues ∪ strings ∪ integers ∪ rationals

The PAL name for Ψ is "nil". PAL provides the infix functor "aug" with the property that the PAL expression

E aug F

denotes the same ob as does

$\underline{\text{Augment}}(\underline{E}, \underline{F})$

where $\underline{E}$ denotes $\underline{E}$ and $\underline{F}$ denotes $\underline{F}$.

As we have already observed in Chapter 1, PAL provides a convenient special syntax for denoting tuples of order ≥ 2. For example, if the Ek are PAL expressions then

E1, E2, E3 (2.1-13a)

is a PAL expression that denotes the 3-tuple

$\underline{\text{Augment}}\ [\underline{\text{Augment}}\ [\underline{\text{Augment}}\ (\Psi, \underline{E1}), \underline{E2}], \underline{E3}]$ (2.1-13b)

where $\underline{Ek}$ is the ob denoted by the expression Ek, k = 1, 2, 3. (See Section 3.2 of the PAL Manual for syntactic details.) There is no convenient notation provided in PAL for a 1-tuple. However,

nil aug x

denotes that 1-tuple whose component is $\underline{x}$.

Functions

The class of obs called functions in a universe of discourse $\mathcal{U}$ may be very large or very small. Obviously, the class includes as a minimum all the basic functions, by which we mean all functions introduced by postulate. Thus in a universe of discourse that includes truthvalues and strings we might expect Nor, Adjoin, Stem and Stern to be members of the class "function". Alternatively, of course, the functions Not, And, and Or might be chosen as basic in lieu of Nor, and Conc in lieu of Adjoin. The complete set of basic functions in PAL is treated in detail in Appendix 4 of the PAL Manual, and the functions corresponding to the arithmetic, relational and logical functors ("*", ">", "&", "not", and so forth) are also discussed in Section 2.2 of the Manual.

Presumably, it is possible that the basic functions should be the only obs of type function in $\mathcal{U}$. But such a universe of discourse would be severely impoverished. Typically we are interested not just in the basic functions themselves, but also in more complicated functions specified in terms of the basic ones by such mathematical techniques as functional composition. The composition of a function f onto a function g is a new function, say h, such that

$$h(x) = f[g(x)] \qquad (2.1\text{-}14)$$

In the mathematics literature composition is often denoted "f∘g". Note that

$$f \circ g \neq g \circ f$$

in general. Conceptually, we can draw an analogy to the case of strings: Given a set of "basic" strings (an alphabet) and appropriate mathematical facilities for operating on them (Adjoin, Stem, and Stern), we generate the set of all strings. Similarly, given a set of basic functions and appropriate facilities for operating on them, we wish to generate the set of all functions.

Immediate questions that arise are, "What are the appropriate mathematical facilities for operating on functions?" and "What do we mean by the set of all functions?" The last three sections of this chapter are devoted to answering these questions. We may observe immediately from definitions such as that of "Conc" on page 2.1-24 that functional composition is not the only facility we need.

We defer further investigation of the class function until Section 2.3, saying now only that PAL's universe of discourse includes all of the basic functions, along with those that can be expressed in terms of them.

Semantics

As has been mentioned, the definition of a programming language involves two components:

(1) specification of the legal sentences of the language, and

(2) specification of the meaning of each legal sentence.

The first component concerns the syntax of the language; the second concerns its semantics.

Up to a few years ago, much more research has been devoted to the study of how to define syntax than to the study of how to define semantics. The approach we follow in defining semantics is to establish rules for reducing every PAL program to the application of basic functions; i.e., to the postulates that define the universe of discourse. With reference to the definition of "algorithm" on page 2.1-14, these postulates are "the basic transformations that need no further specification."

The word "need" in the foregoing sentence is clearly subjective; what one "needs" depends upon his objectives. For example, some mathematicians "need" to specify the usual arithmetic operations -- and even the definition of addition on real numbers is a thoroughly non-trivial undertaking. Similarly, a logician may "need" to reduce all transformations to the simplest possible base -- often, to the integers and then to 0 and the function Succ. Our own objective, however, is to study the building up of large semantic constructs out of small ones. For these purposes, numbers, strings, tuples, truthvalues, and the basic functions that operate on these abstract objects constitute an appropriate base for further semantic exploration.

One comment is in order. In our usage the word "semantics" has a connotation much stricter than it has in common parlance. For the broader, more usual connotation, we reserve the words "user interpretation". (Some use instead the word "pragmatics".) Insight into the distinction we wish to make may be gained by reconsidering the system of Figure 1.2-1 on page 1.2-5. We have already observed that this figure is but one of many isomorphic representations of a single underlying abstract system. Let us assume an implementation of this abstract system in PAL such that the three objects are denoted by "I", "II", "III" and the transformation by "Mod". We now observe that the function f defined by the PAL program

```
def f(x) = Mod(II, x)
```

may be interpreted in various different lights. One programmer might think of it as the modulo 3 successor function, whereas another might think of it as effecting a 120° rotation of a three-cog wheel. It is this choice of point of view that we call user interpretation. Thus the semantics of a transformation has to do with the abstract transformation that takes place, while the user interpretation has to do with how the user thinks of it.

As another example, consider the expression

```
A + P
```

where A and P denote integers. The semantics of this expression has to do with the properties of the integers and the adding function; but the user interpretation might be that A counts apples and P pears, and that the expression itself counts pieces of fruit.

2.2 Functional Application

The universe of discourse of a programming language is the set of obs that the language deals with. In section 2.1 we have defined Ω, the universe of discourse of PAL, the definition being (for the most part) by the mathematical technique of axiomatization.

We have defined Ω in order to answer question (A) on page 2.1-14, and we answered (B) at the same time. It is now time to start to answer (C): What facilities does the language afford for building complicated transformations out of simple ones? This is the question which we skipped earlier in the subsection on functions. As we answer it, we are able to write expressions which denote values which we have not axiomatized, such values always being functions. For example, the PAL definition

```
def S(x) = Stem (Stern x)
```

defines a function S whose domain is strings of length two or more and whose range is strings of length one, S returning the second character of its argument. Clearly S has not been postulated.

Heretofore we have used the word "basic" to refer to the obs we have defined by postulates. We now expand its use, with the following definitions:

> Definition: A basic ob is any member of the universe of discourse Ω.
>
> Definition: A primitive ob is a basic ob that has a name.
>
> Definition: A primitive identifier is the name of a primitive ob.

In the next subsection we discuss the details of the association of primitive identifiers with primitive obs, and in the following subsection we explore facilities for denoting the application of functions to arguments.

Primitive Environment

Our axiomatization of the universe of discourse provides us with the primitive identifiers of PAL: those identifiers that are defined ab initio for users of the language and which denote obs in the universe of discourse. We wish now to clarify the mechanism for this association of identifiers with obs. Our principle reason for doing this is that the mechanism is expanded in Section 2.3 to include also provision for variable identifiers: those defined by the programmer.

The words "identifier", "variable", "constant" and others appear frequently in what follows and are always used in a consistent manner. The basic elements

of the PAL language are classified as follows:

- names
  - identifiers
    - variables
      - primitive variables
      - programmer-defined variables
    - constants
      - quotations
      - numerics
        - integer numerics
        - rational numerics
      - literals
  - functors
- punctuation

Table 2.2-1: Basic Elements of the PAL Language

Names are used to denote obs in the universe of discourse and are either identifiers or functors. The association of a functor, such as "+" or "<", with an ob is implicit, whereas the association of an identifier with an ob is explicit. Punctuations are marks such as parentheses which have a syntactic purpose but which do not denote obs. Literals include "true" and "false" and "nil", and the other sets have already been discussed.

We have defined a "primitive identifier" to be the name of a primitive ob. In terms of the above classification, a primitive identifier is either a constant or a primitive variable. The only identifiers which are not primitive identifiers are the programmer-defined identifiers.

We turn our attention now to the association of primitive identifiers with primitive obs. Figure 2.2-1 is a sketch of what we call a primitive environment for some language. Here we assume the existence in the language of a set of primitive identifiers. Each such identifier is a name associated with an ob in the set of basics, and we call the mapping from identifiers into basics a "primitive environment". We consider the choice of an appropriate primitive environment to be the starting point in the definition of a programming language.

A primitive environment is a meta-function whose domain of definition is a set of primitive identifiers and whose range is a set of primitive obs. The functionality of a primitive environment -- abbreviated "PE" -- is therefore

$$PE \in \text{primitive identifiers} \rightarrow \text{primitive obs} \qquad (2.2\text{-}1)$$

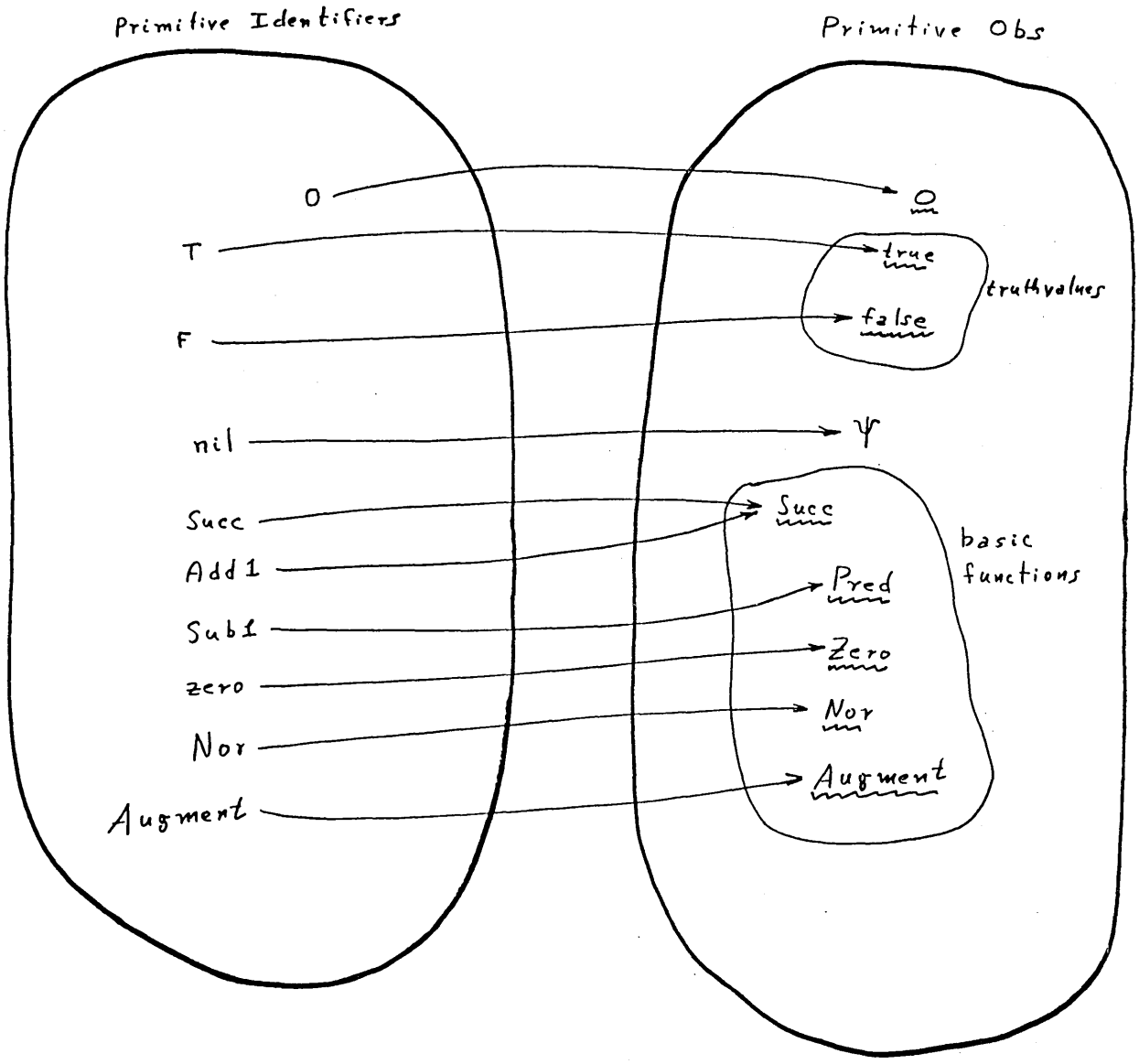


Figure 2.2-1: A possible primitive environment for some language. The primitive identifiers are shown in the circle on the left and the primitive obs in the circle on the right. The syntax of identifiers does not happen to coincide with that of PAL.

Semantic Considerations: The role of the primitive environment in the definition of a language is critical, in that the choice of primitive environment determines the universe of discourse available to the user of the language. Only obs within the closure of the primitive obs under functional application can be denoted in the language, simply because the only way to specify an ob is either directly via a name or indirectly via the application of functions to arguments. Since primitive obs are the only obs with predefined names in the language, only obs ultimately expressible in terms of them can be denoted at all.

By way of example, we see that the universe of discourse of a language whose primitive environment is that of Figure 2.2-1 is the union of the natural numbers, the truthvalues, and tuples whose terminal elements are either natural numbers or truthvalues. On the other hand, if either of the domain-range pairs

(nil, Ψ) (Augment, Augment)

were deleted from this primitive environment, then tuples would be deleted from the corresponding universe of discourse.

Although the range of a primitive environment must be complete, in the sense that its closure under application must include all desired obs, considerable flexibility is still possible. We already know that substituting the domain-range pairs

(Not, Not) (And, And)

for the pair

(Nor, Nor)

in the PE of Figure 2.2-1 leaves the universe of discourse unchanged. The important observation is that in general there are many different sets of primitive obs each of which generates as its closure the same class of obs. From a semantic point of view, all such sets are equivalent.

Syntactic Considerations: Identifiers are linguistic "atoms", by which we mean the building blocks out of which the sentences of a language are constructed. In Chapter 1 we characterized "atoms" as entities whose substructure (if any) is irrelevant, a comment which is valid here as well. From a semantic point of view, the only property we require of identifiers is that, given any two of them, we can tell whether or not they are distinct. (Note that distinctness of identifiers does not necessarily imply distinctness of the obs which they denote, as witnessed by the fact that both "Succ" and "Add1" map onto the ob Succ in Figure 2.2-1.) For example, changing each of the primitive identifiers in Figure 2.2-1 into its Russian equivalent would change the syntax but not the semantics of a language based on that primitive environment: We do not care whether the null tuple is called "nil" or "nyet", so long as the convention is agreed upon. Indeed, there would be no harm in adopting both names, just as

"Add1" and "Succ" are synonyms.

A second syntactic decision concerns whether or not to provide primitive identifiers for every ob of any given type. In Figure 2.2-1 predefined identifiers are provided for each of the two truthvalues, whereas it is necessary to do so for only one of them since

Nor(true, true) = false

and

Nor(false, false) = true

By contrast, 0 is the only ob in the class of natural numbers which is coupled with a primitive identifier. In consequence, a language based on this PE provides no direct means for referencing any number other than zero, which would seem to be a syntactic inconvenience intolerable in any practical language. On the other hand, we know that the inconvenience affects neither the universe of discourse nor our ability to compute the other numbers.

Finally, we observe that a similar situation obtains in conjunction with the class tuples: The null tuple is the only tuple denoted by a primitive, either in Figure 2.2-1 or in PAL. The inconvenience in this case seems unavoidable; we know of no sensible syntactic device for associating every tuple with an appropriate predefined name.

Relation to PAL: The structure of PAL's primitive environment is indicated in Figure 2.2-2. Here the primitive identifiers have been partitioned into the syntactic categories numerics, quotations, literals and predefined variables, in a manner consistent with the classification on page 2.2-34. Although such partitioning is inessential from a theoretical point of view, significant practical advantages accrue from the adoption of a judicious syntax for identifiers. Specifically, in PAL (as in most languages) both the type and the value of the ob associated with a constant identifier is immediately deducible from the syntax, without table look-up. Clearly, one stands to gain efficiency of implementation when table look-ups are minimized -- especially so if the tables are large. For numerics and quotations the tables would be (at least in principle) unboundedly large.

The alternative to providing a "transparent" syntax for numerics and quotations, of course, is to provide primitives for only a small subset of numbers and strings, as in Figure 2.2-1. But as a practical matter we again stand to gain efficiency by increasing the number of primitives beyond the minimum required for semantic completeness: Presumably it is more efficient to recognize that the numeric "3" denotes the third positive integer then it is to compute

Succ (Succ (Succ (0)))

A similar consideration holds with respect to the basic functions.

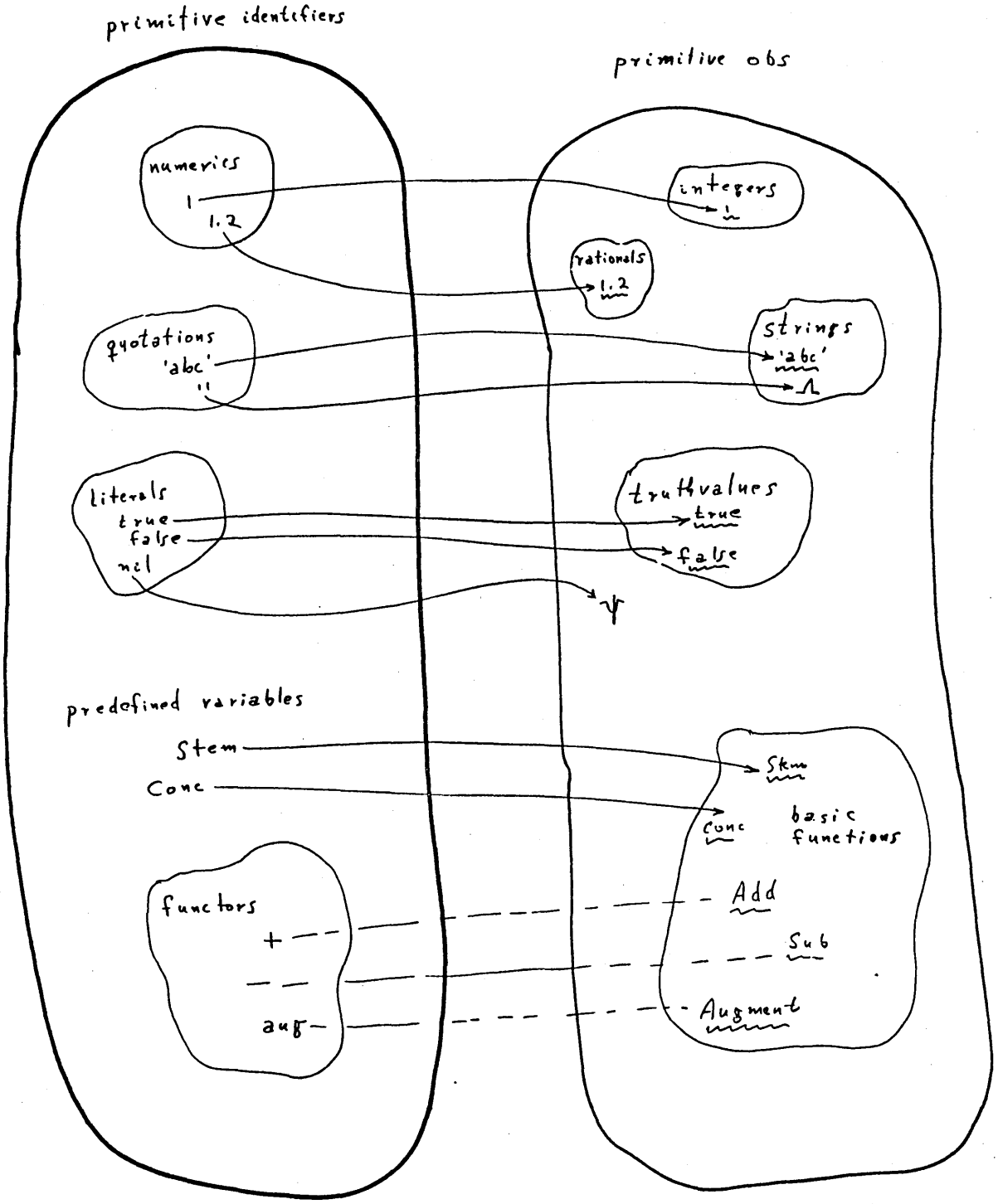


Figure 2.2-2: Skeletal diagram of PAL's primitive environment. Although functors are shown (with a dashed line), it should be understood that they are not identifiers at all.

Although only Succ, Pred and Zero are required to enable arithmetic on the natural numbers, it is far more efficient to provide arithmetic functions directly, as additional basics, than to rely on recursive definitions such as those of (2.1-9). Accordingly, PAL's primitive environment is much richer than it would need to be were our purpose simply to establish the properties of all obs in its universe of discourse. In particular, as we have already remarked, each arithmetic, logical and relational functor corresponds to a basic function.

Note that functors are treated specially in Figure 2.2-2, for example "+" being associated via a dashed line with Add. This suggests that the programmer may write

3 + 5

to denote

Add (3, 5)

We see in the next subsection that we regard the infix functor as an alternate syntactic device for functional application.

Applicative Structure

In the preceding subsection we have seen that a primitive environment provides linguistic facilities for denoting directly some obs in a universe of discourse: the primitive obs. A remaining task is to devise linguistic facilities for writing expressions which denote obs which are not primitive. As a preliminary, in this subsection we analyze the structural aspects of functional application.

In conventional mathematics, many different notations are used to indicate the application of a function to arguments. For example, the operator may be prefix, infix or postfix, as in the expressions

- 4.89
17 + 9
5! (i.e. 5 factorial)

respectively. Alternatively, the physical layout in two dimensions may be significant, as in the expressions

$$2.7^2$$

and

$$\int_a^b x^2 \, dx$$

In every case, however, only one semantic issue is important: Whatever the

syntactic form, it must permit us to determine the operator and the operand. In other words, we must be able to elicit answers to the questions "What ob is the function to be applied?" and "What ob is the argument?" Of course, both operator and operand may themselves be the result of functional application. By an applicative structure we mean a display in which the operator and operand of each subexpression is explicit.

The applicative structure of ordinary arithmetic expressions may be exhibited in any of several ways. For example, we can write

$$(2 - 6) * (-5) \qquad (2.2\text{-}2)$$

which is a linear representation of the same AE that can be represented in tree form as

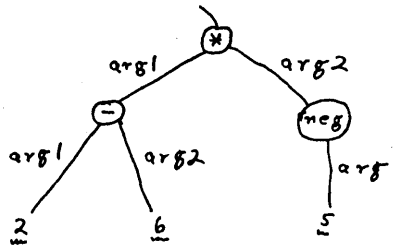


Figure 2.2-3: Tree Form of the AE of (2.2-2)

Each node of the tree represents an operator, and the branches diverging from each node represent the corresponding operands. In general, unfortunately, a display such as that of Figure 2.2-3 does not prove adequate, primarily because of asymmetry in its treatment of operators and operands. The asymmetry is not bothersome in ordinary arithmetic expressions where only operands are themselves the result of functional application. But we must also accommodate situations in which the operator results from an application. For example,

$$[\underline{\text{Stem}} \circ \underline{\text{Stern}}]\ (\underline{\text{'AB'}}) \qquad (2.2\text{-}3)$$

denotes application of the functional composition of Stem and Stern to the string 'AB'. The operator "$\circ$" for functional composition is defined in (2.1-14). Obviously "composition" may be regarded as a function, say Comp, and the operator of (2.2-3) is the result of applying Comp to the functions Stem and Stern.

In simple cases like (2.2-3), presumably one could still represent the operator by a node, like this:

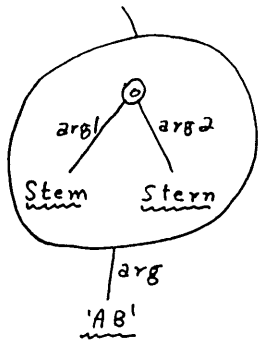


Figure 2.2-4a: Tree Form of the AE of (2.2-3)

The situation can easily get out of hand, however, since the structure of the operator may be arbitrarily complex. Accordingly, we elect always to treat operators on a par with operands, and to display them each along branches of the tree. One possibility is this:

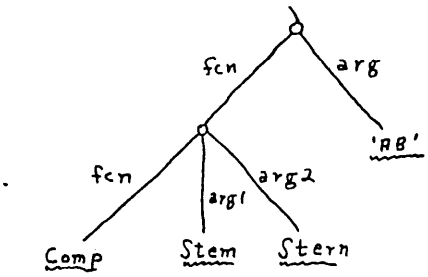


Figure 2.2-4b: Alternate Tree for the AE of (2.2-3)

A PAL function "Comp" corresponding to Comp may be defined by the PAL program

```
def Comp (f, g) =
    P where P(x) = f [g(x)]
```

Then the functionality of Comp is

$$\text{Comp} \in (\beta \to \gamma) \otimes (\alpha \to \beta) \to (\alpha \to \gamma)$$

where α, β and γ stand for (unspecified) types of ob. In other words, the domain of Comp is an ordered pair of functions: if the first maps obs of type β onto obs of type γ, and the second maps obs of type α onto obs of type β, then the result of Comp is a function that maps obs of type α onto obs of type γ.

Adicity: Although not apparent from the tree just drawn, complete parity between operator and operand still remains to be achieved. The residual disparity involves the fact that while there is only one operator in any functional application there may be several arguments, depending upon the "adicity" of the function. By its adicity we mean whether a function is monadic, diadic, triadic, or whatever.

We achieve parity by adopting the convention that every function takes a single operand. The k-adic case (for $k \neq 1$) is then accommodated by stipulating that the operand be a k-tuple. With this convention, the applicative structure of nested applications may be displayed using only two node types: γ and τ. A gamma node indicates functional application and has two sons, the left being the function and the right the argument. A tau node with n sons indicates an n-tuple, the components being taken from left to right. With these conventions, (2.2-2) and (2.2-3) may be represented like this:

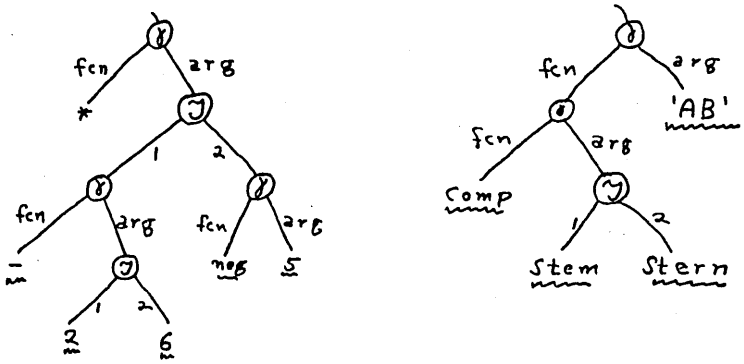


Figure 2.2-5: Trees with γ and τ nodes

In this figure we have labelled the branches from the γ node with "fcn" and "arg" and the branches from the τ node are numbered.

Heretofore we have used the $\otimes$ notation to specify ordered pairs, making no attempt to associate such pairs with PAL's 2-tuples. (On page 2.1-16 we stated explicitly that we were refraining from making such an association.) We now decree that there is no distinction between the ordered pair $\{a, b\}$ and the 2-tuple whose PAL representation is "(a, b)" where a and b denote a and b, respectively. This decision permits us to use the $\otimes$ in functionality notation in connection with functions defined in PAL, even though we have seen that (in a technical sense) all PAL functions are monadic. For example, we say

$$\underline{\text{Conc}} \in \text{string} \otimes \text{string} \rightarrow \text{string}$$

to indicate that Conc must be applied to a 2-tuple of strings. We tolerate the abuse of terminology of using the phrase "a k-adic function" to refer one that must be applied to a k-tuple.

Although adoption of this "single operand" convention is somewhat arbitrary, two advantages accrue. First, the axiomatic treatment of functions and transformations on functions in the remainder of this chapter is simplified. And second, we avoid introducing a semantic distinction which inhibits (to some extent) the expressiveness of a language. To illustrate this last point, consider the implication of rejecting the convention. Then to denote the application of a k-adic function we must specify an "argument list with k elements". But how does such an argument list differ from the k-tuple

with corresponding elements? If there is no difference, then all operands actually are single entities, regardless of protestations to the contrary. Thus we must exhibit some property which tuples and argument lists do not have in common. For instance, we might consider argument lists to be purely syntactic entitities, and ascribe no semantic (i.e., transformational) properties to them at all.

This possible distinction between tuples and argument lists is, in fact, typical of the situation in most programming languages. In this approach, each element of an argument list denotes an ob in the universe of discourse, but the argument list itself does not. Specifically, no functions (comparable, say, to Augment) are postulated for computing an argument list, or for transforming one argument list into another. To see that this syntactic approach does to some extent inhibit freedom of expression, consider the following example. Assume that the syntax for writing an argument list were

$$\langle E1, E2, \ldots, Ek \rangle \qquad (2.2\text{-}5a)$$

in which each Ei stands for an expression denoting the i-th argument. Then we might denote the concatenation of two strings by writing

$$\text{Conc } \langle S1, S2 \rangle \qquad (2.2\text{-}5b)$$

in which any arbitrary expressions denoting strings may be substituted for S1 and S2. But if there is no way to compute an argument list, we would be precluded from writing

$$\text{Conc } (S) \qquad (2.2\text{-}5c)$$

in which any arbitrary expression denoting an ordered pair of strings may be substituted for S. By contrast, adoption of tuples in lieu of argument lists does admit (2.2-5c) as a valid and frequently convenient alternative to (2.2-5b).

In computation as in other fields of engineering, we may hope to gain in one direction if we compromise in another. For example, we can offset the loss in expressiveness by exploiting the fact that the adicity of the operand is manifest from the syntax in (2.2-5b), but not in (2.2-5c), to gain increased efficiency of implementation. In defining PAL, however, adoption of the single-operand convention seems particularly appropriate and is assumed hereafter.

Curried Functions: We have built the universe of discourse Ω so that it includes functions, and we understand that the value of any application may be any ob in Ω. Thus the possibility of the value of an application being itself a function is apparent. A function-producing function is one which, when applied to a suitable argument, has a function as value. We have already encountered one example of a function whose range is functions in the functional composition operator Comp, and it is easy to define others. For instance, suppose that Sum

is a function-producing function with domain integers such that if

$$g = \underline{Sum}\ (m)$$

where $\underline{m}$ is an integer, then for any integer $\underline{n}$ we have

$$g(n) = m+n$$

That is,

$$[\underline{Sum}(m)]\ (n) = m+n \qquad (2.2\text{-}6a)$$

It should be clear that

$$\underline{Sum} \in integer \rightarrow (integer \rightarrow integer) \qquad (2.2\text{-}6b)$$

It is not at all convenient to write a definition of Sum in conventional mathematical notation, although

$$Sum(m) \equiv g \text{ where } g(n) = m+n \qquad (2.2\text{-}6c)$$

comes close. Functions such as Sum are called "curried" functions, after the logician H. B. Curry, and (as we see later) provide a clean logical base for much of our view of computation. Such functions were introduced by Schönfinkel (1924) and have been used extensively by Curry.

For every binary function there is a curried function which is, in a sense, equivalent to it. For example, consider the function Sum defined in (2.2-6) and the function Add defined by

$$Add(x, y) = x + y$$

Clearly,

$$\underline{Add}(x, y) = [\underline{Sum}(x)]\ (y)$$

for all numbers $\underline{x}$ and $\underline{y}$. We say then that Sum then is a curried version of Add.

There is considerable conceptual efficiency to be gained by using curried versions of the basic functions in Λ. In particular, consider the function Aug that is a curried version of Augment, so that

$$[\underline{Aug}(x)]\ (y) = \underline{Augment}(x, y)$$

Thus the 2-tuple

$$(s, t)$$

which we have seen can be written as

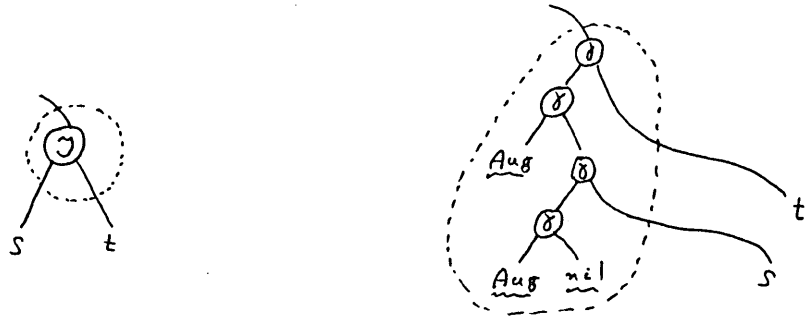
$$\underline{Augment}\ [\ \underline{Augment}(\underline{nil}, s), t\ ]$$

can alternatively be written as

$$[\underline{Aug}\ ((\underline{Aug}\ \underline{nil})\ (s))]\ (t)$$

That there is conceptual advantage to use of curried functions can now be shown.

Recall that in Figure 2.2-5 we saw that any expression can be represented by a binary tree whose non-terminal nodes are γ or τ. We see now that using curried basics lets us dispense with τ nodes. Thus the 2-tuple (s, t) may be represented by either of



In fact, the two trees enclosed in dashed lines are equivalent, an obvious correspondance extending to τ nodes with any number of sons.

Trees without τ nodes are conceptually more fundamental than those with them, in the sense that they need only one node type rather than two. On the other hand, the style with τ nodes is more abbreviated and hence more perspicuous. Since it is clear that there is a one-to-one correspondence between the two display alternatives, we feel free in what follows to use whichever is more appropriate to the purpose of the moment. Figure 2.2-6 shows the tree form of Figure 2.2-5, with τ nodes replaced by γ nodes and Aug. Note the rather dramatic increase in the size of the trees.

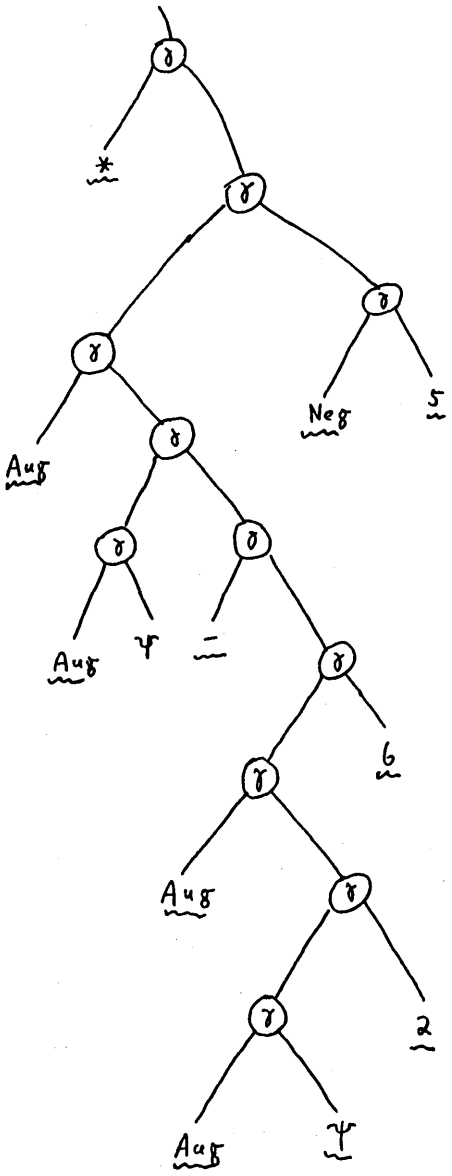
Combinations

A tree displaying the nesting of functional applications is one way to represent an arbitrary ob in terms of the basic obs of a universe of discourse. Although it is clear that such a tree has the virtue of explicitness, the virtue of conciseness is notably absent. We now address ourselves to the linguistic issues involved in devising alternate representations better suited to human needs. All such representations -- that is, all expressions in a language which denote the application of an operator to an operand -- are called combinations.

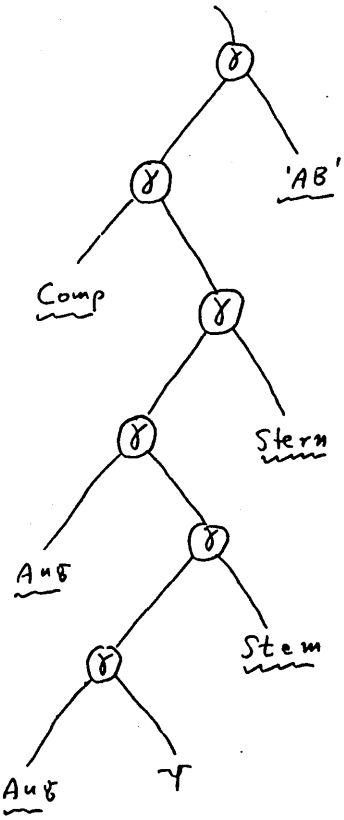
PAL Syntax: The syntax of combinations adopted in PAL corresponds closely to our meta-language, which is conventional mathematical notation. The principal distinction between the two is the replacement of meta-names by PAL primitive identifiers. For example, the PAL combination

```
Conc ('A', '5.7')                                    (2.2-7)
```

denotes application of the function Conc to the strings 'A' and '5.7'. Similarly,



(2-6) * (-5)



[Comp(Stem, Stern)] 'AB'

Figure 2.2-6: Trees without $\mathfrak{J}$ nodes.

These trees correspond to those in Figure 2.2-5, but in these $\mathfrak{J}$ nodes have been replaced by $\mathfrak{g}$ nodes and Aug.

5 + 7 (2.2-8a)

In PAL denotes application of "the function that adds integers" to the integers 5 and 7.

This interpretation of (2.2-8a) reflects our view that infix notation is syntactic "sugar" for a corresponding prefix expression. Thus we consider (2.2-8a) to be merely a more palatable way of writing

⌈Sum (5)⌉ (7) (2.2-8b)

in which (presumably) the identifier "Sum" and the infix functor "+" denote the same curried function.

PAL syntax also differs from ordinary mathematical notation by being specific about conventions governing the use and omission of parentheses in combinations. For example, in PAL the infix combination

a/b*c (2.2-9a)

is construed as

(a/b)*c (2.2-9b)

rather than as

a/(b*c) (2.2-9c)

Similar conventions govern the parenthesization of prefix combinations. For example

f(x) (2.2-10a)

may also be written in PAL as

f x (2.2-10b)

or even as

(f) x (2.2-10c)

Each of these denotes the result of applying the function denoted by "f" to the argument denoted by "x". Finally, PAL uses the convention that combinations associate to the left, so that

A B C (2.2-11a)

is construed as

(A B) C (2.2-11b)

rather than as

A (B C) (2.2-11c)

Thus (2.2-8b) could be written equivalently as

Sum 5 7

The decision to use left association for functional application is particularly convenient given our predilection to the use of curried functions. Were functional composition more common in this work, the decision might have been otherwise.

The details of PAL's syntax are described fully in the Manual. Hereafter, we seek to avoid possible ambiguity and achieve consistency by always using PAL notation when writing combinations. In general we over-parenthesize to remove possible doubt, but as we do make extensive use of the left association convention of (2.2-11) and the omission of parentheses convention of (2.2-10), the reader had best accustom himself to them.

Referential Transparency: Since our interests embrace not only PAL but also the linguistic principles on which PAL is based, it is appropriate to investigate certain aspects of notation from a fundamental point of view. In this subsection we discuss an important linguistic attribute, called referential transparency (cf. Quine (1960), pages 141 to 145), which contributes mightily to the perspicuity of combinations both in conventional mathematics and in PAL.

Roughly speaking, an expression is referentially transparent with respect to a subexpression if and only if the value of the expression depends on that subexpression solely through the value of the subexpression. For example, the expression

$$7 * (1 + 4) \qquad (2.2\text{-}12)$$

is referentially transparent with respect to the subexpression "1 + 4" because any expression whose value is 5 can be substituted for it without changing the meaning of the expression as a whole. By contrast, the definite Reimann integral

$$\int_a^b x^2\ dx \qquad (2.2\text{-}13a)$$

is referentially transparent with respect to "a" and "b", but not with respect to the second occurrence of "x". Obviously, no other expression whatsoever can be substituted for just the second occurrence of "x" without destroying the overall meaning, although equally as obviously the meaning is invariant to certain substitutions for both occurrences, as in

$$\int_a^b y^2\ dy \qquad (2.2\text{-}13b)$$

Note that the definition of referential transparency has to do with subexpressions. Thus "2 * 3 + 4" is not the same as "2 * 7", even though "7" has the same value as does "3 + 4", since "3 + 4" is not a subexpression of the

whole. Even more dramatically, "32 + 21" is not the same as "341".

It is clear from (2.2-13) that referential transparency is not a necessary attribute of viable notation, nor even in all cases a desirable attribute. On the other hand, two advantages accrue in situations in which there is referentially transparency. First, when an applicative structure is deeply nested, the intellectual task of discerning overall meaning is greatly simplified if we can think of the process of evaluation of a tree (such as that of Figure 2.2-6) by replacing each sub-tree in turn by its value. (An example is given below.) And second, our freedom of expression is enhanced by license to substitute for any subexpression any other subexpression having the same meaning. The two advantages are obverses of each other, so that actually only one idea is involved.

An example of the evaluation process aluded to above is shown in Figure 2.2-7, which shows successive steps in the evaluation of (2.2-2). At each step a node is selected both of whose sons are terminal nodes, and the subtree consisting of that node and its sons is replaced by the proper value. Referential transparency not only legitimatizes this process, but also it guarantees that the final value produced does not depend on the order of evaluation.

Semantic Trees: PAL's notation for combinations is influenced strongly by a desire to emulate conventional mathematical notation. Nonetheless, from a syntactic point of view PAL falls far short of conventional mathematics in variety of allowable notation. Today's programming language designer unfortunately is effectively bounded away from bold-face, italic or Greek characters, and usually from superscripted and subscripted symbols, by limitations of the input/output devices commonly available on computers.

On the other hand, from a semantic point of view PAL and conventional notations for combinations are equivalent, in the sense that both satisfy a constraint of referential transparency. Specifically, this constraint requires that any expression denoting a functional application must be analyzable into two subexpressions (say "rator" and "rand"), and must be referentially transparent with respect to them both. Thus we are led to extend our notions of "expression" and "value" by defining a set of objects called applicative expressions (AE's), as follows:

> An AE is a structured object. It is either a
> primitive identifier, or it is a
> combination, in which case it has a
> rator, which is an AE, and a
> rand, which is an AE.

(This is our first use of a structure definition, and the reader is advised to take this opportunity to reread the description of such definitions starting on page 2.1-19.) We can exploit the predicates and selectors implied by this

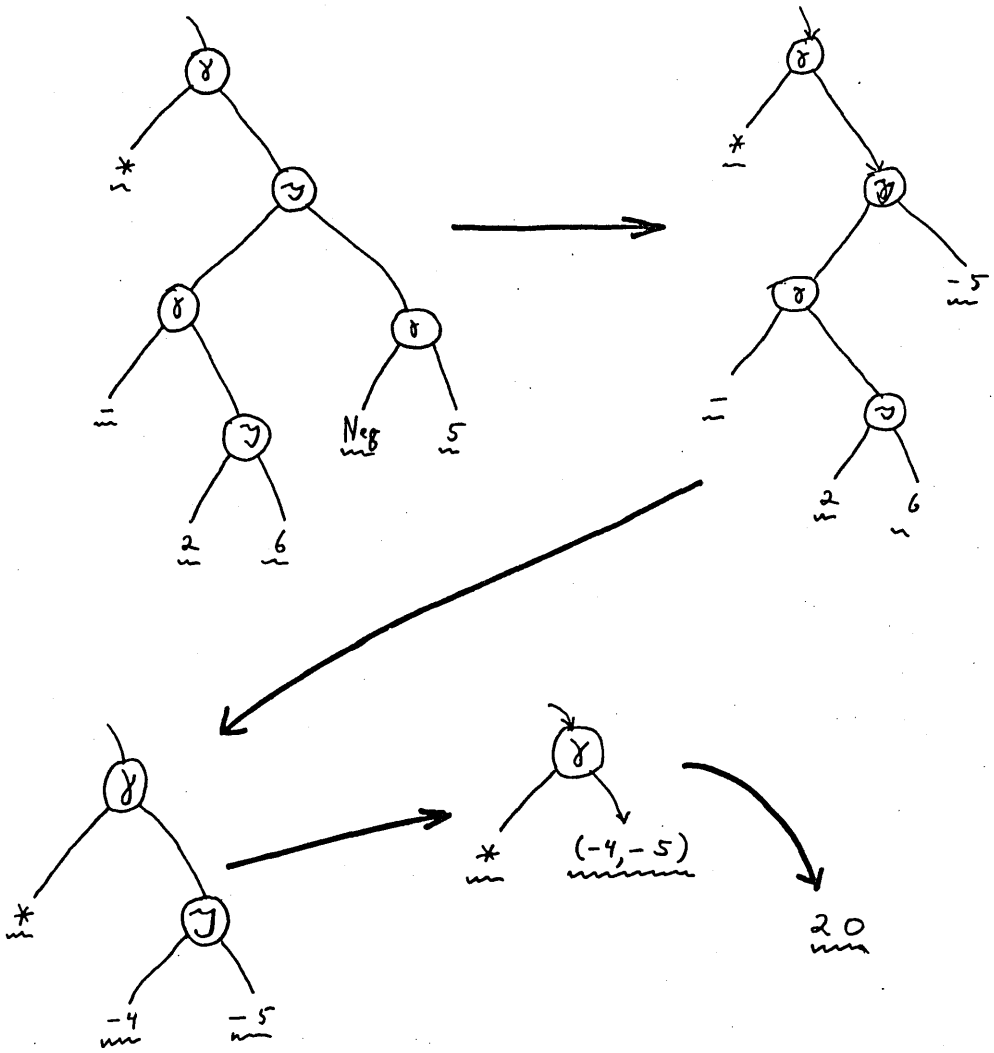


Figure 2.2-7: Evaluation of the AE

(2 - 6) * (-5)

by successive tree contractions. In each step a subtree is replaced by (a representation of) the value which it denotes. Some steps have been omitted.

definition to define the value of an AE:

> Definition: The value of an AE is determined as follows: If the AE is a primitive identifier, its value is the ob associated with it in the primitive environment. The value of a combination is that ob that results when the value of the rator is applied to the value of the rand.

This definition, which depends on the primitive environment used, specifies the value of any combination each of whose atomic constituents is a primitive. Specifically, one procedure for determining the value of any such expression would begin by analyzing and displaying the rator-rand structure of the expression as a semantic tree, as in Figure 2.2-8. Such a tree differs from the trees of Figures 2.2-5 and 2.2-6 primarily in its interpretation: The present tree denotes linguistic ideas, whereas the earlier trees designate abstract obs. Stated differently, the leaves (i.e., terminal nodes) in Figure 2.2-8 are primitive identifiers, while the leaves in the earlier figures are primitive obs. The non-terminal nodes are labelled AP (for application) and COMMA rather than $\aleph$ and $\beth$, to emphasize the difference.

Given such a semantic tree, however, we can still substitute for each identifier the value it denotes, and then proceed to replace each sub-tree by its value, just as in Figure 2.2-7. In accordance with the definitions, the end result of contracting the entire tree is the value denoted by the original expression.

The distinction between the two classes of trees typified in Figures 2.2-6 and 2.2-8 is academic, simply because the two classes are isomorphic, or at least almost so. This isomorphism, of course, is the simplification which referential transparency affords. But each additional linguistic construct encountered hereafter enriches the class of structures we call "semantic trees", whereas the structure of functional applications is fixed by the postulates establishing the universe of discourse.

The relationship established thus far between the worlds of expressions and abstract objects may be visualized as illustrated in Figure 2.2-9. In the linguistic world we generate new expressions by "combining" two expressions called rator and rand. In the universe of discourse we generate new obs by "applying" an ob called function to an ob called argument. As shown in the figure, referential transparency implies invariance with respect to the path followed in moving between worlds.

That the two worlds are in fact distinct is emphasized by realization that the mapping from expressions onto obs is incomplete. By this we mean that not every combination corresponds to an ob in Ω. For example,

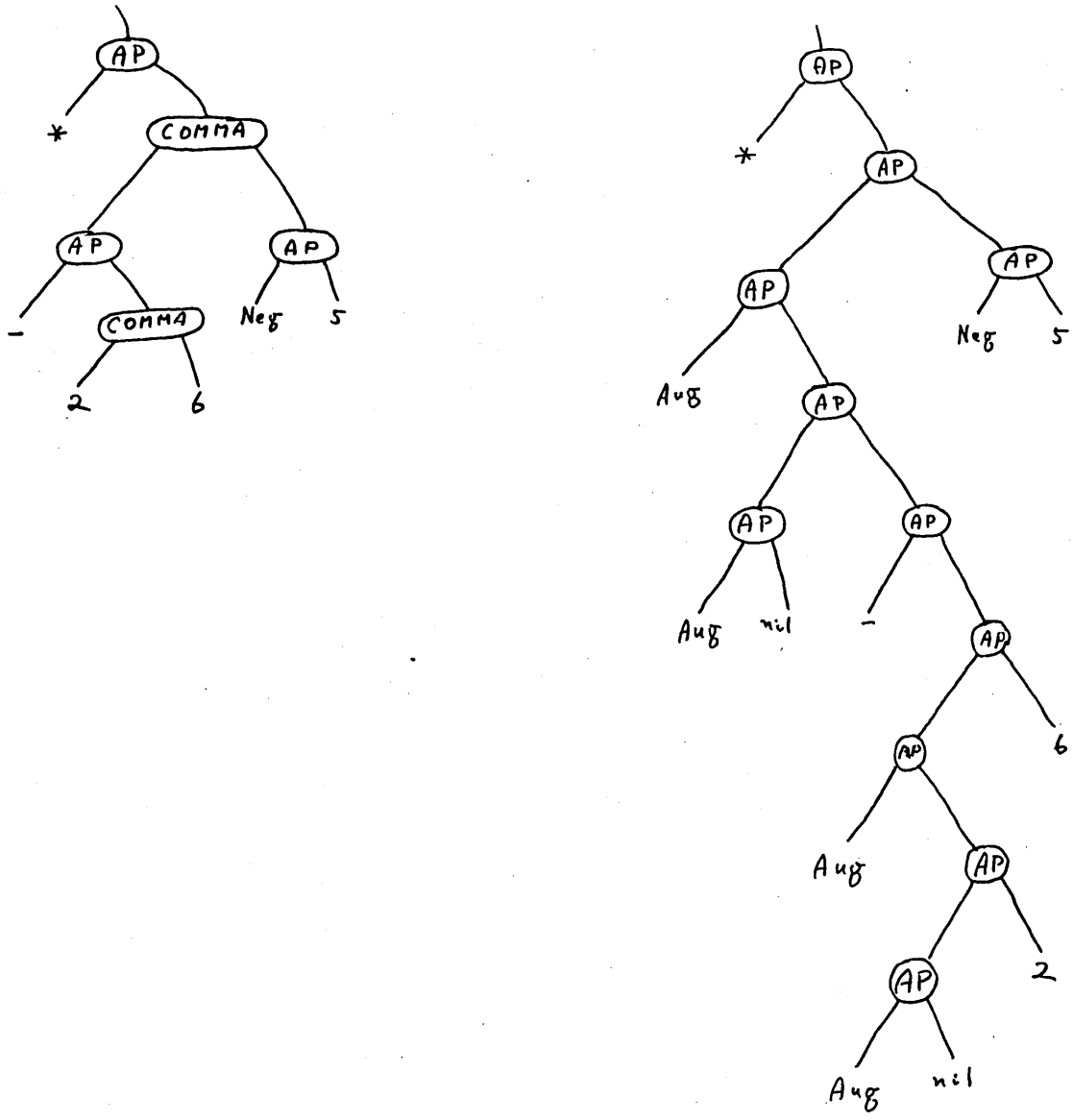


Figure 2.2-8: Examples of semantic trees. Each is a representation of the AE

(2 - 6) * (-5)

The tree on the left uses COMMA nodes and the one on the right does not.

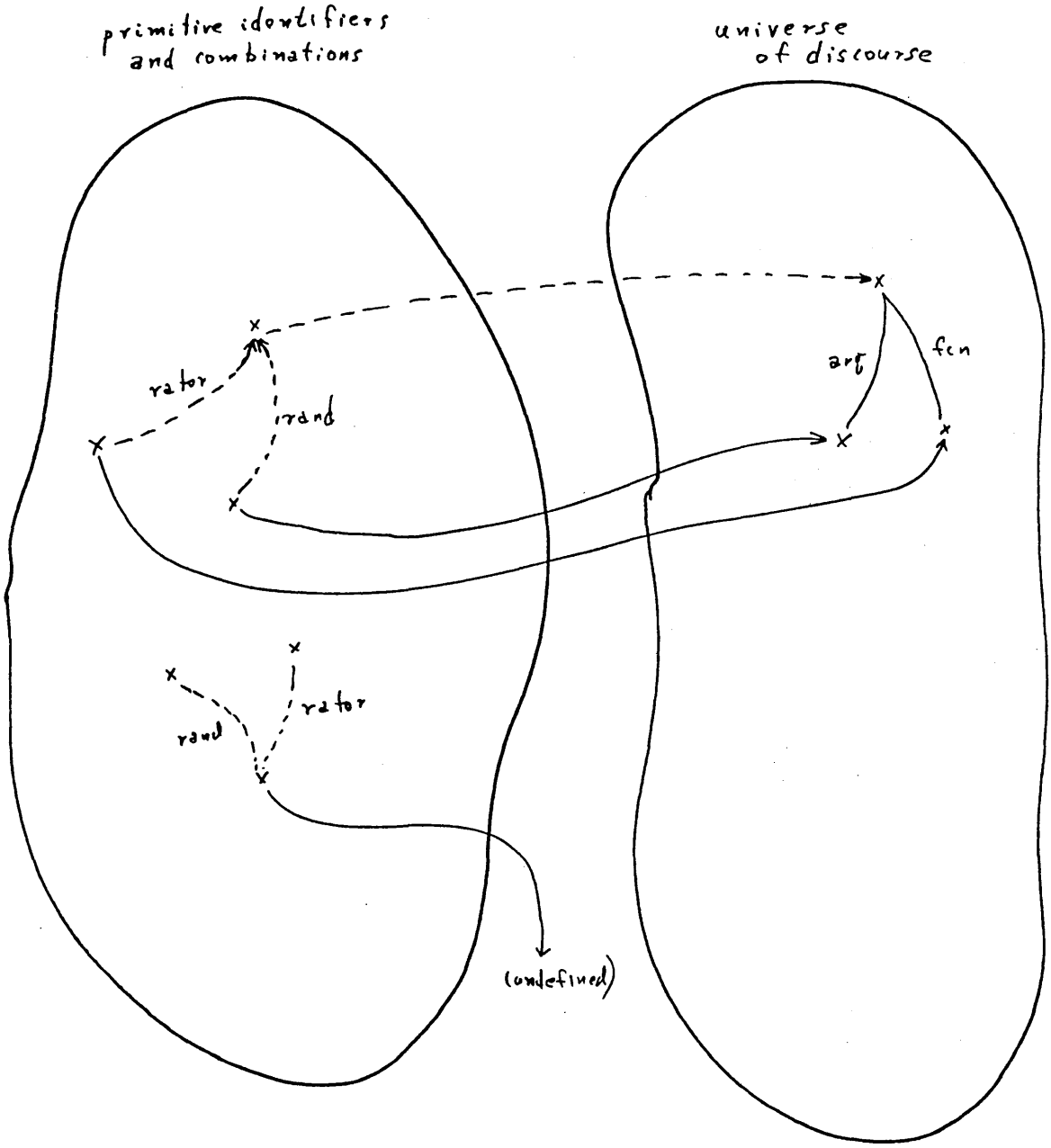


Figure 2.2-9: Relation between combinations and abstract objects. Note the invariance to path in the mapping from expressions onto obs: One reaches the same ob regardless of whether he follows the solid or dashed arrows. This invariance is another aspect of referential transparency. But note also that certain combinations do not map onto any ob.

is an AE which is syntactically correct, but its value is undefined because the ob 5 is not in the domain of the ob Stem. Thus the correspondance is not quite an isomorphism, the problem being that the functions involved are not all total over their domain. In particular, let AE be the set of AE's defined on page 2.2-49, and let AP: AE ⊛ AE → AE be the constructor of combinations implied by that structure definition. That is, if a and b are AE's then

AP (a, b)

is that AE whose rator is a and whose rand is b. Let Ω be the universe of discourse, and let γ: Ω ⊛ Ω → Ω be that function such that

γ (a, b)

is the result of applying a to b.

Now let PE: AE → Ω be the primitive environment which associates with each atomic AE a value in Ω. Finally, consider Val: AE → Ω defined by

$$\text{Val}(x) = \begin{cases} \text{PE}(x) & \text{if } x \text{ is atomic} \\ \gamma[\text{Val}(M), \text{Val}(N)] & \text{if } x \text{ is the combination } (M\ N) \end{cases}$$

It would be very pleasant were Val an isomorphism from (AE, AP) to (Ω, γ) but it is not: Since γ is not total on Ω ⊛ Ω it follows that Val is not total on AE ⊛ AE. Thus for example to evaluate (2.2-14) we have

Val [AP (Stem, 5)]
= γ [Val(Stem), Val(5)]
= γ (Stem, 5)
= Stem 5

which is not defined. Moreover, in general it is not possible to evade this problem by excluding "meaningless" expressions from a language. In particular, in Section 2.3 we extend our set of linguistic constructs to include facilities such that an identifier (say "S") may be defined to denote any ob whatsoever. Whether or not a combination such as

Stem S

is meaningful then depends not just on the expression itself, but on the context in which it occurs. The sad fact is that it is possible to write syntactically correct sentences which are semantic nonsense in any highly developed language.

We summarize much of the preceeding discussion of the evaluation of AE's by the

Definition: Two AE's M and N are said to be **δ-equal**, written M $\overset{\delta}{\leftrightarrow}$ N, if Val(M) = Val(N).

Clearly $\overset{\delta}{\leftrightarrow}$ is an equivalence relation on AE's, since it is *symmetric* (x $\overset{\delta}{\leftrightarrow}$ y implies y $\overset{\delta}{\leftrightarrow}$ x), *reflexive* (x $\overset{\delta}{\leftrightarrow}$ x) and transitive (x $\overset{\delta}{\leftrightarrow}$ y and y $\overset{\delta}{\leftrightarrow}$ z implies that x $\overset{\delta}{\leftrightarrow}$ z). It is because the relation is symmetric that we use the

double-pointed arrow. Note that the relation δ-equal is defined in terms of the function Val, which in turn is defined in terms of some primitive environment PE.

2.3 Functional Abstraction

Although in principle a programmer can specify any ob in the universe of discourse in terms of primitives and combinations of primitives, in practice the desirability of additional linguistic facilities is manifest. This section introduces an additional linguistic concept, called "functional abstraction", which suffices to permit the incorporation of user-coined definitions into a language. More precisely, functional abstraction makes it possible for the programmer to denote arbitrary obs in Λ directly, by identifiers of his own choosing.

Roughly speaking, functional abstraction means using an expression to specify a function by stipulating that some particular identifier in the expression is to be interpreted as a "dummy variable". The idea is a familiar one: In conventional notation, given an arithmetic expression such as

$$3 + x \qquad (2.3\text{-}1a)$$

we can write

$$f(x) = 3 + x \qquad (2.3\text{-}1b)$$

and refer thereafter to "the function f". Alternatively, we could write

$$g(x) = 3 + x \qquad (2.3\text{-}1c)$$

or

$$g(y) = 3 + y \qquad (2.3\text{-}1d)$$

each of which designates the same function (i.e., the same abstract object) that (2.3-1b) does, to wit, "that function of x which 3+x is".

Recall that the definition of a function involves specification of a domain, a codomain and a rule, the rule being a mapping from one set of objects into another. Clearly, in our example we intend the same rule, hence function, regardless of the name "f" or "g" we choose to call it and regardless of the name "x" or "y" we use to denote the dummy variable. Moreover, it is evident that (2.3-1b) does not define any mapping at all except relative to an environment in which all identifiers except the dummy variable are known. In other words, it does not specify a function unless the values denoted by the names "3" and "+" are known. For example, writing

$$h(x) = x+a$$

defines h only if the value of "a" is known. Clearly (2.3-1b) defines f. But can we write an expression that has the same value that f does?

λ-Expressions

Conventional notation is not well suited for formalizing the concept of functional abstraction because it does not allow us to designate a function without simultaneously giving it a name. In consequence, an expression such as f(x) can be interpreted in two ways. If "x" is interpreted as a dummy variable, then f(x) presumably denotes a function which we are naming "f". Alternatively, if "x" is interpreted as a name coupled with a specific value, then f(x) presumably denotes the application of the function named "f" to this value as argument.

Usually we can rely on context to decide which interpretation is correct. But even context is not infallible, as is evidenced by the following example. Consider the function P defined as

$$P\,[f(x)] = \frac{f(x) - f(0)}{x} \qquad (2.3\text{-}2a)$$

whenever "x" is not zero. Both the domain and range of P are number-to-number functions; in other words,

$$P \in (\text{number} \to \text{number}) \to (\text{number} \to \text{number}) \qquad (2.3\text{-}2b)$$

Now, even though the definition of P itself is unequivocable, it is not at all clear what is meant when we write P[f(x+1)]. Presumably the intent is to specify a function in terms of the dummy variable "x", but there are two possible interpretations:

(1) P[f(x+1)] means the function (P(g)), where g is the function of x that f(x+1) is, or

(2) P[f(x+1)] means the function of x that h(x+1) is, where h is the function (P(f)).

That there is a distinction between the two interpretations may be seen by considering the case in which f designates the function "square"; i.e., by letting $f(x) = x^2$. Then (1) yields

$$g(x) = (x+1)^2;\ \text{so}\ (P\ g)\ x = \frac{(x+1)^2 - (1)^2}{x} = x+2 \qquad (2.3\text{-}3a)$$

whereas (2) yields

$$h(x) = (P\ f)\ x = \frac{(x)^2 - (0)^2}{x} = x;\ \text{so}\ h(x+1) = x+1 \qquad (2.3\text{-}3b)$$

Conventional notation does not specify which interpretation is intended. By the end of this section it becomes apparent that the interpretation of (2.3-3a) may be specified by writing P[λx. f(x+1)], and that of (2.3-3b) by writing [λx. P f(x+1)], where in either case

$$P = \lambda t.\lambda u.\ \frac{t(u) - t(0)}{u} \qquad (2.3\text{-}3c)$$

(This example is cited by Curry and Feys (1958) on page 80.)

The problem here was mentioned earlier: The inability in available

notation to designate a function without simultaneously giving it a name. To define the function h that adds three to its argument we can write

$$h(x) = x+3 \qquad (2.3\text{-}4a)$$

but we have no way to write

$$h = \text{that function of "x" that "x+3" is} \qquad (2.3\text{-}4b)$$

other than in English. Following Church (1951), we write

$$h = \lambda x.\ x+3 \qquad (2.3\text{-}4c)$$

to designate the idea of (2.3-4a) or (2.3-4b). Here the λ signifies "function", and the period separates the dummy variable from the expression in terms of which the function is defined. Using this notation, we can rewrite equations (2.3-1b), (2.3-1c) and (2.3-1d) as

$$f = \lambda x.\ 3+x \qquad (2.3\text{-}5a)$$
$$g = \lambda x.\ 3+x \qquad (2.3\text{-}5b)$$
$$g = \lambda y.\ 3+y \qquad (2.3\text{-}5c)$$

An expression like those of (2.3-5) is called a λ-expression. By convention, the components of a λ-expression to the left and right of the period are called its bound variable and body, respectively. Note that a λ-expression is not referentially transparent with respect to its bound variable: The role of "λ" is directly analogous to that of "d" in (2.2-13) on page 2.2-48, in which "dx" signifies that "x" is to be interpreted as a bound variable. Hereafter we use the term "bound variable" consistently in lieu of the more colloquial term "dummy variable", frequently abbreviating it bv.

Informal Reduction Rule: In simple cases, the meaning of combinations involving λ-expressions is easily deducible. Consider, for example, the combination

$$(\lambda x.\ x*3+x)\ 5 \qquad (2.3\text{-}6a)$$

Since the rator denotes "the function of x that x*3+x is", it is clear that (2.3-6a) should have the same meaning as the expression

$$5*3+5 \qquad (2.3\text{-}6b)$$

which results when the rand "5" is substituted for each occurrence of the bound variable "x" in the body of the rator. We say that we have reduced (2.3-6a) to (2.3-6b).

More generally, if "M" and "N" are any two expressions and "y" is any identifier, we decree that

$$(\lambda y.\ M)\ N \rightarrow \underline{subst}\ (N,\ y,\ M) \qquad (2.3\text{-}7)$$

Here "→" should be read "is reducible to" and

$$\underline{subst}\ (a, b, c)$$

may be interpreted (somewhat naively) as "the result of substituting the AE a for the identifier b in the AE c". The rule may be invoked repeatedly, as in the example

$$\begin{aligned}&(\lambda x.\ 3-x)\ [(\underline{\lambda y}.\ 4*y+7)\ 2]\\ &\rightarrow (\underline{\lambda x}.\ 3-x)\ (4*2+7)\\ &\rightarrow 3-(4*2+7)\end{aligned}$$

In each line of this derivation we have underlined the λ that is about to be reduced, to ease the reader's task in following the derivation. We follow that practice hereafter.

A naive attitude towards subst suffices for the moment, but we need to be more careful to avoid inconsistencies when we come to treat reduction rules axiomatically. The problem has to do with multiple use of the same name and requires considerable care in formulating the rules. Note that subst is not a function in the universe of discourse but rather a function that transforms one expression into another expression. We call subst a meta-function.

Our semantic interpretation of reducibility is the obvious one: Given any two expressions E1 and E2 such that

$$E1 \rightarrow E2 \qquad (2.3\text{-}8)$$

we define the meaning (i.e. value) of E1 to be the same as that of E2. In other words, reducible expressions are equivalent in the sense that both denote the same ob in Ω. We have much more to say later about the relation "$\rightarrow$" and others similar to it.

Relation to PAL: Consider the PAL expression

$$\text{let } x = 5 \text{ in } x*3 + x \qquad (2.3\text{-}9a)$$

and the AE

$$(\lambda x.\ x*3 + x)\ 5 \qquad (2.3\text{-}9b)$$

Each involves evaluation of the AE "x*3 + x" with the understanding that "x" is to be replaced by "5". That is, each is equivalent to

$$\underline{subst}(\text{"5"}, \text{"x"}, \text{"x*3 + x"}) \qquad (2.3\text{-}9c)$$

Thus we regard a PAL expression such as

$$\text{let } y = N \text{ in } M \qquad (2.3\text{-}10a)$$

as just a "sugared" syntactic alternative specifying the same abstract transformation as does the more austere expression

$$(\lambda y.\ M)\ N \qquad (2.3\text{-}10b)$$

A third syntactic form in PAL which is also equivalent semantically is

M where y = N (2.3-10c)

Here as in (2.3-7) we are assuming that M and N may be any two expressions, and that "y" may be any identifier.

Function-Form Definitions: The preceding paragraph illustrates that λ-expressions accommodate the definition of variables in PAL. Although perhaps not immediately obvious, it is true that λ-expressions suffice also to accommodate programmer definition of functions. To see this, we need only observe that severing the name being given to a function from the function itself (as in (2.3-5)) implies the equivalence of

let g(x) = P in M (2.3-11a)

and

let g = λx. P in M (2.3-11b)

It follows from (2.3-10) and identification of "N" with "λx. P" and of "y" with "g" that a third equivalent expression is

(λg. M) (λx. P) (2.3-11c)

As an example, consider the PAL expression

let f(x) = x*3 + x in f 5 (2.3-12a)

In accordance with the desugaring of (2.3-11) and the reduction rule of (2.3-7), equivalent expressions are

(λf. f 5) (λx. x*3+x)
→ (λx. x*3+x) 5 (2.3-12b)
→ 5*3+5
$\overset{\delta}{\rightarrow}$ 20

so that the meaning of the PAL expression (2.3-12a) is 20.
Similarly, the value of

let f(x) = x*3 + x in f(f 5) (2.3-13a)

is deducible to be the same as the value of

[λf. f(f 5)] (λx. x*3+x)
→ (λx. x*3+x)[(λx. x*3+x) 5]
→ (λx. x*3+x) (5*3+5) (2.3-13b)
→ (5*3+5) * 3 + (5*3+5)
$\overset{\delta}{\rightarrow}$ 80

and hence equal to 80.

One other notational point need be made: We suggested on page 2.1-17 the possibility of writing

$$f() = 3$$

to define a constant function f with value 3, so we provide the notation

$$\lambda(). 3$$

to denote f's value. Evidently such a function is zero-adic, so for convenience we decree that it can be applied to only the 0-tuple nil. That is,

$$\lambda(). M$$

denotes a constant function such that the combination

$$(\lambda(). M)\ nil$$

denotes the same value as does M.

Curried Functions: If we abstract on an expression that is itself a λ-expression, the resulting expression specifies a curried function. For example, abstracting twice on the expression

$$x + y \qquad (2.3\text{-}14a)$$

produces first

$$\lambda y.\ x+y \qquad (2.3\text{-}14b)$$

and then

$$\lambda x.\ (\lambda y.\ x+y) \qquad (2.3\text{-}14c)$$

It follows in accordance with the reduction rule of (2.3-7) that the combination

$$[\lambda x.\ (\lambda y.\ x+y)]\ 3 \qquad (2.3\text{-}15a)$$

is equivalent to

$$\lambda y.\ 3+y \qquad (2.3\text{-}15b)$$

which in turn specifies the function that adds 3 to its argument. Thus the meaning of

$$\begin{aligned} &\{[\lambda x.\ (\lambda y.\ x+y)]\ 3\}\ 5 \\ &\rightarrow (\lambda y.\ 3+y)\ 5 \\ &\rightarrow 3 + 5 \\ &\xrightarrow{\varepsilon} 8 \end{aligned} \qquad (2.3\text{-}15c)$$

is the ob 8. PAL requires fewer parentheses than used in the first line of (2.3-15c), permitting

$$(\lambda x.\ \lambda y.\ x+y)\ 3\ 5 \qquad (2.3\text{-}15d)$$

with identical meaning.

PAL notation for defining function-producing functions is a natural

extension of the syntactic sugaring involved in (2.3-11). Specifically, the view that the construction

```
let g(x) = N in ...                                    (2.3-16a)
```

is sugaring for

```
let g = λx. N in ...                                   (2.3-16b)
```

leads us to adopt

```
let g(x)(y) = P in ...                                 (2.3-16c)
```

as sugaring for

```
let g = λx. λy. P in ...                               (2.3-16d)
```

(All parentheses in equations (2.3-16) are optional in PAL.) Thus the PAL expression

```
let Sum x y = x + y  in  Sum 3 5                       (2.3-17a)
```

is equivalent to the combination

```
(λ Sum. Sum 3 5) (λx. λy. x+y)                         (2.3-17b)
```

and hence to (2.3-15c). Similarly,

```
let Twice f x = f(f x)  in  Twice Sqrt 16              (2.3-18a)
```

is equivalent to

```
(λ Twice. Twice Sqrt 16) [λf. λx. f (f x)]
→ [λf. λx. f (f x)] Sqrt 16                            (2.3-18b)
→ [λx. Sqrt (Sqrt x)] 16
→ Sqrt (Sqrt 16)
```

which in turn denotes the ob 2, relative to an environment in which the identifier "Sqrt" denotes the function that returns the square-root of its argument.

Referential Opacity: We have observed that functional abstraction adds definitional facilities to a language, but that λ-expressions are not referentially transparent. It is interesting to note that this encroachment of referential opacity is unavoidable. That is to say, it is not possible to accommodate programmer definitions by functional application alone, even if we arm ourselves for this purpose with new and arbitrary basic functions.

To see that this is the case, consider the combination

```
Define (y, M, N)                                       (2.3-19a)
```

and require for all expressions "M" and "N" that its value relative to any environment must be the same as the value of

$$(\lambda y.\ M)\ N \qquad (2.3\text{-}19b)$$

Regardless of the function denoted by the identifier "Define", referential transparency requires that the value of (2.3-19a) depend on "M" and "N" solely through their values. Thus the values denoted by

$$\text{Define } (y,\ 2+y,\ 3) \qquad (2.3\text{-}20a)$$

and

$$\text{Define } (y,\ y+y,\ 3) \qquad (2.3\text{-}20b)$$

relative to an environment in which "y" denotes 2 must be identical, whereas this is not true of the expressions

$$(\lambda y.\ 2+y)\ 3 \qquad (2.3\text{-}21a)$$

and

$$(\lambda y.\ y+y)\ 3 \qquad (2.3\text{-}21b)$$

Even more cogently, (2.3-21a) and (2.3-21b) are meaningful relative to an environment in which "y" does not denote a value, whereas in such a case (2.3-20a) and (2.3-20b) are undefined

Applicative Expressions

Up to this point we have been imprecise in our treatment of λ-expressions on two counts. First, we have been vague about the class of expressions with which we are concerned, and second we have been inaccurate in our treatment of substitution. We remedy the first of these defects in this subsection, in preparation for the axiomatic development of "reduction" which follows.

In order to distinguish between the austere language of the λ-calculus and the sugared constructs of PAL, we call formulas of the λ-calculus "applicative expressions", abbreviated AE's. The class of AE's is an extension of the class previously defined to have that name on page 2.2-49.

Definition: An applicative expression (AE) is a structured object. It is either an
identifier, or it is a
combination, which has a
rator, which is an AE, and a
rand, which is an AE, or it is a
λ-expression, which has a
by-part, which is a variable, and a
body, which is an AE.

As is the case for all structured objects, this definition implies predicates, selectors and constructors, and makes explicit the requirements on any representations of AE's. The term "variable" used here is defined on page

Semantic Trees: Starting on page 2.2-49 we introduced the idea of a semantic tree, suggesting that such a tree is an acceptable representation of an AE as that term was defined on page 2.2-49. Clearly our new definition of AE to include λ-expressions merely requires that we permit also λ-nodes in addition to γ-nodes. As before, we continue to write ℑ nodes, with the understanding that each ℑ node is an abbreviation for a complex of γ nodes.

By way of example, the AE which in (2.3-15d) we wrote

$$(\lambda x.\ \lambda y.\ x+y)\ 3\ 5$$

may be represented by the tree

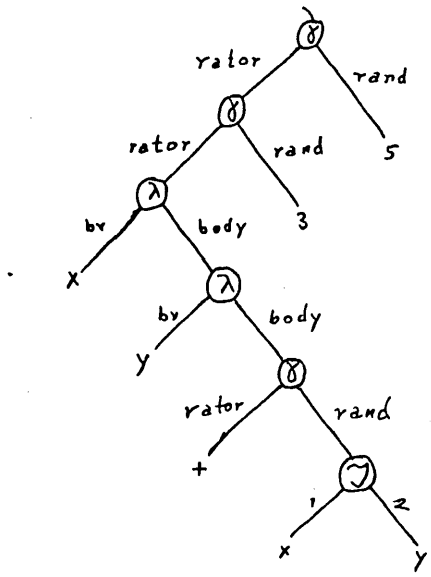


Figure 2.3-1: Tree Form of (2.3-15d)

Here each branch is explicitly labelled. Hereafter we refrain from such labelling, using instead the following conventions:

(a) The left and right sons of a γ node designate the rator and rand, respectively.

(b) The left and right sons of a λ node designate the bv-part and body, respectively.

(c) The sons of a ℑ-node are numbered from left to right.

(These conventions may of course be overruled by labelling, if necessary.) Using these conventions, the tree form of

$$(\lambda f.\ f\ 5\ 7)\ (\lambda x.\ \lambda y.\ x+y) \qquad (2.3\text{-}22)$$

is

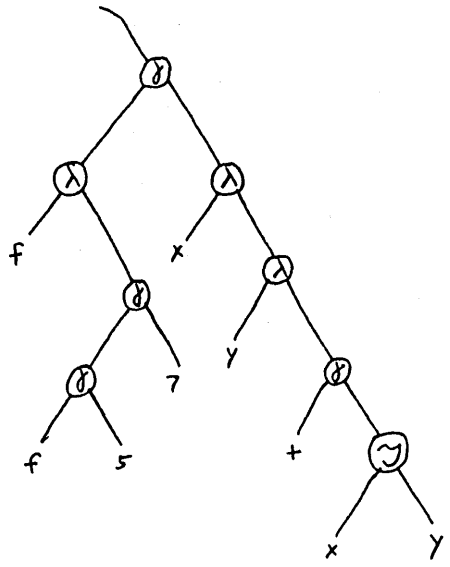


Figure 2.3-2: Tree Form of (2.3-22)

Although we can eliminate τ nodes in favor of γ nodes, it is important to note that we can not eliminate λ nodes in favor of γ nodes, because (as we have shown) definitions can not be accommodated by combinations. Thus the notion of functional abstraction is "linguistically orthogonal" to the notion of functional application, in the sense that one cannot be eliminated in favor of the other. More precisely, one cannot forego the notion of functional abstraction without also foregoing user-coined definitions.

Syntax of AE's: In addition to the tree representation just discussed, we need also a linear representation that can be written as a line of text, both for usage in this text and for inclusion in PAL programs. The linear representation must meet the requirement that any representation of a structured object must meet: that the relevant predicates, selectors and constructors must be realizable. It is the predicates that are the problem, and another way to state the requirement is that it must be possible to construct the semantic tree from any linear AE.

Clearly, sufficient information is provided if we demand that every AE which is a constituent of an AE be enclosed in parentheses. As usual, however, we are interested in economizing on parentheses to improve readability. To this end we adopt the following conventions:

(1) If a λ-expression occurs either as a rator or as a rand, it must be enclosed in parentheses. Otherwise, parentheses are optional.

(2) The body of a parenthesized λ-expression is terminated by the closing parentheses.

(3) When the body of one λ-expression is a second, unparenthesized λ-expression, the bodies of both

λ-expressions terminate together.

These conventions are in substantial agreement with those of PAL. Examples illustrating these conventions follow. As usual, we feel free to employ brackets and/or braces in lieu of parentheses when doing so improves visual clarity.

$$[\lambda_1 x.\ \underbrace{F\ (x+3)\ (\lambda y.\ g)}_{\text{body 1}}]\ 3$$

$$\lambda_1 x.\ \underbrace{(\lambda_2 y.\ \underbrace{y*x}_{\text{body 2}})\ 6}_{\text{body 1}}$$

$$[\lambda_1 x.\ \underbrace{\lambda_2 y.\ \underbrace{F\ (x+3)\ y}_{\text{body 2}}}_{\text{body 1}}]\ 5\ 7$$

Although subscripts on λ's are not a part of our syntax, we sometimes use them (as above) to indicate particular instances of λ-expressions.

We remark in passing that PAL syntax permits the writing of AE's that accord with the foregoing conventions, except that "fn" is used in lieu of "λ". PAL syntax is richer, however, in that "fn" may occur also in constructions (such as assignment statements) which are not AE's.

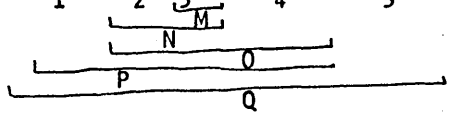
Free and Bound Variables: The semantics of a high-level language such as PAL depend critically on the scope of a definition, by which we mean those parts of the text of a program within which the variable defined by the definition is to be associated with that ob which is denoted by the right side of the definition. When the definitional facilities of a language are modeled on the λ-calculus, as in the case of PAL, all questions about scope ultimately depend on the concepts of free and bound occurrences of an identifier. Since the structure of an AE may be quite complicated and include several λ-expressions each having the same identifier as its bound variable, we must be careful to define these concepts very precisely.

> Definition: Let M and N stand for any two AE's and let x stand for any identifier. Then an occurrence of x is said to be free in an AE if and only if it can be proved to be so by means of the following rules:
>
> 1. The occurrence of x in the AE "x" is free.
> 2. Any free occurrence of x in either M or N is free in the combination (M N).
> 3. If y is any identifier distinct from x, then any free occurrence of x in M is free in λy.M.

Definition: An occurrence of x is said to be bound in an AE if and only if it is not free in that AE.

An equivalent definition of bound is the following: An occurrence of an identifier x in an AE is bound if it is part of a λ-expression whose bv-part is x. The smallest such λ-expression is referred to as the λ-expression that binds the occurrence. The utility of the definition lies in specification of the association of variables. Consider the AE

$$[\lambda x_1.\ (\lambda x_2.\ x_3 * a)\ (x_4 - b)]\ (x_5 + c) \qquad (2.3\text{-}23)$$



and let M, N, O, P and Q denote the subexpressions indicated by the underbars. Using the integer subscripts as labels to distinguish among the various occurrences of x, the definition implies that

occurrence 1 is bound in P;
occurrence 2 is bound in N;
occurrence 3 is free in M but bound in N, O, P and Q;
occurrence 4 is free in O but bound in P and Q; and
occurrence 5 is free in O.

Note that every free occurrence of x in (2.3-23b) except occurrence 5 switches status from free to bound as one considers larger and larger subexpressions. Presumably, occurrence 5 becomes bound in some still larger λ-expression that encompasses Q within its body.

It follows from the reduction axioms for AE's that each free occurrence of x is to be associated with the bound variable of the particular λ-expression at which its status switches. Thus the association is as indicated by the subscripts in

$$[\lambda x_1.\ (\lambda x_2.\ x_2 * a)\ (x_1 - b)]\ (x_3 + c) \qquad (2.3\text{-}23c)$$

The association is described in words by the

Definition: The scope of a bound variable is the entire body-part of the λ-expression to which it belongs, with the specific exception of each included λ-expression whose bound variable is that same identifier.

Alternatively, we may say that any occurrence of an identifier is "bound by" the smallest enclosing λ-expression whose bound variable is that identifier.

The association of variables may be seen in terms of semantic trees in a particularly straightforward manner: To find the binding λ for any occurrence of an identifier, follow up the tree looking for a λ node whose left son is that identifier. If none is found, the occurrence is free in the AE.

Reduction Axioms

Voids in the scope of a bound variable may seem strange at first, but on reflection they turn out to be a natural concomitant of requiring that the rules for interpreting a λ-expression be independent of its context. For example, agreement that the AE's

$$\lambda x.\ x*3 \qquad (2.3\text{-}24a)$$

and

$$\lambda y.\ y*3 \qquad (2.3\text{-}24b)$$

denote the same function regardless of the context in which they occur implies that the meaning of

$$[\lambda x.\ (\lambda x.\ x*3)\ (x-2)]\ 5 \qquad (2.3\text{-}25a)$$

and

$$[\lambda x.\ (\lambda y.\ y*3)\ (x-2)]\ 5 \qquad (2.3\text{-}25b)$$

must be the same. It is obvious that "5" should not be substituted for "y" in (2.3-25b), from which it follows that "5" must not be substituted for the corresponding occurrence of "x" in (2.3-25a).

The concept of "contextual independence" is closely related to the concept of referential transparency; indeed, the former is just a slightly restricted version of the latter. Referential transparency implies freedom to substitute any one subexpression for any other, provided only that both have the same meaning. Contextual independence implies identical freedom except that substitution is restricted to "whole AE's"; specifically, we prohibit splitting a λ-expression into its bound variable and body parts and substituting in the two parts independently. (The same restriction applies in the case of referential transparency: One may not substitute "5" for "1+4" in "21+43" to get "243".)

Formal Postulates: Just as we made a conscious decision to require referential transparency when adopting rules about the meaning of a combination, we now make a conscious decision to require that rules governing the meaning of AE's be contextually independent. The objective behind this decision is to enhance the perspicuity and flexibility of the AE's as a language.

The decision leads us to make our informal reduction rule for AE's precise through postulation of the following axioms:

Axiom α: If x and y are identifiers and M is an AE in which y does not occur free, then in any context

$$(\lambda x.\ M) \xrightarrow{\alpha} \lambda y.\ \underline{subst}\ (y,\ x,\ M)$$

Axiom β: If M and N are any AE's and x any

identifier, then in any context

$$(\lambda x.\ M)\ N \xrightarrow{\beta} \underline{subst}\ (N,\ x,\ M)$$

It is important that these axioms hold in any context. Here Axiom α formalizes the notion that the dummy variable of a function is arbitrary; i.e. that (2.3-24a) and (2.3-24b) should be equivalent in any context. Axiom β formalizes the notion that functional application involves substitution of an actual for a dummy parameter.

> Definition: If $P \xrightarrow{\beta} Q$ we say that P is β-reducible to Q, or that Q is β-expandable to P. If $P \xrightarrow{\alpha} Q$ we say that P is α-convertible to Q.

Since α-conversion is symmetric, it is clear that $P \xrightarrow{\alpha} Q$ implies that $Q \xrightarrow{\alpha} P$.

Hereafter we use $\xrightarrow{\alpha}$ and $\xrightarrow{\beta}$ in an extended sense, writing $P \xrightarrow{\alpha} Q$ if the conversion is either on all of P or on some part of P, and similarly for $\xrightarrow{\beta}$. For example, we might write

$$\lambda x.\ (\lambda y.\ x+y)\ 2 \xrightarrow{\beta} \lambda x.\ x+2$$

even though the reduction is on only part of the AE on the left.

The Substitution Rule: The foregoing statement of the axioms is misleadingly simple, primarily because it shifts the burden of being careful onto the definition of subst. In particular, we must now define subst in such a way that we guard against inconsistencies encroaching through "in any context".

The danger of inconsistency turns on the possibility of a "conflict of variables" arising through substitution. Basically, conflict of variables means inadvertent binding of a variable that should be free, and may occur in two ways. The first involves Axiom α, and is evidenced by the AE

$$[\lambda a.\ (\lambda x.\ a+x)]\ 3\ 5 \qquad (2.3\text{-}26a)$$

the desired interpretation of which is

$$(\lambda x.\ 3+x)\ 5 \xrightarrow{\beta} 3+5 \qquad (2.3\text{-}26b)$$

But if we choose "y" in Axiom α to be "a" and misapply subst to the inner λ-expression of (2.3-26a) in violation of the condition "y not free in M", we get

$$[\lambda a.\ (\lambda a.\ a+a)]\ 3\ 5 \qquad (2.3\text{-}26c)$$

which presumably would be interpreted as (5+5). The inconsistency arises because although the occurrence of "a" in the inner λ-expression of (2.3-26a) is free, the corresponding occurrence in (2.3-26b) is bound. Note that Axiom α as stated prohibits this substitution. However, there is nothing in the axiom to prohibit substituting x for a in the outer λ-expression, leading to

$$\{\lambda x.\ [\underline{subst}\ (x,\ a,\ (\lambda x.\ a+y))]\}\ 3\ 5 \qquad (2.3\text{-}26d)$$

This is permissible, and it should be clear to the reader that subst must be very careful to insure that (2.3-26d) is equivalent to (2.3-26b).

The second way in which a free variable may become bound inadvertently involves Axiom β. Consider the meaning of the AE

$$(\lambda x.\ \lambda y.\ x+y)\ y\ 3 \qquad (2.3\text{-}27a)$$

in a context defining y. Proceeding naively, we might produce

$$\begin{aligned}&(\underline{\lambda x}.\ \lambda y.\ x+y)\ y\ 3\\ &\xrightarrow{\beta} (\underline{\lambda y}.\ y+y)\ 3\\ &\xrightarrow{\beta} 3+3\end{aligned} \qquad (2.3\text{-}27b)$$

whereas using Axiom α first could lead to

$$\begin{aligned}&(\lambda x.\ \underline{\lambda y}.\ x+y)\ y\ 3\\ &\xrightarrow{\alpha} (\underline{\lambda x}.\ \lambda u.\ x+u)\ y\ 3\\ &\xrightarrow{\beta} (\underline{\lambda u}.\ y+u)\ 3\\ &\xrightarrow{\beta} y+3\end{aligned} \qquad (2.3\text{-}27c)$$

These two evaluations would be equivalent only in a context in which y is bound to 3. Since the rules are to be applicable in any context, we have erred. The inconsistency arises because in line 2 of (2.3-27b) we have inadvertently bound the free occurrence of "y" by carrying it inside the body of a λ-expression whose bound variable is also "y".

Difficulty with conflicts of variables may be obviated by defining subst in such a way that Axiom α is invoked (as in 2.3-27c) whenever the possibility of a clash of bound variables arises. The following recursive definition accomplishes this objective.

Definition: Let N and M be AE's and x an identifier. Then by

$$\underline{subst}\ (N,\ x,\ M)$$

we mean

(a) if M is an identifier, then
  (a.1) if it is x, then N
  (a.2) and otherwise M

(b) if M is the combination (P Q), then the combination
  [subst (N, x, P)] [subst (N, x, Q)]

(c) if M is a λ-expression, then
  (c.1) if it is (λx. P) then M
  (c.2) if it is (λy. P) where y is not x then

let z be some identifier (other than x)
not occurring free in either P or N and
the value is
λz. subst (N, x, subst (z, y, P))

Cases (a) and (b) are self-evident and account for situations in which M is an identifier or a combination. Case (c.1) accounts for the situation in which M is a λ-expression in which all occurrences of "x" are bound, so that in fact no substitution is to be performed. Case (c.2) avoids conflicts of variables by changing the bound variable of M from "y" to "z" before substitution of N for "x". By way of example we consider again part of (2.3-27a):

```
(λx. λy. x+y) y
= subst [y, x, (λy. x+y)]
= λz. subst [y, x, subst (z, y, x+y)]          (2.3-28a)
= λz. subst (y, x, x+z)
= λz. y+z
```

Even this lengthy derivation leaves out many steps. Consider

subst (y, x, x+z) (2.3-28b)

in which we are substituting into the AE "x+z". To make clear that this is an AE, we must display it as a combination, such as

Add x z (2.3-28c)

whose rator is "(Add x)" and whose rand is z. Then (2.3-28b) is replaceable by

subst [y, x, (Add x z)] (2.3-28d)

and from rule (b) in the definition of subst to

{subst [y, x, (Add x)]} {subst (y, x, z)} (2.3-28e)

The rand in this combination can be replaced by "z" by rule (a.2), and rule (b) is needed again for the rator.

The substance of the definition may be summarized in words (no longer naively) by stating that

subst (N, x, M) means to substitute N for each free occurrence of "x" in M, changing bound variables whenever necessary to avoid conflict.

Normal Form: Recall that in section 2.2 we discussed evaluation of AE's that are pure combinations not involving λ-expressions. To make the results of that section available to us in the present discussion, we introduce the

Axiom 5: If M and N are AE's which do not include λ-expressions, then

$$M \xrightarrow{\delta} N$$

if $\underline{Val}(M) = \underline{Val}(N)$. We say that $\underline{M}$ is δ-convertible to $\underline{N}$.

As with α-conversion and β-reduction, we extend the use of $\xrightarrow{\delta}$ for the case in which the conversion is on some part of the AE rather than on the whole, permitting

$$\lambda x.\ x+(2+2) \xrightarrow{\delta} \lambda x.\ x+4$$

The function Val used as part of the definition of δ-conversion is defined on page 2.2-54.

It is clear that $\xrightarrow{\alpha}$ and $\xrightarrow{\delta}$ are equivalence relations and that $\xrightarrow{\beta}$ is not. As we need a single equivalence relation encompassing all three types of conversion, we introduce the

Definition: An AE $\underline{M}$ is said to be directly convertible to the AE $\underline{N}$, written $M \approx N$, if any one of the following holds:

$$M \xrightarrow{\alpha} N$$
$$M \xrightarrow{\beta} N$$
$$N \xrightarrow{\beta} M$$
$$M \xrightarrow{\delta} N$$

Definition: Two AE's $\underline{M}$ and $\underline{N}$ are said to be equivalent, written $M \simeq N$, if there exists a sequence

$$M \equiv M_0 \approx M_1 \approx M_2 \approx \ldots \approx M_n \equiv N$$

The implication of this last line is that $\underline{M}_0$ (which is the same AE as $\underline{M}$) is directly convertible to $\underline{M}_1$, which is directly convertible to $\underline{M}_2$, ..., which is directly convertible to $\underline{M}_n$, which is the same AE as $\underline{N}$. Clearly "$\simeq$" is an equivalence relation on AE's.

An advantage of including β-expansion as well as β-reduction in the definition of equivalence is that it permits us to carry out transformations inverse to those of Axiom β. For example, we may abstract on the subexpression "5" to replace

$$5+3*5 \qquad (2.3\text{-}32a)$$

in any context by

$$(\lambda x.\ x+3*x)\ 5 \qquad (2.3\text{-}32b)$$

The technique is especially useful when an expression contains many occurrences of a complicated subexpression, as (to a minor degree) in

$$(5+3*5)/(7+3*7)-(9+3*9)$$

Abstracting twice, we obtain the equivalent expression

$$[\lambda f.\ (f\ 5)/(f\ 7) - (f\ 9)]\ (\lambda x.\ x+3*x)$$

Recall the definition of equivalence relation in Section 2.1. The significance of an equivalence relation on AE's is that it implies a partitioning of the set of AE's into disjoint subsets, called equivalence classes. The reduction rules defining "$=$" have been carefully chosen so that our intuition about "meaning" accords with a definition of value of an AE built on these equivalence classes. Essentially, any two AE's in the same equivalence class have the same value. (This point is pursued shortly.) Thus the reduction rules provide a mathematical basis in terms of which we may hope to extend the concept of value from just primitives and combinations (as in the definition of page 2.2-51) to AE's in general. We understand the value of an AE to be undefined if any equivalent AE is undefined.

Merely to say that all AE's in an equivalence class have the same value does not, of course, pin down what that value is. We must also exhibit some specific member of the class whose value is ascertainable on other grounds. For example, we discover that the value of

$$(\lambda x.\ 3+x)\ 5 \xrightarrow{\beta} 3+5 \xrightarrow{\delta} 8$$

is 8. As far as the δ-conversion is concerned, "2+6" would be just as valid a final result as "8".

An AE is said to have (or be in) normal form if no β-reduction is possible. For example, each of the AE's

$$3 + 5 \qquad (2.3\text{-}33a)$$

$$\lambda x.\ 3+x \qquad (2.3\text{-}33b)$$

$$\lambda x.\ x\ (\lambda y.\ 3+y) \qquad (2.3\text{-}33c)$$

has normal form. We have then the following

> Definition: An AE is said to be in normal form if it is either
>
> (1) an identifier, or
>
> (2) a combination in which the rator is not a λ-expression and in which both rator and rand are in normal form, or
>
> (3) a λ-expression whose body is in normal form.

The task of evaluating an AE in normal form is relatively straightforward. In accordance with the definitions of page 2.2-51, the value in cases (1) and (2) above is implied by the postulates establishing the universe of discourse. In case (3), if the λ-expression is "$\lambda x.M$" the value is "the function of x that M is". It follows that one approach to the problem of evaluating an arbitrary AE involves first trying to produce an equivalent AE having normal form, and then evaluating the result.

A difficulty arises, however, from the fact that not all AE's are reducible to normal form; witness

$$(\lambda u.\ u\ u)\ (\lambda u.\ u\ u) \qquad (2.3\text{-}34a)$$

Although this AE seems somewhat pathological, it cannot be excluded on grounds that $(\lambda u.\ u\ u)$ is vacuous. For example, consider

$$(\lambda u.\ u\ u)\ \text{Twice Sqrt}\ x \xrightarrow{\beta} \text{Twice Twice Sqrt}\ x \qquad (2.3\text{-}34b)$$

which in turn is equivalent, given the definition of "Twice" in (2.3-18), to

$$\text{Sqrt (Sqrt (Sqrt (Sqrt x)))} \qquad (2.3\text{-}34c)$$

In addition, we see in section 2.4 that a construct similar to (2.3-34) is of fundamental interest in the study of recursion. The remainder of this chapter is devoted to exploring the implications of this difficulty from a theoretical point of view. The problem of evaluating AE's is approached somewhat more pragmatically in Chapter 3.

Order of Reduction

An important attribute of functions is the flexibility with which nested functional applications can be evaluated. For example, the tree contractions of Figure 2.2-7 (on page 2.2-50) can be reordered in arbitrary ways without affecting the final value. It is referential transparency that guarantees that this independence of order carries over to any semantic trees involving only γ nodes. (As usual, we note that $\Im$ nodes can be replaced by equivalent subtrees involving only γ nodes. Equivalently, we can stipulate that the components of a $\Im$ node may be evaluated in any order.) An effective procedure for evaluating such a tree is summarized in words by the rule:

(a) Evaluate the rator and rand (in either order)
(b) and then apply the value of the rator to the value of the rand.

In such a world without λ's, every order of evaluation consistent with the rule produces the same final result, including the result of being undefined in case a function is applied to some argument not within its domain.

Church-Rosser Theorem: We now ask whether or not the similarity between referential transparency on the one hand and contextual independence on the other is strong enough to induce a comparable insensitivity to the order in which an AE is reduced. Unfortunately the answer at best is a nicely qualified "yes".

The precise answer to the question is embodied in the fundamental result of the λ-calculus, the Church-Rosser Theorem. As a preliminary, we introduce

Definition: Given an AE M, a reduction sequence on M

is a sequence of AE's

$$M \equiv M_0 \approx M_1 \approx M_2 \approx \ldots$$

Definition: A reduction sequence is said to terminate if its last AE is in normal form.

Definition: Two AE's M and N are said to be congruent, written $M \overset{\alpha}{\leftrightarrow} N$, if there is a sequence

$$M \equiv M_0 \overset{\alpha}{\rightarrow} M_1 \overset{\alpha}{\rightarrow} \ldots \overset{\alpha}{\rightarrow} M_n \equiv N$$

Given any AE, a problem of obvious interest is to find a terminating reduction sequence on that AE. Frequently there is a choice of order of evaluation, since at any stage of the reduction one may have an AE containing more than one combination whose rator is a λ-expression. In such cases one must make an arbitrary choice, and there seem to be three possible consequences of that choice:

(a) One order of evaluation may terminate, while another fails to terminate. (Clearly for (2.3-34) there is no terminating reduction sequence.)

(b) Two orders of evaluation may produce non-congruent results. (We do not mind if, say, one evaluation leads to $(\lambda x.x+1)$ and another leads to $(\lambda y.y+1)$.)

(c) One order of evaluation may be less efficient than another, in that it takes more steps before it terminates.

The Church-Rosser theorem says about all there is to say about (a) and (b), but we have little to say now about (c). (We come back to it in Chapter 3.)

Before presenting the theorem, we find it useful to single out one particular order of evaluation, in which we proceed from left to right.

Definition: A reduction sequence is said to be in normal order if, at each step, the left-most possible λ is reduced.

A simple example of reduction to a normal form in normal order is

$$\begin{aligned}
&(\lambda f.\ f\ 6)\ [\lambda u.\ (\lambda v.\ u+v)\ 4] \\
&\overset{\beta}{\rightarrow} [\lambda u.\ (\lambda v.\ u+v)\ 4]\ 6 \\
&\overset{\beta}{\rightarrow} (\lambda v.\ 6+v)\ 4 \\
&\overset{\beta}{\rightarrow} 6+4
\end{aligned}$$

Note that the final result is the same as that in

$$\begin{aligned}
&(\lambda f.\ f\ 6)\ [\lambda u.\ (\lambda v.\ u+v)\ 4] \\
&\overset{\beta}{\rightarrow} (\lambda f.\ f\ 6)\ (\lambda u.\ u+4) \\
&\overset{\beta}{\rightarrow} (\lambda u.\ u+4)\ 6
\end{aligned}$$

$\xrightarrow{\beta}$ 6+4

We now state (but do not prove) a theorem equivalent to the Church-Rosser Theorem:

A All sequences of reductions on an AE that terminate do so on congruent AE's.

B If there is any sequence of reductions that terminates, then reduction in normal order is guaranteed to terminate.

This theorem is stated, although in a much different appearing form, in Curry and Feys (1958). It is proved for reduction rules involving only α-conversion and β-reduction, but not δ-conversion. A rather complex example of normal order reduction is

$$
\begin{aligned}
&[\underline{\lambda x}.\ \lambda y.\ \lambda z.\ \lambda w.\ x\ z\ (y\ z\ w)]\ (\lambda x.\ \lambda y.\ y)\ (\lambda x.\ \lambda y.\ x)\\
&\xrightarrow{\beta} [\underline{\lambda y}.\ \lambda z.\ \lambda w.\ (\lambda x.\ \lambda y.\ y)\ z\ (y\ z\ w)]\ (\lambda x.\ \lambda y.\ x)\\
&\xrightarrow{\beta} \lambda z.\ \lambda w.\ (\underline{\lambda x}.\ \lambda y.\ y)\ z\ [(\lambda x.\ \lambda y.\ x)\ z\ w]\\
&\xrightarrow{\beta} \lambda z.\ \lambda w.\ (\underline{\lambda y}.\ y)\ [(\lambda x.\ \lambda y.\ x)\ z\ w]\\
&\xrightarrow{\beta} \lambda z.\ \lambda w.\ (\underline{\lambda x}.\ \lambda y.\ x)\ z\ w\\
&\xrightarrow{\beta} \lambda z.\ \lambda w.\ (\underline{\lambda y}.\ z)\ w\\
&\xrightarrow{\beta} \lambda z.\ \lambda w.\ z
\end{aligned}
$$

The reader should convince himself that any other order of reduction he may choose terminates on an AE congruent to $(\lambda z.\ \lambda w.\ z)$.

Insight into Part B of the theorem may be gained by considering the case

$$A \equiv (\lambda x.\ \lambda y.\ x)\ z\ [(\lambda u.\ u\ u)\ (\lambda v.\ v\ v)] \qquad (2.3\text{-}35a)$$

Here normal order reduction yields

$$A \xrightarrow{\beta} (\underline{\lambda y}.\ z)\ [(\lambda u.\ u\ u)\ (\lambda v.\ v\ v)] \xrightarrow{\beta} z \qquad (2.3\text{-}35b)$$

whereas if we attempt first to reduce the AE in brackets (see (2.3-34)) we become involved in a non-terminating reduction. The quintessence of proceeding in normal order is that one postpones any attempt to eliminate a λ by β-reduction until that λ has reached a position where no possibility of its being discarded remains. Indeed, if we restrict the class of AE's by requiring that the body of any λ-expression must contain at least one free occurrence of its bound variable, no subexpression of an AE can ever be discarded. (Note that $(\lambda x.\ \lambda y.\ x)$ and $(\lambda x.\ \lambda y.\ y)$ do not satisfy this requirement.) Were we to accept this restriction, Part B of the Church-Rosser Theorem could be strengthened to read:

B* If any order of reduction of an AE terminates, there exists an integer $\underline{k}$ such that no more than $\underline{k}$ successive λ-reductions to that AE are possible.

The thrust of the Church-Rosser Theorem may be summarized informally as

follows:

1. Some AE's can be reduced to normal form, but not all of them.
2. If a normal form exists, it is unique to within an alternate choice of bound variables.
3. If discarding of subexpressions is impossible and a normal form exists, every order of reduction terminates.
4. If a normal form exists, a reduction in normal order always produces it in a finite number of steps.

Conditionals: The fact that not all AE's can be reduced to normal form is analogous to the fact that not all functions can be applied to all arguments; for example, numbers cannot be divided by zero. Thus insofar as sensitivity to order of procedure is concerned, reduction of AE's differs from evaluation of "pure" combinations only when discarding subexpressions is permitted. Since invariance to order is appealing, why not disallow any λ-expression whose body does not involve at least one free occurrence of its bound variable?

In answer to this question we observe that if E1 and E2 are AE's, then

$$B\ E1\ E2 \xrightarrow{\beta} \begin{cases} E1, & \text{if } B \equiv \lambda x.\ \lambda y.\ x \\ E2, & \text{if } B \equiv \lambda x.\ \lambda y.\ y \end{cases} \qquad (2.3\text{-}36a)$$

Thus the AE of (2.3-36a) embodies the concept of a conditional expression, written in PAL as

$$B \rightarrow E1 \mid E2 \qquad (2.3\text{-}36b)$$

As mentioned earlier, the value of a PAL expression such as (2.3-36b) is defined to be that of E1 or of E2 depending on whether B denotes true or false. If we adopt (λx. λy. x) as a representation in AE's of the ob true, and (λx. λy. y) as the representation of the ob false, then (2.3-36b) may be viewed as syntactic sugaring for (2.3-36a).

It may seem strange at first to think of truthvalues playing a role as functions (specifically, as curried functions that select one of their two arguments and discard the other). But there is nothing inconsistent in doing so. Indeed, we have already agreed that abstract objects are "bundles of properties", and that the properties are established by definition. Moreover, the definition on page 2.1-20 serves only to establish properties of truthvalues as arguments and leaves open all questions related to how truthvalues are to transform when (or if) they occur as functions. Thus we are free (if so inclined) to expand the properties assigned to truthvalues by postulating that, for all obs a and b in Ω:

$$\begin{aligned} \underline{\text{true}}\ a\ b &= a \\ \underline{\text{false}}\ a\ b &= b \end{aligned} \qquad (2.3\text{-}37)$$

Actually, truthvalues in PAL are not defined as operators. But the decision is clearly arbitrary and could have been made differently. Although in Chapter 3 we select still another method for dealing with conditionals such as (2.3-36b), in dealing with such expressions in the rest of this chapter we regard

$$B \text{ -> } E \mid F \qquad (2.3\text{-}38a)$$

as a syntactic variant for the combination

$$Q\ B\ E\ F \qquad (2.3\text{-}38b)$$

where Q is defined as follows:

$$\begin{aligned} Q\ \text{true} &\xrightarrow{\delta} \lambda x.\ \lambda y.\ x \\ Q\ \text{false} &\xrightarrow{\delta} \lambda x.\ \lambda y.\ y \end{aligned} \qquad (2.3\text{-}39)$$

Q is undefined for all other arguments. The advantage of introducing this Q, as opposed to using the definitions of (2.3-37), is that we want expressions such as (2.3-38a) to be undefined in the case in which B fails to denote a truthvalue. The definition of (2.3-39) has that effect.

As one might expect, conditional expressions are sensitive to order of evaluation. Specifically, we require that the premise (B) must be evaluated before either "arm" (E1 or E2) in order to avoid encountering an expression whose value may be undefined, as in the example

$$(a \text{ eq } 0) \text{ -> } a \mid (1/a) \qquad (2.3\text{-}40)$$

As an example of the use of the Q defined in (2.3-39), we replace the a of (2.3-40) by zero to get

$$(0 \text{ eq } 0 \text{ -> } \mid .(1/0) \qquad (2.3\text{-}41a)$$

Using the desugaring of (2.3-38) leads to

$$\begin{aligned} &Q\ (0 \text{ eq } 0)\ 0\ (1/0) \\ &\xrightarrow{\delta} Q\ \text{true}\ 0\ (1/0) \end{aligned} \qquad (2.3\text{-}41b)$$

(Here we have used the δ-rule that "0 = 0" has the same value as does "true".)

We then use the definition of $\underline{0}$ in (2.3-39) to get

$$\begin{aligned} &\xrightarrow{\delta} (\lambda \underline{x}.\ \lambda y.\ x)\ 0\ (1/0) \\ &\xrightarrow{\beta} (\lambda \underline{y}.\ 0)\ (1/0) \\ &\xrightarrow{\beta} 0 \end{aligned} \qquad (2.3\text{-}41c)$$

Note that no attempt is made in this evaluation to divide by zero.

2.4 Recursive Functions

We have shown that PAL constructs can be regarded as alternate syntactic ways of writing AE's, so that, for example,

```
let x = P in Q
```

is an alternate form for

```
(λx. Q) P
```

and

```
x+y
```

is an alternate form for

```
Add x y
```

The question of the moment, then, is this: For what AE is

```
let rec f n = P in Q
```

an alternate form? The answer to this question is the topic of the remainder of this section. It is a distinctly non-trivial question.

A recursive definition is one in which the object being defined is used as part of its definition. Every definition of a structured object which we have seen so far is recursive; for example, that of AE on page 2.3-62 is recursive since it defines the class AE and uses that class three times in the definition. The definition "works" since each use is a smaller entity than the whole, and the definition can terminate on identifier, which uses no recursion.

We have also shown several PAL programs which are recursive, such as that of "Equal" on page 2.1-24. In that program the function "Equal" is invoked in order to compute "Equal". Again, the definition "works" because each recursive invocation is on shorter strings than the previous one, so that initial application of "Equal" to finite-length strings must eventually terminate.

Formalization of Recursion

Our intuition about rec is the following: When we write a PAL program such as

```
let rec f n = P in Q                                    (2.4-1)
```

(where P and Q are AE's), we intend that any free occurrence of f in P is to be bound to the f being defined. Without the "rec", (2.4-1) would be equivalent to the AE

```
(λf. Q) (λn. P)
```

in which it is clear that any f that occurs free in P is free in the entire expression.

An example may help, and we use the factorial function. The PAL program

```
let rec f n = n eq 0 -> 1 | n * f (n-1)
in                                                    (2.4-2a)
f 3
```

consists of a definition of f to be the factorial function, and the application of f to 3. Were the "rec" not present, (2.4-2a) could be rewritten as the AE

$$(\lambda f.\ f\ 3)\ [\lambda n.\ n \text{ eq } 0 \text{ -> } 1 \mid n * f\ (n-1)] \qquad (2.4\text{-}2b)$$

in which it is clear that the appearance of f in the square brackets is free in the entire expression. Evidently "rec" is a magical operator whose effect is to cause that occurrence of f to be bound to the bracketted expression. How do we dispel the magic? To answer that question we need some mathematics.

Fixed Points: If F is a function and w is a value such that

$$w = F\ w \qquad (2.4\text{-}3)$$

then w is called a fixed point of F. If F: A → B, then clearly w ∈ A ∩ B. For example, the function S defined by

$$S = \lambda x.\ x^2 - 6 \qquad (2.4\text{-}4a)$$

nas the two fixed points 3 and -2, as can be verified by

$$\begin{aligned} &(\lambda x.\ x^2-6)\ 3 \xrightarrow{\beta} 3^2 - 6 \xrightarrow{\delta} 3 \\ &(\lambda x.\ x^2-6)\ (-2) \xrightarrow{\beta} (-2)^2 - 6 \xrightarrow{\delta} -2 \end{aligned} \qquad (2.4\text{-}4b)$$

(It is usually easier to verify a fixed point than to find one.) Here S has functionality

$$S \in N \to N$$

and the fixed points are, therefore, numbers. (Here and through the rest of this section we use "N" for numbers.)

Now consider the function T defined by

$$T = \lambda f.\ \lambda().\ (f\ nil)^2 - 6 \qquad (2.4\text{-}5a)$$

One fixed point of T is (λ(). 3), since

$$\begin{aligned} &T\ (\lambda().\ 3) \\ &= [\lambda f.\ \lambda().\ (f\ nil)^2 - 6]\ (\lambda().\ 3) \\ &\xrightarrow{\beta} \lambda().\ [(\lambda().\ 3)\ nil]^2 - 6 \\ &\xrightarrow{\beta} \lambda().\ [3]^2 - 6 \\ &\xrightarrow{\delta} \lambda().\ 3 \end{aligned} \qquad (2.4\text{-}5b)$$

The reader can easily verify that (λ(). -2) is also a fixed point of T.

The functionality of T is easy to work out. If T is applied to some ob g, we get

$$\lambda().\ (g\ nil)^2 - 6 \qquad (2.4\text{-}5c)$$

Evidently g is a function of no arguments whose codomain is numbers (to be within the domain of "-"), and equally evidently the expression in (2.4-5c) is also a function of no arguments whose codomain is numbers. Thus

$$T \in (nil \to N) \to (nil \to N) \qquad (2.4\text{-}5d)$$

and the two fixed points of I which we have seen are elements of

$$nil \to N \qquad (2.4\text{-}5e)$$

As another example, consider

$$U = \lambda f.\ \lambda n.\ n \text{ eq } 0 \to 1 \mid n * f\ (n-1) \qquad (2.4\text{-}6a)$$

Clearly

$$U \in (N \to N) \to (N \to N) \qquad (2.4\text{-}6b)$$

We now show two things: The function Square (that is, $\lambda x.\ x^2$) is not a fixed point of U, and Factorial is. Consider first "Square":

$$\begin{aligned} &U\ Square \\ &\equiv [\lambda f.\ \lambda n.\ n \text{ eq } 0 \to 1 \mid n * f\ (n-1)]\ [\lambda x.\ x^2] \\ &\xrightarrow{\beta} \lambda n.\ n \text{ eq } 0 \to 1 \mid n * (\lambda x.\ x^2)\ (n-1) \\ &\xrightarrow{\beta} \lambda n.\ n \text{ eq } 0 \to 1 \mid n * (n-1)^2 \end{aligned}$$

This last is clearly not Square. (If applied to 0 it returns 1.) Now,

$$\begin{aligned} &U\ Factorial \\ &\equiv [\lambda f.\ \lambda n.\ n \text{ eq } 0 \to 1 \mid n * f\ (n-1)]\ Factorial \\ &\xrightarrow{\beta} \lambda n.\ n \text{ eq } 0 \to 1 \mid n * Factorial\ (n-1) \end{aligned}$$

This last is the factorial function, since it is clear that its domain of definition is the non-negative integers and that its value at n is the factorial of n. Of course,

$$Factorial \in N \to N$$

As a final example, we leave it to the reader to verify that the square-root function is a fixed point of

$$\lambda f.\ \lambda x.\ x\ /\ (f\ x)$$

Two points have been made:

(1) The fixed point of a function may itself be a function.

(2) If $F \in \alpha \to \alpha$ and $w \simeq F\ w$, then $w \in \alpha$.

Desugaring of "rec": Let us go back now to the problem of getting the magic out of rec. What we want to do is capture formally our intuitive idea that "rec" is to cause any free occurrence of f in P (in (2.4-1)) to be bound to the f being defined.

Before proceeding, note that

$$(\lambda x.\ M)\ x \simeq M \qquad (2.4\text{-}7)$$

for any AE M, since β-reduction of the expression on the left leads to

$$subst\ (x,\ x,\ M)$$

which will be M, whether or not there are free occurrences of x in M.

Now let us again consider

$$\text{let rec } f\ n = P \text{ in } Q \qquad (2.4\text{-}8a)$$

We can rewrite this as

$$\text{let rec } f = \lambda n.\ P \text{ in } Q \qquad (2.4\text{-}8b)$$

as our usual desugaring of a function-form definition. Next, we take $(\lambda n.\ P)$ as $\underline{u}$ in (2.4-7) to get

$$\text{let rec } f = (\lambda f.\ \lambda n.\ P)\ f \text{ in } Q \qquad (2.4\text{-}8c)$$

(Here we have used (2.4-7) "backwards".)

Now let $\underline{F}$ be defined as

$$F \equiv \lambda f.\ \lambda n.\ P \qquad (2.4\text{-}8d)$$

With this abbreviation, then, we can rewrite (2.4-8c) as

$$\text{let rec } f = F(f) \text{ in } Q \qquad (2.4\text{-}8e)$$

This form has a rather desirable property, one that is not in evidence in (2.4-8a). In the nature of things there can be no free occurrence of $\underline{f}$ in $\underline{F}$, since any free $\underline{f}$ in $\underline{P}$ is bound by the "λf". Thus (2.4.8e) is some sort of standardized form of the general recursive definition, one in which there is on the right exactly one free occurrence of the recursive variable being defined.

Now recall the intuition that we started out to capture: that free occurrences of $\underline{f}$ on the right are to be bound to the $\underline{f}$ being defined. That is, in

$$\text{rec } f = F\ f$$

the two occurrences of $\underline{f}$ are to be the same. Another way to state this is that "$\underline{f}$" *is to be a fixed point of* "$\underline{F}$". With this in mind we can rewrite (2.4-8e) as

$$\text{let } f = \text{a\_fixed\_point\_of } F \text{ in } Q \qquad (2.4\text{-}8f)$$

where "a_fixed_point_of" is the name of a function which produces fixed points. Since we plan to study this function extensively we give it a shorter name, $\underline{Y}$, following the notation of Curry. Replacing $\underline{F}$ as in (2.4-8d) we conclude that

$$\text{let rec } f\ n = P \text{ in } Q \qquad (2.4\text{-}9a)$$

can be rewritten as

$$\text{let } f = Y\ (\lambda f.\ \lambda n.\ P) \text{ in } Q \qquad (2.4\text{-}9b)$$

and hence as

$$(\lambda f.\ Q)\ [Y\ (\lambda f.\ \lambda n.\ P)] \qquad (2.4\text{-}9c)$$

There can be no free occurrence of $\underline{f}$ in (2.4-9c).

Let us review what we have done. We started out to ask what AE (2.4-9a) could be a sugaring of, so that any free occurrence of $\underline{f}$ in $\underline{P}$ would be the same as the $\underline{f}$ being defined. Our answer seems to be the AE in (2.4-9c), an AE whose only free identifier is $\underline{Y}$. (Any other identifier free in (2.4-9c) is also free in (2.4-9a) and not of concern at the moment.) Thus we have replaced the study of recursion by a study of $\underline{Y}$. We devote the remainder of this section to that study.

The Fixed Point Operator "Y"

We have just seen that a PAL program involving "rec" can be regarded as syntactic sugar for an AE involving Y, an operator that produces fixed points. It therefore follows that if we understand Y we understand recursion. In the present section we investigate the properties of Y itself, and in the next section we consider the properties of the fixed point which it produces.

The Fundamental Identity: We have postulated a Y that produces fixed points, and that is all we need to assume about Y to deduce its important properties. For assume F is some function and that (Y F) is a fixed point of F. Now, a fixed point w of F has the property that

$$w = F\ w \tag{2.4-10}$$

by definition of what it means to be a fixed point. If (Y F) is a fixed point of F, we may substitute it for w in (2.4-10) to get

$$\boxed{Y\ F = F\ (Y\ F)} \tag{2.4-11}$$

This is the fundamental identity for "Y", and we use it to derive all the properties of Y that we need. Note carefully that we deduced it by assuming only that Y applied to F yields a fixed point of F.

The reader is advised to stop at this point to reflect on Y and on the fundamental identity. We have seen two results on recursion, and although neither is particularly deep they are quite important and often misunderstood. (The last sentence means that the authors have seen many students misunderstand them.) More to the point, much of the development in the rest of these notes requires a good guts understanding of two points. The first is the explanation of "rec" in terms of Y as in (2.4-8) and (2.4-9), and the second is the derivation of the fundamental identity. You should be able to explain either of these to a friend, without reference to these notes. Unless you are sure you can, read again from the beginning of Section 2.4. (Note the recursive instruction.)

Evaluation Using "Y": The explanation of "rec" in terms of Y may seem plausible up to this point, but what is still lacking is any evidence that, in any practical sense at all, it works. It turns out that AE's involving Y can be evaluated using only the fundamental identity (and also, of course, α and β and δ rules).

To make explicit our rules, we add now a fourth axiom to be used in addition to the previous three.

Axiom ρ: If F is any AE, then in any context

$$Y\ F \xrightarrow{\rho} F\ (Y\ F)$$

We modify our definition of normal form to require that an AE in normal form not be a combination whose rator is Y. Finally, we redefine normal order reduction to require that at each step we reduce the left-most possible Y or λ.

With these preliminaries complete, let us try to compute the factorial of three, using the PAL program

```
let rec f n = n eq 0 -> 1 | n * f(n-1) in f 3          (2.4-12a)
```

The equivalent AE is

$$(\lambda f.\ f\ 3)\ [Y\ (\lambda f.\ \lambda n.\ n\ eq\ 0 \rightarrow 1 \mid n * f(n-1))] \qquad (2.4\text{-}12b)$$

We now do normal order reduction on this AE, using our new rules. To save writing, we abbreviate

$$F \equiv \lambda f.\ \lambda n.\ n\ eq\ 0 \rightarrow 1 \mid n * f\ (n-1) \qquad (2.4\text{-}12c)$$

Then we have

$$\begin{aligned}
&(\lambda f.\ f\ 3)\ (Y\ F) && (2.4\text{-}12d)\\
&\xrightarrow{\beta} Y\ F\ 3\\
&\xrightarrow{\rho} F\ (Y\ F)\ 3\\
&\equiv [\lambda f.\ \lambda n.\ n\ eq\ 0 \rightarrow 1 \mid n * f(n-1)]\ (Y\ F)\ 3\\
&\xrightarrow{\beta} [\lambda n.\ n\ eq\ 0 \rightarrow 1 \mid n * Y\ F\ (n-1)]\ 3\\
&\xrightarrow{\beta} 3\ eq\ 0 \rightarrow 1 \mid 3 * Y\ F\ (3-1)\\
&\equiv Q\ (3\ eq\ 0)\ 1\ (3 * Y\ F\ 2)\\
&\xrightarrow{\delta} Q\ false\ 1\ (3 * Y\ F\ 2)\\
&\xrightarrow{\delta} 3 * (Y\ F\ 2)
\end{aligned}$$

In going from the second line to the third we used ρ-conversion. Line 4 involves replacing F by that which it abbreviates. The last two lines involve use of δ-rules, including selection of an arm of a conditional.

The derivation of (2.4-12d) shows that

$$Y\ F\ 3 \simeq 3 * Y\ F\ 2 \qquad (2.4\text{-}13a)$$

and it is clear that, for any integer $k>0$,

$$Y\ F\ k \simeq k * Y\ F\ (k-1) \qquad (2.4\text{-}13b)$$

Further,

$$\begin{aligned}
&Y\ F\ 0 && (2.4\text{-}13c)\\
&\equiv [\lambda n.\ n\ eq\ 0 \rightarrow \mid n * Y\ F\ (n-1)]\ 0\\
&\xrightarrow{\beta} 0\ eq\ 0 \rightarrow 1 \mid 0 * Y\ F\ (-1)\\
&\simeq 1
\end{aligned}$$

Thus for the F of (2.4-12c) we see that (Y F) is the factorial function since it has the same domain of definition as does factorial and the same values at those points.

This derivation shows us that regarding PAL programs involving "rec" as sugaring for certain AE's involving Y "works", in the sense that we get AE's that we can evaluate using our axioms.

There remain many issues to discuss. For what class of PAL programs does this method work? What is the nature of the fixed point produced by Y? Is it unique? We return to these and related questions after discussion of several λ-calculus versions of Y.

A λ-calculus "Y" : By adding ρ-conversion as a new axiom we have made Y somewhat of a special case in evaluation. A sensible question to ask is whether or not we could get the effect of Y without postulating a new ob in the universe of discourse. That is, can we find an AE that satisfies the fundamental identity? The answer is "yes"; and in fact there exists a whole family of such AE's. We use a rather roundabout way to derive one of them.

We first try to write a program for factorial without using recursion. Consider

```
let f (g, n) = n eq 0 -> 1 | n * g(g, n-1)
in
let h n = f (f, n)                                    (2.4-14)
in
h
```

The definition of f is not recursive, since f does not call itself. To see that h is the factorial function, consider h(3). Proceeding informally, we have

```
h 3  =  f (f, 3)
     =  3 eq 0 -> 1 | 3 * f (f, 2)
     =  3 * f (f, 2)
     =  3 * 2 * f (f, 1)
     ...
```

Further, h(0) is one.

The f of (2.4-14) is dyadic, but we find it more convenient to consider a curried version. Currying f requires that g be curried also, and we have

```
let f g n = n eq 0 ->1| n* g g (n-1)
in                                                    (2.4-15a)
f f
```

We assert first that the value of this AE is Factorial, leaving it to the reader to verify this assertion. (Let w be the AE. Show first that w(k) = k*w(k-1) and then that w(0)=1.)

We now proceed with a derivation to show that the AE of (2.4-15a) is equivalent to an AE of the form (Z F), where F is as in (2.4-12c). Since F encompasses all there is to be said about factorial, Z is, as it were, a "recursion maker".

We start by rewriting (2.4-15a) as an AE:

$$(\lambda f.\ f\ f)\ [\lambda g.\ \lambda n.\ n \text{ eq } 0 \text{ -> } 1 \mid n * g\ g\ (n-1)] \qquad (2.4\text{-}15b)$$

We use β-expansion on this to get the AE

$$(\lambda f.\ f\ f)\ \{\lambda g.\ [\lambda f.\ \lambda n.\ n \text{ eq } 0 \text{ -> } 1 \mid n*f\ (n-1)]\ (g\ g)\} \qquad (2.4\text{-}15c)$$

That this is in fact equivalent may be most easily seen by using β-reduction on (2.4-15c) with the rator in square brackets and (g g) as the rand. The result is (2.4-15b). Note that the expression in square brackets in (2.4-15c) is precisely the F defined in (2.4-12c). Using F as an abbreviation, (2.4-15c) is the same as

$$(\lambda f.\ f\ f)\ [\lambda g.\ F\ (g\ g)] \qquad (2.4\text{-}15d)$$

and we do one final β-expansion to get

$$[\lambda F.\ (\lambda f.\ f\ f)\ (\lambda g.\ F\ (g\ g))]\ F \qquad (2.4\text{-}15e)$$

Letting Z be the abbreviation

$$Z \equiv \lambda F.\ (\lambda f.\ f\ f)\ (\lambda g.\ F\ (g\ g)) \qquad (2.4\text{-}16)$$

we have shown that (2.4-15a) is equivalent to (Z F), as promised.

Note what happened: We started out with an AE which does not involve recursion but which nonetheless denotes factorial. The AE has two aspects to it: A part that defines factorial and a part that does (in some sense) the recursion. The derivation served to separate these two parts: We know that F relates to factorial, and we want to study Z. It is clearly related to Y, since both (Z F) and (Y F) have been shown to denote factorial.

We first show that Z satisfies the fundamental identity. For any G we have

$$\begin{aligned} & Z\ G && (2.4\text{-}17a)\\ &\equiv [\lambda F.\ (\lambda f.\ f\ f)\ (\lambda g.\ F\ (g\ g))]\ G\\ &\xrightarrow{\beta} (\lambda f.\ f\ f)\ (\lambda g.\ G\ (g\ g))\\ &\xrightarrow{\beta} [\lambda g.\ G\ (g\ g)]\ [\lambda g.\ G\ (g\ g)] \end{aligned}$$

This last is of course equivalent to (Z G). One more β-reduction gives

$$G\ \{[\lambda g.\ G\ (g\ g)]\ [\lambda g.\ G\ (g\ g)]\} \qquad (2.4\text{-}17b)$$

in which the expression in braces is the same AE as the last line of (2.4-17a), so

$$Z\ G \simeq G\ (Z\ G) \qquad (2.4\text{-}17c)$$

We have a λ-expression that satisfies the fundamental identity. Although we have shown that (Z F) denotes factorial, the reader is advised to replace Z and F by the λ-expressions they abbreviate ((2.4-16) and (2.4-12c), respectively) and to evaluate (Z F 2) using only β-reduction and δ-rules. The result is instructive (though tedious).

A Family of Functions: It turns out that there is a whole class of Z's that satisfy the fundamental identity. More remarkable, each of them is itself a fixed point of the function

$$H \equiv \lambda y.\ \lambda f.\ f\ (y\ f) \qquad (2.4.18)$$

Let

$$\begin{aligned} Z_0 &\equiv Z \equiv \lambda F.\ (\lambda f.\ f\ f)\ (\lambda g.\ F\ (g\ g))\\ Z_{K+1} &\equiv Z_K\ H \end{aligned} \qquad (2.4\text{-}19)$$

Consider Z_1. We have

$$
\begin{aligned}
& Z_1 \\
& \equiv Z_0\ H \\
& = [\underline{\lambda F}.\ (\lambda f.\ f\ f)\ (\lambda g.\ F\ (g\ g))]\ (\lambda y.\ \lambda f.\ f\ (y\ f)) \\
& \xrightarrow{\beta} (\lambda f.\ f\ f)\ [\lambda g.\ (\underline{\lambda y}.\ \lambda f.\ f\ (y\ f))\ (g\ g)] \\
& \xrightarrow{\beta} (\lambda f.\ f\ f)\ [\lambda g.\ \lambda f.\ f\ (g\ g\ f)]
\end{aligned}
\tag{2.4-20}
$$

We show first that Z_1 satisfies the fundamental identity. First define the abbreviation

$$G \equiv \lambda g.\ \lambda f.\ f\ (g\ g\ f) \tag{2.4-21a}$$

Then

$$
\begin{aligned}
& Z_1\ F \\
& = (\lambda f.\ f\ f)\ (\lambda g.\ \lambda f.\ f\ (g\ g\ f))\ F \\
& \equiv (\underline{\lambda f}.\ f\ f)\ G\ F \\
& \xrightarrow{\beta} G\ G\ F \\
& \equiv [\underline{\lambda g}.\ \lambda f.\ f\ (g\ g\ f)]\ G\ F \\
& \xrightarrow{\beta} [\underline{\lambda f}.\ f\ (G\ G\ f)]\ F \\
& \xrightarrow{\beta} F\ (G\ G\ F) \\
& \simeq F\ (Z_1\ F)
\end{aligned}
\tag{2.4-21b}
$$

since (G G F) appears on the fourth line of the derivation. It is not hard to see that $\underline{Z}_0$ is a fixed point of $\underline{H}$. We must show that $H\ Z_0 \simeq Z$, and we have

$$
\begin{aligned}
& H\ Z_0 \\
& \equiv [\underline{\lambda y}.\ \lambda f.\ f(y\ f)]\ Z_0 \\
& \xrightarrow{\beta} \lambda f.\ f(Z_0\ f) \\
& \equiv \lambda f.\ f\{[\underline{\lambda F}.(\lambda f.f\ f)\ (\lambda g.F(g\ g))]\ f\} \\
& \xrightarrow{\beta} \lambda f.\ f[(\underline{\lambda f}.f\ f)\ (\lambda g.f(g\ g))] \\
& \xrightarrow{\beta} \lambda f.\ f[(\lambda g.f(g\ g))\ (\lambda g.f(g\ g))] \\
& \xleftarrow{\beta} \lambda f.\ (\lambda g.f(g\ g))\ (\lambda g.f(g\ g)) \\
& \xleftarrow{\beta} \lambda f.\ (\lambda f.f\ f)\ (\lambda g.f(g\ g)) \\
& \xrightarrow{\alpha} Z_0
\end{aligned}
$$

Derivations such as this one that use β-expansion have a certain mysterious air about them. Although it is easy for the reader to verify that the expansion is correct (by doing the corresponding reduction), it is probably not at all obvious why this path was chosen -- other than the fact that it works.

We leave it as an exercise to the reader to show that other $\underline{Z}_k$ are fixed points of $\underline{H}$ and that they satisfy the fundamental identity. These $\underline{Z}_k$ are due to Bohm (1966) and are discussed in Morris (1968) who proves (page 71) these results for all $\underline{k}$. It is apparently true (although it has yet to be proved) that no two Z_k are interconvertible by λ-reduction. That is, it is not true that

$$Z_0 \simeq Z_1$$

Since they both do the same thing, this is a rather remarkable result.

The Minimal Fixed Point

We have seen that our decision to desugar "rec" in terms of Y works for factorial, leading to an AE that can be evaluated using either ρ-reduction or (by replacing Y by a suitable λ-expression) by β-reduction. What is not obvious is how to characterize the class of λ-expressions F for which (Y F) produces a useful fixed point.

That every λ-expression F has at least one fixed point -- namely (Y F) -- follows from the definition of Y. Thus the problem is the nature of the fixed point, not its existence. Specifically, does (Y F) act like the function we want?

We saw that it does for factorial. Let us look at some other examples.

Some Examples: We first consider the case in which the "rec" is, in a sense, superfluous. Consider

```
let rec f n = P in Q                                    (2.4-22)
```

in which there is no free occurrence of f in P. This desugars to

```
(λf. Q) [Y (λf. λn. P)]
→ρ (λf. Q) (λf. n. P) [Y (λf. λn. P)]
→β (λf. Q) {subst [Y (λf. λn. P), f, (λn. P)]}
```

Since we have postulated no free occurrences of f in P, no substitution in fact takes place, and we get

```
(λf. Q) (λn. P)
```

which can be sugared to

```
let f n = P in Q
```

Thus we have confirmed what seems plausible: Adding an unneeded "rec" has no semantic effect. (It may adversely affect efficiency.)

As another example, recall that the square-root function is a fixed point of

```
λf. λx. x / (f x)                                       (2.4-23a)
```

This suggests that we can define s in PAL to be the square root function by

```
let rec s x = x / (s x) in ...                          (2.4-23b)
```

Desugaring leads to

$$
\begin{aligned}
&Y\ [\lambda s.\ \lambda x.\ x\ /\ (s\ x)] = Y\ S \\
&\xrightarrow{\beta} S\ (Y\ S) \\
&= \lambda x.\ x\ /\ (Y\ S\ x) \\
&= \lambda x.\ x\ /\ [\underline{\lambda x}.\ x\ /\ (Y\ S\ x)]\ x \\
&\xrightarrow{\beta} \lambda x.\ x\ /\ (x\ /\ Y\ S\ x) \\
&\xrightarrow{\eta} \lambda x.\ Y\ S\ x
\end{aligned}
$$

and this never terminates. Thus $\underline{Y}$ does not produce a useful fixed point. It would have been surprising in some sense had it done so, since there is an air of getting something for nothing in using (2.4-23b) as a definition of square root.

As another example in which we should expect failure of $\underline{Y}$ to produce a fixed point consider

$$\text{let rec } x\ () = (x\ \text{nil})^2 - 6 \text{ in } \ldots \qquad (2.4\text{-}24a)$$

in which we are concerned with

$$F \equiv \lambda x.\ \lambda().\ (x\ \text{nil})^2 - 6 \qquad (2.4\text{-}24b)$$

This is the same function that we considered in (2.4-5a), and we know its fixed points are $(\lambda().\ 3)$ and $(\lambda().\ -2)$. It would be rather surprising were $(Y\ F)$ to select either of these two fixed points (how to choose?), so let us see. We have

$$
\begin{aligned}
&Y\ F \\
&\xrightarrow{\beta} F\ (Y\ F) \\
&= [\underline{\lambda x}.\ \lambda().\ (x\ \text{nil})^2 - 6]\ (Y\ F) \\
&\xrightarrow{\beta} \lambda().\ (Y\ F\ \text{nil})^2 - 6
\end{aligned}
$$

Since this is $(Y\ F)$, it follows that

$$
\begin{aligned}
&Y\ F\ \text{nil} \\
&\simeq (Y\ F\ \text{nil})^2 - 6 \\
&\simeq ((Y\ F\ \text{nil})^2 - 6)^2 - 6 \\
&\simeq [((Y\ F\ \text{nil})^2 - 6)^2 - 6]^2 - 6
\end{aligned}
$$

which clearly does not terminate. Further, there can exist no $\underline{x}$ such that $(Y\ F\ x)$ terminates.

Let us investigate still another $\underline{F}$:

$$F = \lambda x.\ \lambda y.\ x \qquad (2.4\text{-}25)$$

we have

$$
\begin{aligned}
& Y\ F \\
\beta\rightarrow\ & F\ (Y\ F) \\
\equiv\ & (\underline{\lambda x}.\ \lambda y.\ x)\ (Y\ F) \\
\beta\rightarrow\ & \lambda y.\ Y\ F \\
\simeq\ & \lambda y.\ \lambda y.\ Y\ F \\
\alpha\ & \lambda y.\ \lambda y.\ \lambda y.\ Y\ F
\end{aligned}
$$

Not only does this (Y F) not admit a normal form, but it is a "universal annihilator" in the sense that

$$Y\ F\ V1\ V2\ \ldots\ Vn \simeq Y\ F$$

for all AE's Vk.

Now we consider a case in which (Y F) has no normal form but for which fixed points exist. If

$$F \equiv \lambda x.\ x\ x\ x \qquad (2.4\text{-}26)$$

we have

$$
\begin{aligned}
& Y\ F \\
\beta\rightarrow\ & F\ (Y\ F) \\
\equiv\ & (\underline{\lambda x}.\ x\ x\ x)\ (Y\ F) \\
\beta\rightarrow\ & (Y\ F)\ (Y\ F)\ (Y\ F)
\end{aligned}
$$

which clearly has no normal form. But it is easy to verify that both of

$$
\begin{aligned}
true\ &\equiv\ \lambda x.\ \lambda y.\ x \\
false\ &\equiv\ \lambda x.\ \lambda y.\ y
\end{aligned}
\qquad (2.4\text{-}27)
$$

are fixed points of this $\underline{F}$, and are in normal form. In this case as in (2.4-24) $\underline{Y}$ fails to produce a fixed point, although two exist.

We now consider a more involved example, one in which (Y F) has no normal form but nonetheless acts like a function, in the sense that we show an $\underline{x}$ such that (Y F x) has normal form. Let

$$F \equiv \lambda x.\ \lambda y.\ y\ y\ (x\ false) \qquad (2.4\text{-}28)$$

(Hereafter in this section "true" and "false" are as in (2.4-27).) Then

$$
\begin{aligned}
& Y\ F \\
\beta\rightarrow\ & F\ (Y\ F) \\
\equiv\ & (\underline{\lambda x}.\ \lambda y.\ y\ y\ (x\ false))\ (Y\ F) \\
\beta\rightarrow\ & \lambda y.\ y\ y\ (Y\ F\ false)
\end{aligned}
$$

Then

$$
\begin{aligned}
& Y\ F\ false \\
\simeq\ & (\underline{\lambda y}.\ y\ y\ (Y\ F\ false))\ false \\
\beta\rightarrow\ & false\ false\ (Y\ F\ false) \\
\simeq\ & Y\ F\ false
\end{aligned}
$$

Thus (Y F) has no normal form, and we also see that (Y F false) has no normal form either. Now,

$$
\begin{aligned}
&Y\ F\ true \\
&\simeq\ (\lambda y.\ y\ y\ (Y\ F\ false))\ true \\
&\beta\ true\ true\ (Y\ F\ false) \\
&\simeq\ true
\end{aligned}
$$

Thus we see that (Y F) acts like a function, a function whose domain of definition inclues "true" but not "false". It is easy to see that

$$G \equiv \lambda z.\ z\ z\ false$$

is also a fixed point of F, and that

$$
\begin{aligned}
G\ true &\simeq true \\
G\ false &\simeq false
\end{aligned}
$$

Thus we have two fixed points: (Y F) and G. At a point at which they are both defined (the point "true") they agree, but G is defined for at least one point at which (Y F) is not defined. Thus we have some evidence (but no proof) that G is a functional extension of (Y F). (Reread now the definition of functional extension on page 2.1-19. We see more of it in this section.)

For our final example we consider again

$$F \equiv \lambda f.\ \lambda x.\ x\ eq\ 0\ \rightarrow\ 1\ |\ x * f\ (x-1) \qquad (2.4\text{-}29)$$

we have seen that (Y F) is the factorial function. Has F any other fixed points? The answer is yes. Consider the function g defined (mathematically) by

$$
g\ (x) = \begin{cases} 1 & x = 0 \\ x*g\ (x-1) & x = 1,\ 2,\ 3,\ \ldots \\ k & x = 1/2 \\ x*g\ (x-1) & x = 3/2,\ 5/2,\ 7/2,\ \ldots \\ g\ (x+1)\ /\ (x+1) & x = -1/2,\ -3/2,\ -5/2,\ \ldots \end{cases}
$$

Here k is any number at all. It is easy to verify that g is a fixed point of f by letting h be (F g) and showing that h and g are the same function. We have

$$h \equiv F\ g \simeq \lambda x.\ x\ eq\ 0\ \rightarrow\ 1\ |\ x * g\ (x-1)$$

and thus

```
h(0)   = 1
h(n)   = n*g (n-1)          n = 1, 2, ...
h(0.5) = 0.5 * g (-0.5)
       = 0.5 * (g (0.5) / 0.5)
       = g (0.5)
h(x)   = x*g (x-1)          x = 3/2, 5/2, ...
h(x)   = x*g (x-1)          x = -1/2, -3/2, ...
       = x * (g (x) /x))
       = g (x)
```

Further, h is defined at no other points. Thus h = g so

$$g \simeq F\ g$$

and g is a fixed point of F.

We have two fixed points: (Y F) and g. What is the relation between them? It is clear that g is a functional extension of (Y F), in that it is defined whenever (Y F) is and has the same values at those points. Further, any function like g but with a different value at 0.5 (i.e., a different value of k) would also be a fixed point of F and a functional extension of (Y F). Finally, we could add, say, 0.4 to the domain of g with any value at all and define g with the usual recursion at (1.4, 2.4, ...) and (-0.6, -1.6, ...) to get still another fixed point of F which is an extension of (Y F).

Characterization of (Y F): The preceding discussion serves to motivate the following theorem, which we state without proof. It is due to Morris (1968).

> Theorem: If w is any fixed point of F, then w is a functional extension of (Y F).

(Morris' proof was for the λ-calculus Y of (2.4-16). The theorem has not been proved for the Y we are using.)

The theorem says two things: First, of all possible fixed points of F, (Y F) is the one with "smallest" domain. (Its domain is contained in the domain of any other fixed point.) And second, all fixed points agree with (Y F) at points at which (Y F) is defined. Thus we say that (Y F) is the minimal fixed point. The last example brought this out clearly.

The fact that (Y F) is minimal is distressing in the sense that (Y F) represents the weakest possible response to the question,"What is the fixed point of F?" On the other hand, in a more important sense this fact is reassuring. Recall for example that both "true" and "false" are fixed points of the AE in (2.4-26). We should look askance on any scheme that arbitrarily returned one or the other when both are equally appropriate. Only the minimal fixed point of an AE is uniquely specified solely by the AE itself. To expect Y to produce any other fixed point would indeed be magical.

Continuing with consideration of (2.4-26), the reader might wonder how our results in connection with that AE are consistent with Morris' theorem. Recall that every function is a functional extension of ϕ, the empty function. Since the (Y F) of (2.4-26) is nowhere defined (in the sense that there is no x such that (Y F x) terminates), (Y F) is indistinguishable from ϕ. Both "true" and "false" are perfectly good extensions of ϕ.

Limitations of the λ-calculus Approach

Our entire approach to explication of PAL's semantics is based on the λ-calculus. Other approaches to the study of programming linguistics could be taken, and indeed have been taken by others. Our claim about our approach is not that it is the best approach, but only that it is an adequate approach to address the problems which we feel are important. We claim that the material of these notes, read and understood, provides the professional in computer science with the tools he needs for the study of programming languages.

One shortcoming of the λ-calculus approach should be pointed out. A fundamental idea in programming is the recursive function, a function that calls itself in the course of its operation. As we have seen, regarding PAL as sugaring for λ-expressions leads to the problem of deciding just what sort of λ-expression, when sugared, results in a PAL program with a rec. This is an interesting problem, and the mathematics needed to answer it is distinctly non-trivial.

The objection voiced by some is that recursion is a syntactic problem, having to do with that part of the program text wherein a particular name is known, and that it should not be necessary to develop complicated mathematics to explain it. We answer that objection on two different levels: In the first place, we do not feel that recursion is quite that simple. Although it is possible to explain recursion syntactically as just suggested, such explanations that we have seen seem to lack the mathematical rigour we feel appropriate for this study. Our objective is to develop techniques for formalizing the semantics of computer languages, and facing squarely the hard problems of recursion seems appropriate. The problems don't go away through being ignored.

This leads us to the second answer to the objection. There is no denying that the mathematics we have presented in this section is harder than that in the rest of this text. But one should not avoid problems just because they are hard. The mathematician can claim (correctly) that the mathematics is beautiful, and worth studying just for its own sake. While the authors of these notes happen to agree, we do not base our argument on that belief. Instead, we claim that the material is relevant to an understanding of programming linguistics as we see it, and further claim that this view of programming linguistics is an appropriate one for the computer scientist.

Chapter 3

Evaluation of Applicative Expressions

In Chapter 2 we provided a mathematical formalization of the semantics of R-PAL by showing first how any R-PAL program may be replaced by an AE with equivalent semantics and second how to evaluate AE's using the axioms of the λ-calculus. As we know, R-PAL is a subset of the entire PAL language, and our job next is to formalize L-PAL (a larger subset) and J-PAL (the entire language). It is unfortunately the case that the technique we have been using so far lends itself poorly to L-PAL and not at all to J-PAL. Let us consider briefly use of it for L-PAL.

In L-PAL, as in Algol, Fortran, PL/I and other languages, a statement such as

```
x := x + 1                                        (3.0-1a)
```

means to add 1 to x and to store the result back into x. (In Fortran and PL/I the ":=" would be replaced by "=".) Suppose we try to formalize this idea as we have been doing. We first imbed (3.0-1a) in a complete program so that x does not occur free:

```
let x = 3
in                                                (3.0-1b)
x := x + 1; x
```

This correct L-PAL program has the value 4, as one would expect. It can be desugared into the AE

```
(λx. x := x + 1; x) 3                             (3.0-1c)
```

but the next step -- β-reduction -- leads to

```
3 := 3 + 1; 3                                     (3.0-1d)
```

which is clearly meaningless. We see in Chapter 4 that the AE in (3.0-1c) is correct, but that β-reduction is in general <u>not</u> valid in AE's involving assignment.

The conclusion we should reach is that the λ-calculus as we have presented it is not an appropriate vehicle for explication of the rest of PAL. We clearly have two alternatives: to extend the λ-calculus or to develop a different technique for language formalization. We choose the latter, for two reasons:

- Although it is possible to extend the λ-calculus to accommodate L-PAL in a reasonably natural manner, it has not proved possible to extend it to J-PAL in any appealing way. Such extension has been done by Landin (1966b), but the handling of jumps is quite forced and unnatural.

- The techniques we develop in our alternate approach are related to many important ideas in computer language processing, ideas which are an important part of the study of programming linguistics.

We therefore reject the possibility of extending the λ-calculus and adopt another approach to formalization.

The approach we adopt is to explain PAL by exhibiting an algorithm for evaluating PAL programs. Assume for the moment that we are concerned with the value of every legal PAL program. (This assumption is not completely valid, since we are concerned in J-PAL with the output printed as a result of running the program. It is nonetheless a useful assumption for the moment. This point is returned to in Chapter 5.) Then our objective is a function whose domain is correct PAL programs and whose range is Ω. To the extent that one understands such a function, then, one understands PAL. We call the mechanism which this function represents the gedanken evaluator, and we give the function the name Evaluate.

Clearly, the gedanken evaluator is quite complex, and its explication is a non-trivial task. One way to do it is to define Evaluate using the usual mathematical techniques, as suggested by Strachey (1966). This approach, although attractive, is not followed in these notes. We take instead the following approach. The function Evaluate defines a transformation (from PAL programs to Ω) and is, therefore, an algorithm. (Reread the three definitions in the first paragraph of Chapter 2.) Thus we can explicate Evaluate by selecting an appropriate language and writing in it a program which is a representation of that algorithm. Thus writing this program is our objective.

The first question to ask is, "In what language shall we write the program?" This is an important question, since our understanding of the program can be no better than our understanding of the language in which it is expressed. (It is avoiding this problem that makes the mathematical definition mentioned earlier so attractive.) Let us consider the implications of using PAL as the language in which to write Evaluate. We can clearly do it and the result is a long PAL program whose understanding is needed to understand PAL. We seem to be caught in a circular trap, though, since we must understand PAL in order to understand the program, but we must understand the program in order to understand PAL. How to break the circularity?

Fortunately we have a starting point: We already understand R-PAL, so we can write Evaluate in R-PAL. We can then explain Evaluate by appeal to the techniques of Chapter 2, and Evaluate explains all of PAL. This is in fact precisely what we do, although there are additional complications.

The function Evaluate is quite complex, and a major task before us in the next three chapters is to explain it. It proves expedient to explain it in stages. Thus in the present chapter we explicate a version of Evaluate that works only for R-PAL programs, the next two chapters handling L-PAL and J-PAL. We derive an interesting advantage from this three step process: By the time we explain J-PAL we have completed the formalization of L-PAL, and so are able to

write the J-PAL version of Evaluate in L-PAL. Although it could be written in R-PAL, there are aspects of the algorithm which are expressed much more conveniently in L-PAL.

3.1 Primitives and Combinations of Primitives

In Chapter 2 we defined a class of objects called AE's and developed rules for their evaluation. We developed first (in Section 2.1) the properties of the universe of discourse and then (in Section 2.2) we considered AE's made up from the primitive identifiers and functional application. Not until section 2.3 did we introduce λ-expressions. In our present treatment of the evaluation of AE's, we similarly start with pure combination, treating AE's with λ-expressions in Section 3.2.

As we already know, the basic building blocks of AE's -- that is, the atomic elements without semantic substructure -- are identifiers. Each identifier provides direct reference to an ob in the universe of discourse called its value, and a mapping of identifiers into obs is called an environment. PAL's primitive environment (i.e. the mapping of primitive identifiers into primitive obs provided ab initio by PAL's designers) has been diagrammed in skeletal form in Figure 2.2-2, page 2.2-38.

We study extension of the primitive environment via user-coined definitions in considerable detail in Section 3.2. For the moment, however, we need only stipulate that evaluation of an identifier relative to an environment involves looking up that identifier in the environment and returning the associated value. Here we exploit the crucial property of identifiers, namely that given any two of them we can determine whether or not they are the same. Otherwise, the look-up operation could not be effected.

As a practical matter, in any PAL implementation all numerics, literals, functors and quotations are converted directly on input to the computer into an internal representation of their value, so that look-up in a table in actually necessary only in the case of variables. (The internal representation of a functor is a piece of machine code that does the work.) Conceptual (as opposed to practical) simplicity is obtained, however, by viewing all identifiers as evaluated in the same way.

Pushdown Evaluation of Polish Expressions

Assuming ability to determine the value of any identifier, we want to organize a mechanical bookkeeping procedure for evaluation of combinations. We have seen one possible procedure, that shown in Figure 2.2-7, but we want one in which order of evaluation is explicit. Consider, for example, the AE

a * b + c (3.1-1)

(We assume for the moment an environment in which the identifiers a, b and c are known.) Evaluation of this AE involves

- noticing that the multiplication is to be done before the addition;
- evaluating each of a and b;
- multiplying their values;
- evaluating c; and
- adding the value of c to the product .

All of this ordering is implicit in the expression. Clearly desirable would be a way to rewrite (3.1-1) so that the order of evaluation is explicit.

A writing such as (3.1.1) in which each binary operator is between its operands is said to be in infix form. An equivalent writing such as

```
+ * a b c                                    (3.1-2a)
```

in which each operator precedes its operands is called prefix form, and

```
a b * c +                                    (3.1-2b)
```

is called postfix form. Now note that the order of the appearance of identifiers in (3.1-2b) is exactly the same as the order of evaluation given just below (3.1-1). That is, a left-to-right scan of postfix gives the order of evaluation explicitly. For various reasons we find it convenient to scan from right to left rather than from left to right, so we use prefix form exclusively hereafter for evaluations.

Prefix and postfix forms were introduced by Łukasiewicz and are called Polish form after him. We do not devote any effort to the study of the transformation from infix to prefix, assuming that the reader can work out the details himself. For example, the arithmetic expression

```
a*3 - (-b)/(4+c)                             (3.1-3a)
```

has prefix form

```
- * a 3 / neg b + 4 c                        (3.1-3b)
```

and postfix form

```
a 3 * b neg 4 c + / -                        (3.1-3c)
```

Figure 3.1-1 shows a tree form of (3.1-3a) and shows how the prefix form may be obtained by means of a suitable walk over the tree in which each node is written down the first time it is visited.

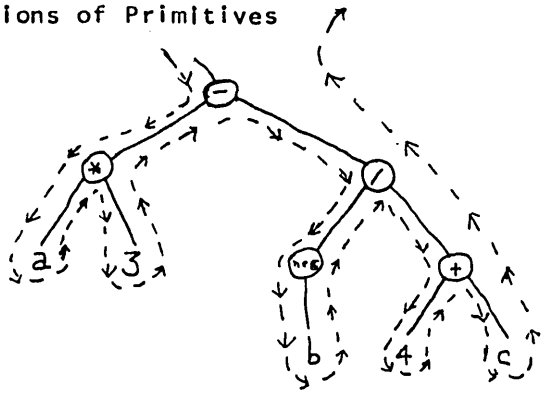


Figure 3.1-1: Tree Flattening to Prefix Form

The success of Polish form requires that it be possible to tell by looking at each operator how many operands it has. Thus unary minus is written as neg while binary minus as "-".

A push-down list, or stack, is a last-in-first out device often likened to a plate dispenser at a cafeteria. A new item inserted into the device becomes the 1st item, the old 1st item becomes the 2nd, the old 2nd item becomes the 3rd, and so forth. When the device is "popped", the transformation is reversed: The 1st item (also called the top item) is removed from the device, the old 2nd item becomes the top, the old 3rd item becomes the 2nd, etc. We assume that any push-down list is arbitrarily long in the sense that there is always room for one more item.

The use of two push-down lists (called the control and the stack) to evaluate (3.1-3b) is illustrated in Figure 3.1-2. The top of each device is adjacent to the vertical line which separates them. Initially, the prefix string is loaded into the control and the stack is empty, as shown on line 1 of the figure. An environment giving values to a, b, and c is shown on the right. Evaluation then proceeds in accordance with the following rules:

(1) Whenever an identifier appears as the top item of the control, it is evaluated and its value is pushed onto the stack. The identifier is popped from the control.

(2) Whenever an n-ary functor is the top item of the control, it is applied to the top n stack items. The functor and its n arguments are popped, and the value of the application is pushed onto the stack.

Thus the state (i.e. the configuration) of the system after the first two control items have been evaluated is as shown in line 3 of the figure. The state after the first invocation of rule (2) is exhibited on line 4 of the figure. As illustrated on succeeding lines, the evaluation cycle is iterated until the control is empty, at which time the final result occupies the top of the stack.

Economy of penmanship may be obtained by displaying successive states as shown in Figure 3.1-3. Here the control is left-justified, so that its top is its right-most item. Similarly, the stack is right-justified so that its top is its left-most item. On each line the top of each push-down list is always

	Control	Stack	Environment
1	− * a 3 \ neg b + 4 c	⊢	a = 2
2	− * a 3 \ neg b + 4	3	b = 21
3	− * a 3 \ neg b +	4 3	c = 3
4	− * a 3 \ neg b	7	
5	− * a 3 \ neg	21 7	
6	− * a 3 \	-21 7	
7	− * a 3	-3	
8	− * a	3 -3	
9	− *	2 3 -3	
10	−	6 -3	
11	⊢	9	

Figure 3.1-2: Evaluation of (3.1-3b)

	Control	Stack	Environment
1	− * a 3 / neg b + 4 c	⊢	a = 2
2	+ 4	3	b = 21
3	b +	4 3	c = 3
4	neg b	7	
5	/ neg	21	
6	3 /	-21 7	
7	a 3	-3	
8	* a	3 -3	
9	− *	2 3	
10	−	6 -3	
11	⊢	9	

Figure 3.1-3: Alternate form of Figure 3.1-2

written explicitly, but the deeper items (which of course remain unchanged from the preceding line) need not be rewritten.

Blackboard Evaluation

Recall that in section 2.2 we considered first trees like that in Figure 3.1-1 whose non-terminal nodes are operators such as "+" or "*", and then found it useful to restrict our attention to trees all of whose non-terminal nodes are γ. The reasoning that lead to this simplification continues to apply, and we consider here evaluation of such forms. The semantic tree equivalent to (3.1-3a) is

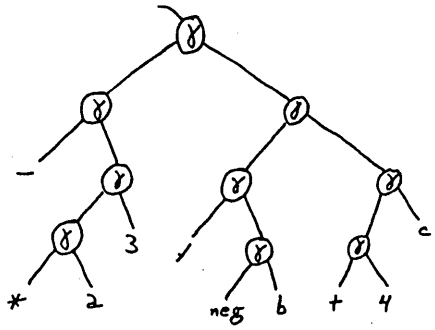


Figure 3.1-4: Semantic tree for (3.1-3a)

and a prefix walk over this tree yields

γ γ − γ γ * a 3 γ γ / γ neg 3 γ γ + 4 c (3.1-4)

Evaluation of this control sequence is shown in a display in Figure 3.1-5. Here

1	γ γ − γ γ * a 3 γ γ / γ neg b γ γ + 4 c	⊢	a = 2
2	γ γ	+ 4 3	b = 21
3	b γ	4+ 3	c = 3
4	γ neg b	7	
5	γ / γ	neg 21 7	
6	γ γ	/ -21 7	
7	γ * a 3 γ	-21 7	
8	γ γ − γ γ	* 2 3 -3	
9	γ γ	− 6 -3	
10	⊢	9	

Figure 3.1-5: Evaluation of (3.1-4)

the rule is simpler than that stated earlier:

(1) If the top control item is an identifier, pop it and push its value onto the stack.

(2) If the top control item is γ, pop it and pop the first two stack items, and push onto the stack the value obtained by applying the old top stack item to the old second stack item.

Several conventions have been used in Figure 3.1-5:

- The evaluation of the identifier "+" leads to the ob +, assumed to be a curried addition function like "Sum" in (2.2-6).
- Application of + to 4 yields a function which adds four. We write it as 4+. Similarly, -21/ represents a function which, applied to an integer n, returns -21/n.
- We leave out many uninteresting lines of evaluation. For example, two lines are left out between lines 1 and 2, so that all of the identifier evaluations seem to take place "at once". Similarly, between lines 5 and 6 neg is applied to 21 and the identifier "/" is evaluated. After line 8 two γ's are applied and "-" is evaluated, and line 9 shows application of two γ's.

A display such as Figure 3.1-5 is called a blackboard evaluation, since it can be done in class on a blackboard. We give much attention in the rest of these notes to developing conventions for the blackboard evaluation of PAL programs, culminating in Chapter 5 with a J-PAL evaluator. The reader should keep clearly in mind our objective in doing so: We wish to exhibit a mechanism that evaluates PAL programs, and the blackboard evaluator serves a a bookkeeping technique for simulating operation of that mechanism. Given that point of view, it seems appropriate to adopt any conventions that simplify use of the blackboard machine, since such simplifications serve to make more transparent the operation of the gedanken evaluator.

As an example of such simplification, compare (3.1-3b) with (3.1-4). The latter is obtained from the former by preceding every binary functor by "γ γ" and every unary functor ("neg") by "γ". Thus we use hereafter in blackboard evaluation forms such as (3.1-3b), but the reader should understand that, for example, "+" is an abbreviation for "γ γ +". The distinction becomes important in Section 3.5 in which we exhibit the PAL program for the gedanken evaluator, for then we use the "γ γ +" form.

Tuples in Blackboard Evaluation: In Section 2.2 we observed that while only γ nodes are needed for semantic trees, use also of τ nodes makes for trees which are simpler and, hence, more perspicuous. A similar argument applies to blackboard evaluation. For example, the PAL 2-tuple

a, b (3.1-5a)

could be represented by the control

γ γ aug γ γ aug nil a b (3.1-5b)

or, using "aug" as an abbreviation for "γ γ aug" as done above for "+", by

aug aug nil a b (3.1-5c)

Instead, however, we choose to use

$$\mathfrak{I}_2\ a\ b \qquad (3.1\text{-}6)$$

We can think of $\mathfrak{I}_2$ as an abbreviation for the control sequence

$$\gamma\ \gamma\ \text{aug}\ \gamma\ \gamma\ \text{aug nil} \qquad (3.1\text{-}7)$$

and similarly for a whole set of $\mathfrak{I}_k$. For example, the PAL expression

$$\text{Conc ('ab', 'cd')} \qquad (3.1\text{-}8a)$$

is equivalent to the control sequence

$$\delta\ \text{Conc}\ \mathfrak{I}_2\ \text{'ab'}\ \text{'cd'} \qquad (3.1\text{-}8b)$$

The blackboard evaluation is then

Control	Stack	Environment
γ Conc $\mathfrak{I}_1$ 'ab' 'cd' ⊢		PE
Conc $\mathfrak{I}_2$	'ab' 'cd'	
γ Conc	'ab', 'cd'	
γ	Conc 'ab', 'cd'	
⊢	'abcd'	

Figure 3.1-6: Evaluation of (3.1-8)

Here we write 'ab' to indicate a particular string, and 'ab', 'cd' to indicate a particular 2-tuple.

It is interesting that the operators $\mathfrak{I}_K$ denote functions which are already in the universe of discourse. Consider the function Tuple defined by the following PAL program:

```
def Tuple n =
    let rec Q k T =
       k eq 0 -> T | ( λx. Q (k-1)(T aug x))
    in
    Q n nil
```
(3.1-9)

Then for every n = 0, 1, 2, ..., it is true that (Tuple n) transforms precisely as does $\mathfrak{I}_n$. This can be seen by normal order reduction, and we can show it by blackboard evaluation in Section 3.4 after we have learned to accommodate recursions in the blackboard machine.

3.2 λ-Expressions

We have observed previously, on page 2.2-73, that the referential transparency of a semantic tree comprised only of γ-nodes implies invariance to order of evaluation. In particular, the ob denoted by any such tree may be determined by

(1) evaluating first its rator and rand (in either order)
(2) and then applying the one to the other.

since definitions were not involved, considerable flexibility was therefore available when deciding in Section 3.1 how to collapse a semantic tree into a control. Indeed, under these conditions the blackboard decision always to evaluate a rand before its rator is purely administrative; referential transparency guarantees that the order of evaluation could be changed -- say to rator always first and then rand, or (if parallel processors were available) to both together -- without altering semantics.

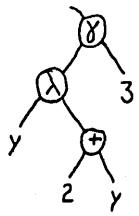
The definitional facilities of a language, however, can not be referentially transparent, so that greater care presumably is required in producing a control from semantic trees involving λ-nodes. Consider, for example, the PAL expression

```
let y = 3 in 2 + y                    (3.2-1a)
```

and hence the AE

```
(λy. 2+y) 3                           (3.2-1b)
```

The semantic tree representation of this AE is



(3.2-1c)

and a prefix walk over this tree (as in Figure 3.1-1) leads to the control sequence

```
γ γ y + 2 y 3                         (3.2-1d)
```

Evaluation of this control clearly aborts as soon as a value of y is needed, since y is not in the PE. Needed is a way to construct a control sequence for (3.2-1b) which leads, under blackbaord evaluation, to semantics equivalent to that of normal order evaluation. The modification adopted in the next subsection turns on the observation that a λ-expression denotes a function.

Blackboard Evaluation

In the absence of conflicting requirements, a sound linguistic principle is "to treat similar entities similarly". Specifically, notice in Figures 3.1-2, 3.1-3, 3.1-5 and 3.1-6 that

(1) basic functions are denoted by single entities (identifiers); and that
(2) when such an entity occupies the top of the control, it is popped and its value pushed into the stack; and that
(3) the value of such a function incorporates all information necessary to apply it to any argument.

since λ-expressions also denote functions, we elect to use these observations as guideposts in extending our blackboard procedure to accommodate semantic trees involving λ-nodes.

Representing λ-Expressions: Obviously there are many different syntactic devices whereby a λ-expression can be recognized as a single entity. In normal-order reduction we have used parentheses for this purpose, whereas in semantic trees each λ-expression is represented by the subtree diverging from a λ-node. For blackboard purposes it is convenient to introduce still a third convention. Noting that a λ-expression is a bundle of information with two parts (a bv part and a body), we first attempt to write a λ-expression such as

$$\lambda x.\ x+3$$

as

$$\lambda^{x}_{x+3}$$

with the bv-part as a superscript and the body as a subscript. Since the body can be arbitrarily complicated, perhaps even involving other λ-expressions, this scheme may prove impractical. Hence we elect consistently to abbreviate body parts, using subscripted δ's to stand for that which is abbreviated. Thus we choose to write the above as

$$\lambda_1^x$$
$$\delta_1 = x+3$$

with this convention the AE of (3.2-1b) is represented as

$$\gamma\ \lambda_1^y\ 3 \qquad (3.2\text{-}2)$$
$$\delta_1 = +\ 2\ y$$

Similarly, we can desugar the PAL program

```
let x =
      let y = 7
      in                                  (3.2-3a)
      2 * y
in
x + 3
```

into the AE

$$(\lambda_1 x.\ x+3)[(\lambda_2 y.\ 2*y)7] \qquad (3.2\text{-}3b)$$

and thence into the control sequence

$$\begin{aligned} &\gamma\ \lambda_1^x\ \gamma\ \lambda_2^y\ 7 \\ &\delta_1 = +\ x\ 3 \\ &\delta_2 = *\ 2\ y \end{aligned} \qquad (3.2\text{-}3c)$$

The critical aspect of our convention is that each λ-expression must be associated with a unique subscript. In addition, however, it is often convenient when writing down each λ-body to represent it in prefix form. Clearly this can be done even when λ-expressions are nested, as in

$$\begin{aligned} &\text{let } y = 7 \\ &\text{in} \\ &\text{let } x = 2 * y \\ &\text{in} \\ &x + 3 \end{aligned} \qquad (3.2\text{-}4a)$$

The equivalent AE is

$$[\lambda_1 y.\ (\lambda_2 x.\ x + 3)(2 * y)]\ 7 \qquad (3.2\text{-}4b)$$

and the corresponding control sequence is

$$\begin{aligned} &\gamma\ \lambda_1^y\ 7 \\ &\delta_1 = \gamma\ \lambda_2^x * 2\ y \\ &\delta_2 = +\ x\ 3 \end{aligned} \qquad (3.2\text{-}4c)$$

The Value of a λ-expression: We are concerned now with the evaluation in the blackboard mechanism of AE's involving λ-expressions, and our immediate question is this: What is the value of a λ-expression? Our principle guideline in answering this question, as given in point (3) on page 3.2-105, is that the value must include all information needed to apply it to arguments. Clearly needed as part of the value are the bv-part and the body of the λ-expression. However, this is not enough, since the body may contain free variables. It is the nature of the binding rules of the λ-calculus, as given in Section 2.3, that the binding of such free variables is determined contextually at the point of appearance of the λ-expression. It therefore follows that the value of a λ-expression must include the environment that exists when the λ-expression is evaluated. We call the value of a λ-expression a λ-closure. A λ-closure is then a bundle of information with three parts:

- a bv,
- a body, and
- an environment.

The bv-part is a single variable and we already have a convention for abbreviating λ-bodies. Needed is a convenient notation for dealing with environments.

The one environment we know about is the primitive environment, PE, which is a mapping from primitive identifiers to primitive obs. As suggested on the right side of Figure 3.1-5, the primitive environment can be augmented by additional name-value couplings, the evaluation of that figure being done in an environment consisting of PE argumented by pairings for the identifiers a, b and c. To see how such pairings come about, consider a combination such as

$$(\lambda x.\ M)\ E \qquad (3.2\text{-}5)$$

In Section 2.3 we considered at length the implications of the intuition that this AE may be evaluated by substituting the AE E for all free occurrences of x in M. But there is another intuitively appealing interpretation of (3.2-5): that it is the same as evaluating M in an environment in which x is paired with the value of E. It is this latter approach that is used in the blackboard evaluator.

Blackboard Mechanism: We want now conventions for blackboard evaluation of AE's involving λ-expressions, conventions that agree with the intuitions just developed. There are only two questions to answer:

(1) What is the nature of a λ-closure, the value of a λ-expression?
(2) How do we apply a λ-closure?

We know enough to answer these questions. In environment Ek, the value of the λ-expression λ^x_m is the λ-closure ${}^k\lambda^x_m$. This writing is clearly a bundle of information with the three proper components. The application of ${}^k\lambda^x_m$ to an argument, say A, is achieved by the following steps:

- Create a new environment Ep, where p has not previously been used. Ep is a copy of environment Ek.
- Add to Ep the coupling of x, the bv of the rator, with A, the rand.
- Evaluate M, the body of the rator, in this environment.
- The value thus produced is returned, in the environment that existed at the time of the original application, as the value of the combination.

All of this is best seen by example. Consider the PAL expression

$$\text{let } x = 3 \text{ in } x - 2 \qquad (3.2\text{-}6a)$$

and the corresponding AE

$$(\lambda x.\ x - 2)\ 3 \qquad (3.2\text{-}6b)$$

with control sequence

$$\begin{array}{l} \delta\ \lambda^x_1\ 3 \\ \delta_1\ =\ -\ x\ 2 \end{array} \qquad (3.2\text{-}6c)$$

We evaluate this control in the primitive environment, the evaluation being shown in Figure 3.2-1. We discuss this evaluation in detail.

It starts in line 1 with the control sequence to be evaluated loaded into

Figure 3.2-1: Evaluation of (3.2-6)

	Control	Stack	Environment
1	$E_0\ \gamma\ \lambda_1^x\ 3$	E_0	0: PE
2	$\gamma\ \lambda_1^x$	3	
3	γ	$^\circ\lambda_1^x\ 3$	
4	$E_1\ \delta_1$	E_1	1: x = 3 (0)
5	$E_1\ -\ x\ 2$	E_1	
6	$-\ x$	2	
7	$E_1\ -$	$3\ 2$	
8	$E_0\ E_1$	$1\ E_1\ E_0$	
9	E_0	$1\ E_0$	
10	⊢	1	

the control of the evaluator. By convention we assume E0 to be the primitive environment, placing into the control and stack matching environment markers. (Only the environment marker in the control serves a necessary purpose. However, placement of a matching environment marker in the stack serves to provide a pleasing symmetry, as well as a useful check that certain kinds of error have not been made.) The transition to line 2 involves looking up the identifier "3" in the primitive environment, yielding 3. We next evaluate the λ-expression λ_1^x. Since evaluation is in environment E0, we get the closure $^\circ\lambda_1^x$. On line 3 we apply this closure to 3, leading to the evaluation of its body (δ_1) in environment E1 which is achieved by augmenting E0 with the pairing (x, 3). The zero in parentheses indicates that E1 is an appendment to E0.

On line 4 we replace the abbreviation δ_1 by that which it abbreviates (as given in (3.2-6c)), leading to line 5. We next look up the identifier "2" in E1. (We know to use E1 because that is the top environment marker in the control on the line we do the evaluation.) Since "2" is not "x", we look next in the environment to which environment 1 is an appendment: E0. In evaluating x in line 6 we immediately get 3. After doing the subtraction we find in line 8 that an environment marker is the top stack item. There is precisely one item in the stack above the top environment marker (the error check mentioned earlier), so the matching environment markers are deleted from control and stack. This operation is repeated on line 9, leading to the answer on line 10. The dashed line encloses that part of the evaluation corresponding to the subproblem of applying the closure.

Let us look at two more examples, the programs of (3.2-3) and (3.2-4), evaluations of which are shown in Figure 3.2-2. As usual, we elide uninteresting steps. For example, rather than put δ_1 into the control in line 3 of Figure 3.2-2a (as we did in line 4 of Figure 3.2-1), we instead load immediately that control sequence which it abbreviates. Note the evaluation of y in line 6 of Figure 3.2-2b. Since the evaluation takes place in environment 2, as evidenced by the fact that E2 is the highest environment marker in the control, we look first in environment layer 2, failing to find y there. Since layer 2 is an appendment to layer 1, as evidenced by the parenthesized 1 at the right end of line 6, we look in layer 1, there finding y.

	Control	Stack	Environment
1	$E_0\ \gamma\ \lambda_1^x\ \gamma\ \lambda_2^y\ 7$	E_0	0: PE
2	γ	$\lambda_2^y\ 7\ E_0$	
3	$E_1\ *\ 2\ y$	E_1	1: $y \rightsquigarrow 7$ (0)
4	$E_1\ *$	$2\ 7$	
5	$\lambda_1^x\ E_1$	$14\ E_1$	
6	$\gamma\ \lambda_1^x$	14	
7	γ	$\lambda_1^x\ 14$	
8	$E_2\ +\ x\ 3$	E_2	2: $x = 14$ (0)
9	$E_2\ +$	$14\ 3$	
10	$E_0\ E_2$	$17\ E_2\ E_0$	
11	E_0	$17\ E_0$	
12	⊢	17	

Figure 3.2-2a: Evaluation of (3.2-3)

	Control	Stack	Environment
1	$E_0\ \gamma\ \lambda_1^y\ 7$	E_0	0: PE
2	γ	$\lambda_1^y\ 7\ E_0$	
3	$E_1\ \gamma\ \lambda_2^x\ *\ 2\ y$	E_1	1: $y = 7$ (0)
4	$\gamma\ \lambda_2^x\ *$	$2\ 7\ E_1$	
5	γ	$\lambda_2^x\ 14$	
6	$E_2\ +\ x\ y$	E_2	2: $x = 14$ (1)
7	$E_0\ E_1\ E_2\ +$	$14\ 7\ E_2\ E_1\ E_0$	
8	⊢	21	

Figure 3.2-2b: Evaluation of (3.2-4)

As another example, consider the PAL program

```
let x = 2
in
let y = 3                                                    (3.2-7a)
in
x*y + (let y = y+3 in x*y) + (let x = y+3 in x*y)
```

with corresponding AE

$$\{\lambda_1 x.\ [\lambda_2 y.\ x*y + (\lambda_3 y.\ x*y)(y+3) + (\lambda_4 x.\ x*y)(y+3)]\ 3\}\ 2 \qquad (3.2\text{-}7b)$$

and control sequence:

$$\begin{aligned} &\gamma\ \lambda_1^x\ 2 \\ &\delta_1 = \gamma\ \lambda_2^y\ 3 \\ &\delta_2 = +\ *\ x\ y\ +\ \gamma\ \lambda_3^y\ +\ y\ 3\ \gamma\ \lambda_4^x\ +\ y\ 3 \\ &\delta_3 = *\ x\ y \\ &\delta_4 = *\ x\ y \end{aligned} \qquad (3.2\text{-}7c)$$

Evaluation is shown in Figure 3.2-3, in which we have introduced one more convention to facilitate these evaluations. In lines 3 and 5 in which new environment layers are formed, it is the case that in <u>both</u> the control <u>and</u> the stack the new environment marker is adjacent to an old one, as it was in lines 3 and 6 in Figure 3.2-2. In such cases we delete the old ones since they can serve no useful purpose in the rest of the evaluation except to be discarded later. (Our objective here is to economize on horizontal space.) Note each evaluation of <u>x</u> and <u>y</u>, since it is important that the right lookup operation be done.

<u>Environment Trees</u>

Although perhaps not immediately obvious, it is true that the notion of a <u>λ-closure</u> as the value of a λ-expression suffices to accommodate all occurrences of λ's in blackboard evaluation. To see this, we need to reconcile blackboard evaluation and normal-order reduction. The key to the relation between the two lies in recognition that blackboard evaluation in general entails the construction of a <u>tree</u> of environments. We see below that the structure of the tree, together with appropriate definition of what is meant by "looking up" an identifier, resolves the conflict of variables problem that complicates normal order reduction. In addition, blackboard evaluation is vastly more efficient than normal order reduction. On the other hand, however, we shall encounter certain meaningful AE's for which the blackboard procedure does not terminate.

<u>Function-Form Definitions</u>: An embryonic tree of environments has already been encountered in Figure 3.9a. To gain further insight into the structure of the environment tree, we now investigate blackboard evaluation of expressions involving programmer-defined functions. Consider, for example, the PAL program

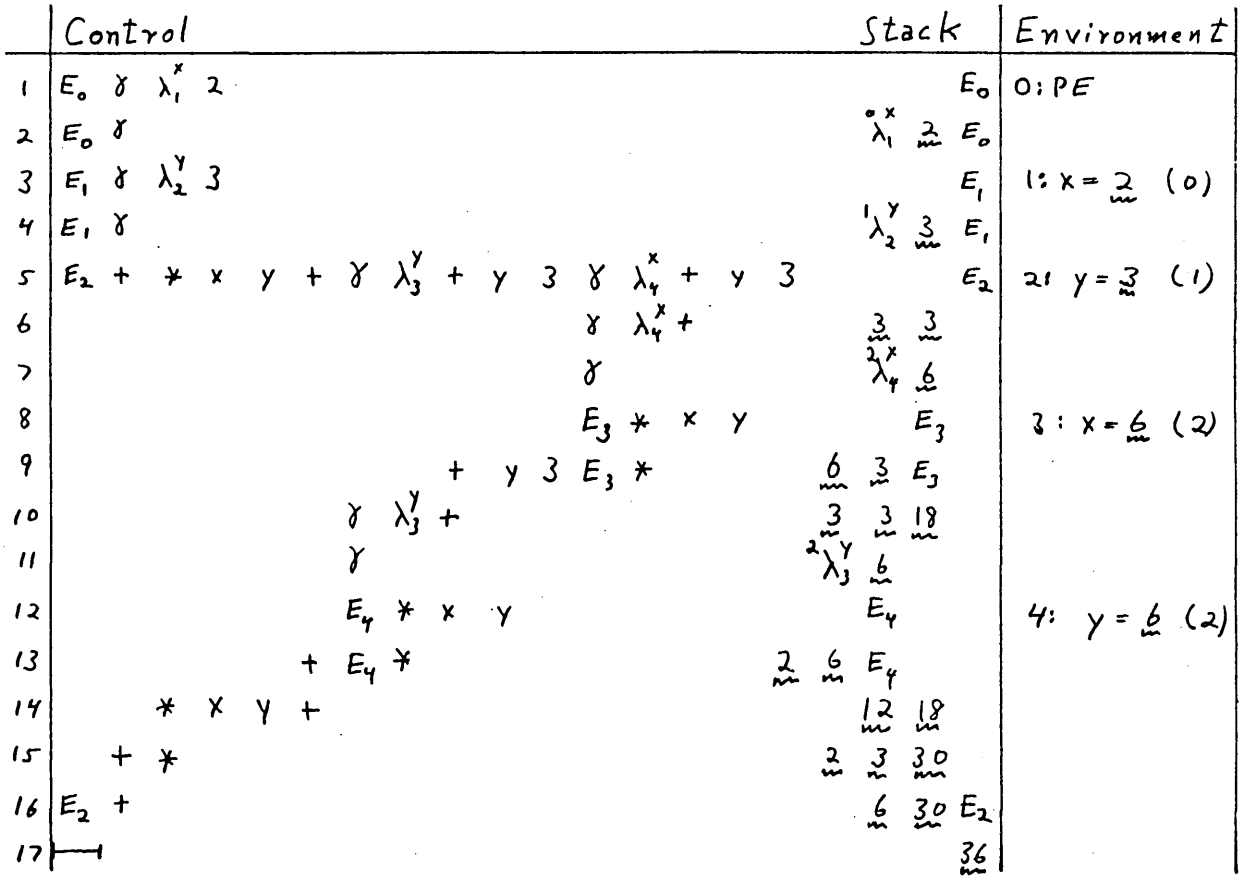


Figure 3.2-3: Evaluation of (3.2-7)

```
let f x = 2 + x
in                                        (3.2-8a)
f 3
```

for which the equivalent AE is

$$(\lambda_1 f.\ f\ 3)\ (\lambda_2 x.\ 2+x) \qquad (3.2\text{-}8b)$$

The corresponding control sequence is

$$\begin{aligned} &\gamma\ \lambda_1^f\ \lambda_2^x \\ &\delta_1 = \gamma\ f\ 3 \\ &\delta_2 = +\ 2\ x \end{aligned} \qquad (3.2\text{-}8c)$$

and evaluation is shown in Figure 3.2-4. We observe from the figure that no new ideas are involved in the evaluation, even though for the first time our blackboard machine now encounters the occurrence of a λ-expression as an

	Control	Stack	Environment
1	$E_0\ \gamma\ \lambda_1^f\ \lambda_2^x$	E_0	0: PE
2	$E_0\ \gamma$	$^\circ\lambda_1^f\ ^\circ\lambda_2^x\ E_0$	
3	$E_1\ \gamma\ f\ 3$	E_1	1: $f = {}^\circ\lambda_2^x$ (0)
4	$E_1\ \gamma$	$^\circ\lambda_2^x\ \underline{3}\ E_1$	
5	$E_2 + 2\ x$	E_2	2: $x = \underline{3}$ (0)
6	$E_2 +$	$\underline{2}\ \underline{3}\ E_2$	
7	⊢	$\underline{5}$	

Figure 3.2-4: Evaluation of (3.2-8)

operand. (Heretofore, they have occurred only as operators.) The reason is that we have already stipulated that the value of a λ-expression relative to the environment in which it occurs is a λ-closure. But a value is a suitable object to couple with a name in an environment layer, so that application of one λ-closure to another (as in the transition from line 2 to line 3) accords with the evaluation procedure already established. The essential concept is that packaging all information necessary to apply a function into a "λ-closure" permits us to treat such entities like any other "value" occurring in the stack. Note that in line 5 in environment 2 the only variable defined beyond PE is x, and that were there an f to be evaluated on that line the evaluation would abort. This is consistent with the fact that we are evaluating a piece of the AE, "2+x", which is not within the scope of f. This is clear from examination of either (3.2-8a) or (3.2-8b).

A still more interesting evaluation arises from the PAL program

```
let g x y = x + y
in
let f = g 2                                    (3.2-9a)
in
f 3 * g 5 4
```

Here we have a function-producing function, and it is gratifying to see that no new techniques are required. The AE equivalent to this PAL program is

$$[\lambda_1 g.\ (\lambda_2 f.\ f\ 3\ *\ g\ 5\ 4)(g\ 2)](\lambda_3 x.\ \lambda_4 y.\ x+y) \qquad (3.2\text{-}9b)$$

and the control sequence is

$$\begin{aligned} &\gamma\ \lambda_1^g\ \lambda_3^x \\ &\delta_1 = \gamma\ \lambda_2^f\ \gamma\ g\ 2 \\ &\delta_2 = *\ \gamma\ f\ 3\ \gamma\ \gamma\ g\ 5\ 4 \\ &\delta_3 = \lambda_4^y \\ &\delta_4 = +\ x\ y \end{aligned} \qquad (3.2\text{-}9c)$$

Examination of the evaluation of this control as shown in Figure (3.2-5) reveals some new points. Note that the λ-expression λ_4^y is evaluated twice, once on line 5 and again on line 9. These evaluations lead to different values, the one on line 5 leading to the closure ${}^2\lambda_4^y$ coupled to an environment in which $\underline{x}$ is $\underline{2}$ and the one in line 9 leading to the closure ${}^4\lambda_4^y$ coupled to an environment in which $\underline{x}$ is $\underline{5}$. Each closure is subsequently used, so the difference has an effect. Curiously, the closure formed first is used second.

The reader is strongly encouraged to test his understanding of the blackboard procedure by attempting to reproduce Figure 3.2-5 without referring to it. Computation is a dynamic process, and mere static observation of a completed blackboard evaluation does not provide adequate insight into the interrelationship between subproblems and the environment tree.

Scope of Variables: The procedure for evaluating -- i.e. looking up -- an identifier was not critical when only the primitive environment was involved. Now, however, we have an environment tree and must be careful about what "looking up an identifier in an environment" means.

The algorithm which we have obeyed implicitly in the preceding examples may be stated explicitly as follows:

1. Let Ek be the current environment, as evidenced by Ek being the topmost environment marker in the control.
2. Does environment layer Ek define the desired identifier? If so the associated value is the one desired and evaluation is complete.
3. Is k zero? That is, did we just look in the primitive environment? If so the evaluation aborts.
4. Let k be the environment to which the layer we examined in step 2 is appended, as evidenced by k appearing in parentheses in the

	Control	Stack	Environment
1	$E_0\ \gamma\ \lambda_1^g\ \lambda_3^x$	E_0	0: PE
2	$E_0\ \gamma$	${}^0\lambda_1^g\ {}^0\lambda_3^x\ E_0$	
3	$E_1\ \gamma\ \lambda_2^f\ \gamma\ g\ 2$	E_1	1: $g = {}^0\lambda_3^x$ (0)
4	γ	${}^0\lambda_3^x\ \underline{2}\ E_1$	
5	$\gamma\ \lambda_2^f\ E_2\ \lambda_4^y$	E_2	2: $x = \underline{2}$ (0)
6	$E_1\ \gamma$	${}^1\lambda_2^f\ {}^2\lambda_4^y\ E_1$	
7	$E_3\ *\ \gamma\ f\ 3\ \gamma\ \gamma\ g\ 5\ 4$	E_3	3: $f = {}^2\lambda_4^y$ (1)
8	γ	${}^0\lambda_3^x\ \underline{5}\ \underline{4}$	
9	$\gamma\ E_4\ \lambda_4^y$	E_4	4: $x = \underline{5}$ (0)
10	γ	${}^4\lambda_4^y\ \underline{4}$	
11	$E_5\ +\ x\ y$	E_5	5: $y = \underline{4}$ (4)
12	$\gamma\ f\ 3\ E_5\ +$	$\underline{5}\ \underline{4}\ E_5$	
13	γ	${}^2\lambda_4^y\ \underline{3}\ \underline{9}$	
14	$E_6\ +\ x\ y$	E_6	6: $y = 3$ (2)
15	$*\ E_6\ +$	$\underline{2}\ \underline{3}\ E_6$	
16	$E_3\ *$	$\underline{5}\ \underline{9}\ E_3$	
17	$\vdash$	$\underline{45}$	

Figure 3.2-5: Evaluation of (3.2-9)

$\gamma\ \lambda_1^g\ \lambda_3^x$

$\delta_1 = \gamma\ \lambda_2^f\ \gamma\ g\ 2$

$\delta_2 = *\ \gamma\ f\ 3\ \gamma\ \gamma\ g\ 5\ 4$

$\delta_3 = \lambda_4^y$

$\delta_4 = +\ x\ y$

environment column to the right of the name-value coupling. Continue at step 2.

Insight into how this procedure accords each definition its proper scope may be gained by reexamination of Figure 3.2-2. The two AE's (rewritten here for ease of reference) evaluated in that figure are

$$(\lambda x.\ x+3)[(\lambda y.\ 2*y)\ 7] \qquad (3.2\text{-}10a)$$

and

$$[\lambda y.\ (\lambda x.\ x+3)(2*y)]\ 7 \qquad (3.2\text{-}10b)$$

Since the scope rules of the λ-calculus (see page 2.3-66) imply that each occurrence of "x" in these two AE's can be changed to "y" without affecting meaning, blackboard evaluation must also be invariant to such renaming. To see that this is so, note in Figure 3.2-2a that the environment layers corresponding to the definitions of "x" and "y" occupy parallel branches of the environment tree, reflecting the "parallel" occurrence of the λ's in (3.2-10a). In Figure 3.2-2b the two layers are in series, reflecting the nested occurrence of the λ's in (3.2-10b). In both cases our rule for looking up identifiers returns the correct value even if all x's are changed to y's because conflicting definitions are either by-passed or obscured by a subsequent definition.

Contrast with Normal Order Reduction: An important aspect of our blackboard procedure is that it departs from normal order. Blackboard order of evaluation is clearly as follows:

> (1) Evaluate the rand. (2) Evaluate the rator. (3) Apply the one to the other.

Although the Church-Rosser theorem guarantees that we cannot get wrong answers from this order, it also alerts us to the possibility that we sometimes may fail to get an answer when one exists. We return to this point later.

There are other implications to blackboard procedure. First and most obvious, in blackboard evaluation the meta-function subst is superseded by separate evaluation of each free occurrence of an identifier as it reaches the top of the control. Thus we obviate the complicated operations of changing bound variables to avoid conflicts, searching through a λ-body to determine all free occurrences of the bound variable, and substituting an AE (possibly of great complexity) for each such occurrence.

The second simplification is less obvious but even more substantive, and concerns efficiency. Often, a programmer chooses to name an AE because he intends to make numerous references to it, as in

$$(\lambda c.\ \_\ c\ \_\ c\ \_\ c\ \_)\ M \qquad (3.2\text{-}11a)$$

where M stands for an arbitrary AE and the body of the λ-expression indicates three free occurrences of the bound variable "c". Normal order reduction produces the AE

$$\sim M \sim M \sim M \sim \qquad (3.2\text{-}11b)$$

so that when evaluation finally does take place, M is evaluated three times. By contrast, in the blackboard procedure M is evaluated only once, and this value then looked-up three times. Since the complexity of M is arbitrary and "c" may occur free any number of times within the λ-body, the efficiency gained by blackboard evaluation can be enormous. Indeed, normal order reduction of functional composition implies a blow-up of computation that grows exponentially with depth of nesting. For example, the PAL expression

$$f\ (f\ (7\text{-}2))\ \text{where}\ f\ u = u+(u*u) \qquad (3.2\text{-}12a)$$

translates into the AE

$$[\lambda f.\ f\ (f\ (7\text{-}2))]\ (\lambda u.\ u+u*u) \qquad (3.2\text{-}12b)$$

which under normal order reduction yields

$$\begin{aligned}&(\lambda u.\ u+u*u)\ [(\lambda u.\ u+u*u)\ (7\text{-}2)]\\ &\xrightarrow{\beta} (\lambda u.\ u+u*u)\ (7\text{-}2) + (\lambda u.\ u+u*u)\ (7\text{-}2) * (\lambda u.\ u+u*u)\ (7\text{-}2)\\ &\simeq [(7\text{-}2) + (7\text{-}2) * (7\text{-}2)] + [(7\text{-}2) + (7\text{-}2) * (7\text{-}2)]\\ &\qquad * [(7\text{-}2) + (7\text{-}2) * (7\text{-}2)]\end{aligned} \qquad (3.2\text{-}12c)$$

Evaluation of the resulting normal-form expression involves 3^2 separate calculations of the value of the subexpression "(7-2)", plus (3^2-1) arithmetic operations on its value, whereas Figure 3.2-6 shows that blackboard evaluation

Control	Stack	Environment
$E_0\ \gamma\ \lambda_1^f\ \lambda_2^u$	E_0	0: PE
$E_0\ \gamma$	$q_1^f\ q_2^u\ E_0$	
$E_1\ \gamma\ f\ \gamma\ f\ -\ 7\ 2$	E_1	1: $f = {}^\circ\lambda_2^u$ (0)
$\gamma\ f\ -$	$7\ 2\ E_1$	
γ	${}^\circ\lambda_2^u\ 5$	
$E_2\ +\ u\ *\ u\ u$	E_2	2: $u = 5$ (0)
$+\ u\ *$	$5\ 5\ E_2$	
$\gamma\ f\ E_2\ +$	$5\ 25\ E_2$	
$E_1\ \gamma$	${}^\circ\lambda_2^u\ 30\ E_1$	
$E_3\ +\ u\ *\ u\ u$	E_3	3: $u = 30$ (0)
$+\ u\ *$	$30\ 30\ E_3$	
$E_3\ +$	$30\ 900\ E_3$	
$\vdash$	930	

Figure 3.2-6: Evaluation of (3.2-12b)
$\gamma\ \lambda_1^f\ \lambda_2^u$
$\delta_1 = \gamma\ f\ \gamma\ f\ -\ 7\ 2$
$\delta_2 = +\ u\ *\ u\ u$

of (3.2-12b) involves essentially only the calculations

$$\begin{aligned} (7-2) &= 5 \\ 5*5 &= 25 \\ 5+25 &= 30 \\ 30*30 &= 900 \\ 900+30 &= 930 \end{aligned}$$

More generally, for k nested compositions

$$f\ (f\ (\ldots\ (f\ (7-2))\ldots))\quad \text{where}\quad f(u) = u+u*u \qquad (3.2\text{-}13)$$

normal order reduction implies 3^k separate calculations of "(7-2)" and (3^k-1) subsequent arithmetic operations, whereas the number of calculations with blackboard evaluation grows only linearly, not exponentially, with k.

In the overall scheme of things, it can be argued that inefficiencies to within a linear factor are not of dominant concern. But one can not trifle with exponentials! The engineering motivation for evaluation before functional application, instead of normal order reduction before evaluation, is avoidance of an exponential blow-up in amount of computation.

On the other hand, efficiency is not gained without cost: By departing from normal order we forfeit ability to evaluate certain expressions which are meaningful in the sense that normal order reduction terminates. For example, the familiar AE

$$(\lambda x.\ \lambda y.\ x)\ 5\ [(\lambda u.\ u\ u)(\lambda v.\ v\ v)] \qquad (3.2\text{-}14)$$

terminates under normal-order reduction, but not under blackboard evaluation. Moreover, the cost is not restricted to uninteresting cases such as (3.2-14): It is easy to verify that the blackboard procedure does not terminate for any AE of the form

$$Z\ F\ a1\ a2\ \ldots\ ak \qquad (3.2\text{-}15)$$

where Z is the fixed point generating λ-expression defined in (2.4-16), page 2.4-86, so that we need alternate methods for accommodating recursion.

Special Constructs

In this section we investigate certain constructions of applicative expressions that prove especially useful in programming practice. In particular we discuss simultaneous and within definitions.

Simultaneous Definitions: It is frequently convenient to have several definitions in parallel, and PAL provides for the possibility. The PAL program

```
let a = A
and b = B
in
P
```
(3.2-16a)

means to evaluate both A and B in the current environment, and then to evaluate P in an environment in which a is coupled to the value of A and b to the value

of B. (Assume A, B and P to be arbitrary AE's.) Note how (3.2-16a) differs from

```
let a = A
in
let b = B                                   (3.2-16b)
in
P
```

Here any free occurrence of a in B is bound to the value of A, while in the former program any a or b which is free in either of A or B is free in the entire expression.

An alternate form for (3.2-16a) is

```
let a, b = A, B
in                                          (3.2-16c)
P
```

The comma to the right of the equal signifies a tuple, but that on the left is a syntactic device specifying how the right side is to be decomposed. For example, in

```
let a, b, c = W in P                        (3.2-17)
```

it is necessary that W denote a 3-tuple.

There are two choices available to us to explain the semantics of simultaneous definitions: We can show how to handle them in the λ-calculus, or we can explicate them initially in terms of the gedanken evaluator. The latter choice would have the consequence that we would not be able to use simultaneous definitions in writing the PAL program for the gedanken evaluator. (Why not?) We thus elect to show how simultaneous definitions can be handled in the λ-calculus. Since the explanation leads to inefficient evaluation, we provide an alternate explanation for blackboard evaluation and for the gedanken evaluator.

An obvious way to desugar (3.2-16c) is into the following AE:

```
[λ(a, b). P] (A, B)                         (3.2-18a)
```

This captures immediately the intuition we expect from (3.2-16c), just as

```
let x = E in M
```

and

```
(λx. M) E
```

are equivalent. Unfortunately this is not a desirable way to proceed. Recall that in Section 2.3 the precise specification of a suitable subst metafunction was a distinctly non-trivial business, the problem being to avoid ill effects from the clash of bound variables. Rules for simultaneous substitutions such as would be required for (3.2-18) would be even more complex. Fortunately, we can avoid the need for them.

Consider the AE

$$[\lambda\pi.\ (\lambda a.\ \lambda b.\ P)(\pi\ 1)(\pi\ 2)]\ (A,\ B) \qquad (3.2\text{-}18b)$$

where π is some identifier not occurring in P, A or B. In evaluating this AE (either in the blackboard mechanism or by normal order reduction), we first associate π with the 2-tuple (A, B) and then associate a with the first component of the 2-tuple and b with the second. That is, any a in P is associated with A and any b with B. But this is precisely the effect intended from (3.2-18). Thus we can regard (3.2-19) as a desugaring of (3.2-18) into a form we already can handle.

There is a minor difficulty with this desugaring: It ascribes semantics to certain AE's which we would prefer to call erroneous. Consider the PAL program

$$\text{let a, b = W in P} \qquad (3.2\text{-}19a)$$

and the corresponding AE

$$[\lambda(a,\ b).\ P]\ W \qquad (3.2\text{-}19b)$$

Our semantic intent is that each of these be undefined if W denotes any value other than a 2-tuple. By our rule, (3.2-19b) desugars further into

$$[\lambda\pi.\ (\lambda a.\lambda b.\ P)(\pi\ 1)(\pi\ 2)\ \ W \qquad (3.2\text{-}19c)$$

This AE is defined if W denotes any object whatsoever which may be applied to each of 1 and 2. Thus W might denote a 3-tuple, or in fact any function which includes 1 and 2 in its domain of definition. Since this state of affairs is not our intent, we characterize this as a weak representation -- one that has extra properties not intended. The problem is easy to fix: We can replace (3.2-19c) by the AE

$$[\lambda\pi.\ \text{Order}\ \pi\ \text{eq}\ 2\ \text{->}\ (\lambda a.\ \lambda b.\ P)(\pi\ 1)(\pi\ 2)\ |\ \text{error}]\ W \qquad (3.2\text{-}19d)$$

where "error" is some expression whose value is undefined. This AE is clearly defined only if W denotes a 2-tuple. (If W does not denote a tuple at all, then "Order" aborts.)

An interesting problem obtrudes. There is no particular reason to replace (3.2-19b) by (3.2-19c) rather than by

$$[\lambda\pi.\ (\lambda b.\ \lambda a.\ P)(\pi\ 2)(\pi\ 1)]\ W$$

which has identical semantics. But consider the rather foolish PAL program

$$\text{let x = A and x = B in W} \qquad (3.2\text{-}20a)$$

which desugars as

$$[\lambda(x,\ x).\ W]\ (A,\ B) \qquad (3.2\text{-}20b)$$

If the next step is

$$[\lambda\pi.\ (\lambda x.\ \lambda x.\ W)(\pi\ 1)(\pi\ 2)]\ (A,\ B) \qquad (3.2\text{-}20c)$$

then any free x is associated with the value of B, whereas if the next step is

$$[\lambda\pi.\ (\lambda x.\ \lambda x.\ W)(\pi\ 2)(\pi\ 1)]\ (A,\ B) \qquad (3.2\text{-}20d)$$

such x's are associated with A. We have a design choice:

(1) We can disallow repeated occurrence of the same identifier in a bv-part, thus rendering illegal (3.2-20b) and hence (3.2-20a).

(2) We can arbitrarily adopt one or the other ordering as part of the definition of PAL.

(3) We can leave undefined the meaning of AE's with this problem.

Although (1) is common in most programming languages, we have opted for (3) for PAL. The force of this decision is that the user of PAL cannot predict the effect of such expressions, so that (3) should be as effective as (1) in discouraging use of them. We return to this discussion in Section 3.5, in which we suggest a mechanism for the gedanken evaluator which causes expressions such as (3.2-20a) to be undefined.

For the purpose of explaining the R-PAL gedanken evaluator programs, we adopt the desugaring of (3.2-19c). (We do not need (3.2-19d) since the programs are correct!) It is easy to provide an alternate approach which is more attractive for blackboard evaluation. Consider the PAL program

```
let a = 1
and f (x, y) = x + y
in                                        (3.2-21a)
f (a, 2)
```

we desugar this first into

```
let a = 1
and f = λ(x, y). x + y
in                                        (3.2-21b)
f (a, 2)
```

and then into

$$[\lambda(a,\ f).\ f(a,\ 2)]\ [1,\ (\lambda(x,\ y).\ x + y)] \qquad (3.2\text{-}21c)$$

for which the control sequence is

$$\begin{aligned} &\gamma\ \lambda_1^{a,f}\ \mathcal{J}_2\ 1\ \lambda_2^{x,y} \\ &\delta_1 = \gamma\ f\ \mathcal{J}_2\ a\ 2 \\ &\delta_2 = +\ x\ y \end{aligned} \qquad (3.2\text{-}21d)$$

Blackboard evaluation of this control is shown in Figure 3.2-7:

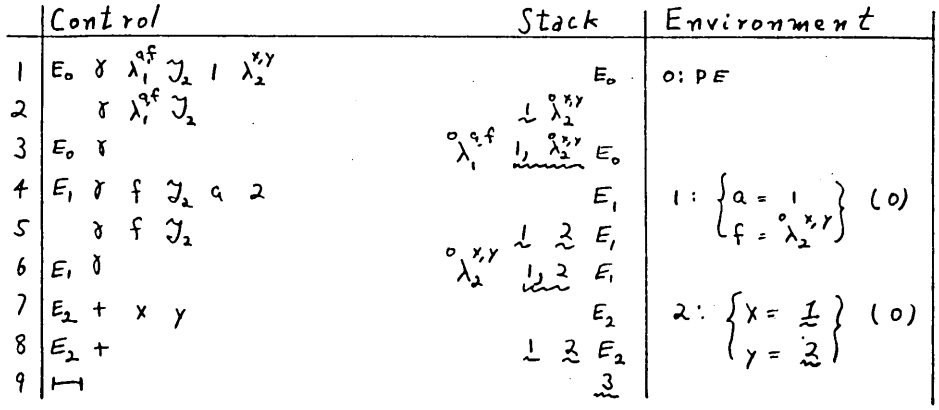


Figure 3.2-7: Evaluation of (3.2-21)

On each of lines 3 and 6 we apply a λ-closure whose bv has two components. Since the rand is a 2-tuple the application is legal, and we create an environment layer in which two variables are defined. The rest of the evaluation is as before.

Definitions Using "within": Consider the PAL program

```
let f x =
      let c = C in P
in
f 2 + f 3 + f 4
```
(3.2-22a)

This program desugars into the AE

$$(\lambda_1 f.\ f\ 2 + f\ 3 + f\ 4)\ [\lambda_2 x.\ (\lambda_3 c.\ P)\ C] \qquad (3.2\text{-}22b)$$

In blackboard evaluation the body of λ_2^x is evaluated three times -- once for each free occurrence of f in δ_1. That is, C is evaluated each time f is applied. But if C is a rather complex expression, we would like to be able to rewrite (3.2-22a) so as to evaluate it only once. It is not hard to rework (3.2-22b) into an AE with the desired effect. Since such a construction is useful, PAL includes a special sugaring for it.

The AE form of the expression we want is easily seen to be

$$(\lambda_1 f.\ f\ 1 + f\ 2 + f\ 3)\ [(\lambda_2 c.\ \lambda_3 x.\ P)\ C] \qquad (3.2\text{-}23a)$$

Evaluation of this AE in blackboard order leads to exactly one evaluation of C, as is easy to see. The control sequence for this AE is

$$\begin{aligned} &\gamma\ \lambda_1^f\ \gamma\ \lambda_2^c\ C \\ &\delta_1 = +\ \gamma\ f\ 1 + \gamma\ f\ 2\ \gamma\ f\ 3 \\ &\delta_2 = \lambda_3^x \\ &\delta_3 = P \end{aligned} \qquad (3.2\text{-}23b)$$

and the first few lines of blackboard evaluation are shown in Figure 3.2-8.

	Control	Stack	Environment
1	$E_0\ \gamma\ \lambda_1^f\ \gamma\ \lambda_2^c\ C$		E_0 0: P E
2	γ	$^0\lambda_2^c\ \underline{C}_m$	
3	$\gamma\ \lambda_1^f\ E_1\ \lambda_3^x$	E_1	1: c = $\underline{C}$ (0)
4	$E_0\ \gamma$	$^0\lambda_1^f\ ^1\lambda_3^x$	E_0
5	$E_2\ \delta_1$		E_2 2: f = $^1\lambda_3^x$ (0)

Figure 3.2-8: Partial Blackboard Evaluation of (3.2-23b)

Clearly $\underline{C}$ is evaluated exactly once in this evaluation, so that application of $\underline{f}$ in δ_1 leads to evaluation of $\underline{P}$ in an environment in which $\underline{c}$ is associated with the value of $\underline{C}$. Note that the efficiency gain shows up only in blackboard order and not in normal order. In the latter there are still three evaluations of $\underline{C}$ since (3.2-23a) leads in one beta-reduction to

$$[(\lambda c.\ \lambda x.\ P)\ C]\ 1\ +\ [(\lambda c.\ \lambda x.\ P)\ C]\ 2\ +\ [(\lambda c.\ \lambda x.\ P)\ C]\ 3 \qquad (3.2\text{-}24)$$

Since this form of AE is useful, we want to make it easy to write PAL programs which desugar to it. For simplicity let us consider

$$(\lambda f.\ F)\ [(\lambda c.\ \lambda x.\ P)\ C] \qquad (3.2\text{-}25)$$

This can be sugared to

$$\text{let f = } (\lambda c.\ \lambda x.\ P)\ C \text{ in F} \qquad (3.2\text{-}26a)$$

and then into

```
let f =
      let c = C
      in
      λx. P                    (3.2-26b)
in
F
```

However, there is no way to sugar this into a PAL program not involving λ's, unless one introduces a new variable. (The λ's can <u>always</u> be sugared out of an AE by introducing new variables. Consider

$$\lambda x.\ P \overset{\beta}{\leftarrow} (\lambda f.\ f)\ (\lambda x.\ P)$$

which sugars to

```
let f x = P in f
```

But this is not what we want.)

The PAL form for (3.2-25) involves a <u>within</u> construct, like this:

```
let
        c = C
        within
    f x = P                                    (3.2-27)
in
F
```

The semantics of the within construct is given by saying that the definition

```
a = A within b = B                             (3.2-28a)
```

is equivalent to the definition

```
b = (let a = A in B)                           (3.2-28b)
```

and hence to the definition

```
b = (λa. B) A                                  (3.2-28c)
```

The form becomes particularly useful in connection with simultaneous definitions, since we can have

```
let
        c = C
        within
    f x = F
    and                                        (3.2-29a)
    g y = G
in
P
```

Here the scope of c is to be F and G, and no more. (Parentheses are not needed.) Successive desugarings are as follows:

```
let c = C within [f = λx. F and g = λy. G] in P
 let c = C within f, g = (λx. F), (λy. G) in P
let f, g = [let c = C in (λx. F), (λy. G)] in P        (3.2-29b)
    let f, g = (λc. (λx. F), (λy. G)) C in P
    [λ(f, g). P] [(λc. (λx. F), (λy. G)) C]
```

Except in the last line, the square brackets are not needed but are included only to aid the reader. (The parentheses around the λ-expressions are required by PAL's syntax.) Note that the scope of c in the last line is clearly F and G, as required.

It turns out that we can write PAL programs which, under the rules given so far, desugar into forms we have not yet seen. Consider

```
let
     (          c = C
                within
           f x = F
           and
           g y = G                                   (3.2-30a)
     )
     and
     h z = H
in
P
```

The effect of the parentheses is that the scope of c is to include F and G as in (3.2-29), but not H. Desugaring the within construct as in the first four lines of (3.2-29b) leads to

```
let f, g = (λc. (λx. F), (λy. G)) C
and h z = H                                          (3.2-30b)
in
P
```

To save writing we define the abbreviation

$$W \equiv [\lambda c.\ (\lambda x.\ F),\ (\lambda y.\ G)]\ C \qquad (3.2\text{-}30c)$$

Then the next step in desugaring leads to

$$\text{let } (f,\ g),\ h = W,\ (\lambda z.\ H) \text{ in } P \qquad (3.2\text{-}30d)$$

and the next step is

$$[\lambda((f,\ g),\ h).\ P]\ [W,\ (\lambda z.\ H)] \qquad (3.2\text{-}30e)$$

Even though (3.2-30a) is a perfectly valid PAL program, (3.2-30d) and (3.2-30e) are not legal PAL: We do not permit "structured bv-parts" in PAL if there is more structure than just a listing. (Each line in (3.2-29b) is a legal PAL program -- if each λ is replaced by fn.)

We elect not to attempt to explain (3.2-30) at this time, postponing the explanation to Section 3.5. At that time, we give the explanation in terms of the gedanken evaluator. The effect of this decision is that we may not, in writing the PAL programs for the gedandken evaluator, use constructs which desugar into AE's with structured bv-parts. We accept the restrictions, since the facility is not one which is needed and the explanation is most naturally given later. Note that this is the opposite decision from the one made for simultaneous definitions.

3.3 Conditional Expressions

Heretofore our only stated decision concerning order of evaluation in the blackboard machine has been to defer evaluation of a λ-body until application of the λ-closure to which it belongs. We saw at the end of section 2.3 conjunction with the Church-Rosser theorem that order becomes particularly significant when conditionals (a form of annihilator) are introduced. Specifically, PAL expressions of the form

```
B -> M | N                                    (3.3-1)
```

are to have the meaning denoted by M if the AE B denotes true, and that of N if B denotes false. We know that we must evaluate the premise B before either the true arm M or the false arm N if we are to avoid undefinedness in cases such as

```
c eq 0 -> c/2 | 2/c                           (3.3-2a)
```

or infinite loops in recursive functions such as

```
let rec f n = n eq 0 -> 1 | n*f(n-1)
in                                            (3.3-2b)
f 3
```

Recall from section 2.4 that evaluation of the AE corresponding to (3.3-2b) depends critically on evaluation of the boolean ("n eq 0") and selection of one of the two arms of the conditional before attempting to evaluate either arm.

Control of Order of Evaluation

Needed is a method for evaluating AE's involving conditionals which insures that only the desired arm is evaluated. As is often the case, we have a choice of ways to proceed, and the decision is somewhat arbitrary. We will discuss two solutions: one conceptually more elegant, and the other with noticeable practical advantages.

It is always pleasing conceptually to be able to describe a new construct in terms of existing ones without the need to postulate new "built-in" functions, and doing so turns out to be possible in the present case. The possibility parallels the treatment of conditionals in the λ-calculus (cf. page 2.3-77), in which the conditional expression

```
B -> M | N                                    (3.3-3)
```

is considered as sugaring for the curried combination

```
Q B M N                                       (3.3-4)
```

and Q has the property that

$$\begin{aligned} Q\ \text{true} &\xrightarrow{\delta} \lambda x.\ \lambda y.\ x \\ Q\ \text{false} &\xrightarrow{\delta} \lambda x.\ \lambda y.\ y \end{aligned} \qquad (3.3.5)$$

we must now take account of the difference in order of evaluation between the blackboard procedure and normal-order reduction, since it turns out to be an important difference. In the case of (3.3-4) the blackboard mechanism evaluates first the control structure corresponding to N, then that of M, and finally that of B. This obviously is unacceptable in cases such as (3.3-2).

Fortunately, it is easy to force the blackboard mechanism into simulating the effect of normal-order reduction. The trick turns upon the blackboard decision to defer evaluation of a λ-body until the closure of which it is a part is applied. When an expression such as

$$[\lambda x.\ \sim\sim x \sim\sim]\ M \qquad (3.3\text{-}6)$$

is evaluated in normal order, M is not evaluated until, in the course of evaluating the body of the λ-expression, its instance which replaces x is encountered. In evaluation under blackboard rules M will be evaluated first. To achieve in the blackboard mechanism the effect of normal order, we observe that M in

$$[\lambda x.\ \sim\sim(x\ \text{nil})\sim\sim][\lambda().\ M] \qquad (3.3\text{-}7a)$$

is not evaluated until x is applied to nil, since (3.3-7a) is equivalent under axiom β to

$$\sim\sim([\lambda().\ M]\ \text{nil})\sim\sim \qquad (3.3\text{-}7b)$$

and hence to

$$\sim\sim M \sim\sim \qquad (3.3\text{-}7c)$$

It follows then that blackboard evaluation of (3.3-7a) simulates normal order evaluation of (3.3-6), at least insofar as evaluation of M is concerned.

The equivalence of (3.3-7a) and (3.3-6) suggests a general technique whereby a programmer can often override the PAL decision to evaluate operands before doing functional application. (We see in Section 3.4 that the technique does have limitations.) In particular, we can accomodate conditionals in our blackboard evaluator by using

$$Q\ B\ [\lambda().M]\ [\lambda().\ N]\ \text{nil} \qquad (3.3\text{-}8)$$

instead of (3.3-4) as the desugaring of (3.3-3).

An Example: To see that the desugaring suggested in (3.3-8) works, we examine blackboard evaluation of the PAL program

```
let f x = x > 0 -> x | -x
in                                   (3.3-9a)
f 2 + f(-3)
```

The corresponding AE is

$$[\lambda_1 f.\ f\ 2 + f(-3)]\ [\lambda_2 x.\ x>0 \rightarrow x \mid -x] \qquad (3.3\text{-}9b)$$

Given our decision of (3.3-8), the body of the second λ-expression is further desugared as

$$Q\ (x > 0)\ (\lambda_3().\ x)\ (\lambda_4().\ -x)\ nil \qquad (3.3\text{-}10)$$

Then the entire control structure is

$$\begin{aligned} &\gamma\ \lambda_1^f\ \lambda_2^x \\ &\delta_1 = +\ \gamma\ f\ 2\ \gamma\ f\ neg\ 3 \\ &\delta_2 = \gamma\ \gamma\ \gamma\ \gamma\ Q > x\ 0\ \lambda_3^{()}\ \lambda_4^{()}\ nil \\ &\delta_3 = x \\ &\delta_4 = neg\ x \end{aligned} \qquad (3.3\text{-}11)$$

We must now implement our decision of (3.3-5) about application of Q. First we must decide on what abstract object in the universe of discourse is denoted by Q. Let us use the writing $\underline{Q}$ to denote that ob. All that we can know about an ob is how it transforms, and all of $\underline{Q}$'s properties are given in (3.3-5). We thus decree that application of $\underline{Q}$ to true in the blackboard machine leads to the closure λ_t^x, and that application of $\underline{Q}$ to false leads to λ_f^x, where

$$\begin{aligned} \delta_t &= \lambda_{t'}^{\gamma} \\ \delta_{t'} &= x \\ \delta_f &= \lambda_{f'}^{\gamma} \\ \delta_{f'} &= y \end{aligned} \qquad (3.3\text{-}12)$$

Note that (3.2-12) follows directly from (3.3-4). When we make the substitution of (3.3-12) we choose to form the closures in the then-current environment. (No consequences follow from this decision, since there are no free variables in either of the λ-expressions shown in (3.3-5).)

An example using this method is shown in Figure 3.3-1. In line 6 we evaluate "-3 > 0", leading to false in the stack. We next apply $\underline{Q}$ to false in line 7, and get the closure $^2\lambda_f^x$ in the stack in line 8. The actual selection of an arm of the conditional is performed in line 11, leading to $^2\lambda_f^0$ in the stack in line 12. Similarly, $^6\lambda_t^x$ is produced in line 18, leading later to the selection in line 23 of the true arm, $^6\lambda_3^{()}$. Environment layers 5 and 9, produced by the application to nil of λ-closures with empty bv-parts, have no name-value associations; but they are required to complete the linkage of environment layers. It is this linkage, for example, that permits evaluation of $\underline{x}$ in line 13, $\underline{x}$ being defined in layer 2.

Another Method: Although the method we have discussed clearly works, it is obviously clumsy: It seems reasonable that it should be possible to do the entire selection of lines 7 to 12 in one step. The problem is that we have used general tools to solve a particular problem. Since conditionals are an important concept in programming linguistics, it seems appropriate to provide a specific tool to process them, a tool that leads to greater efficiency. We now develop such a tool.

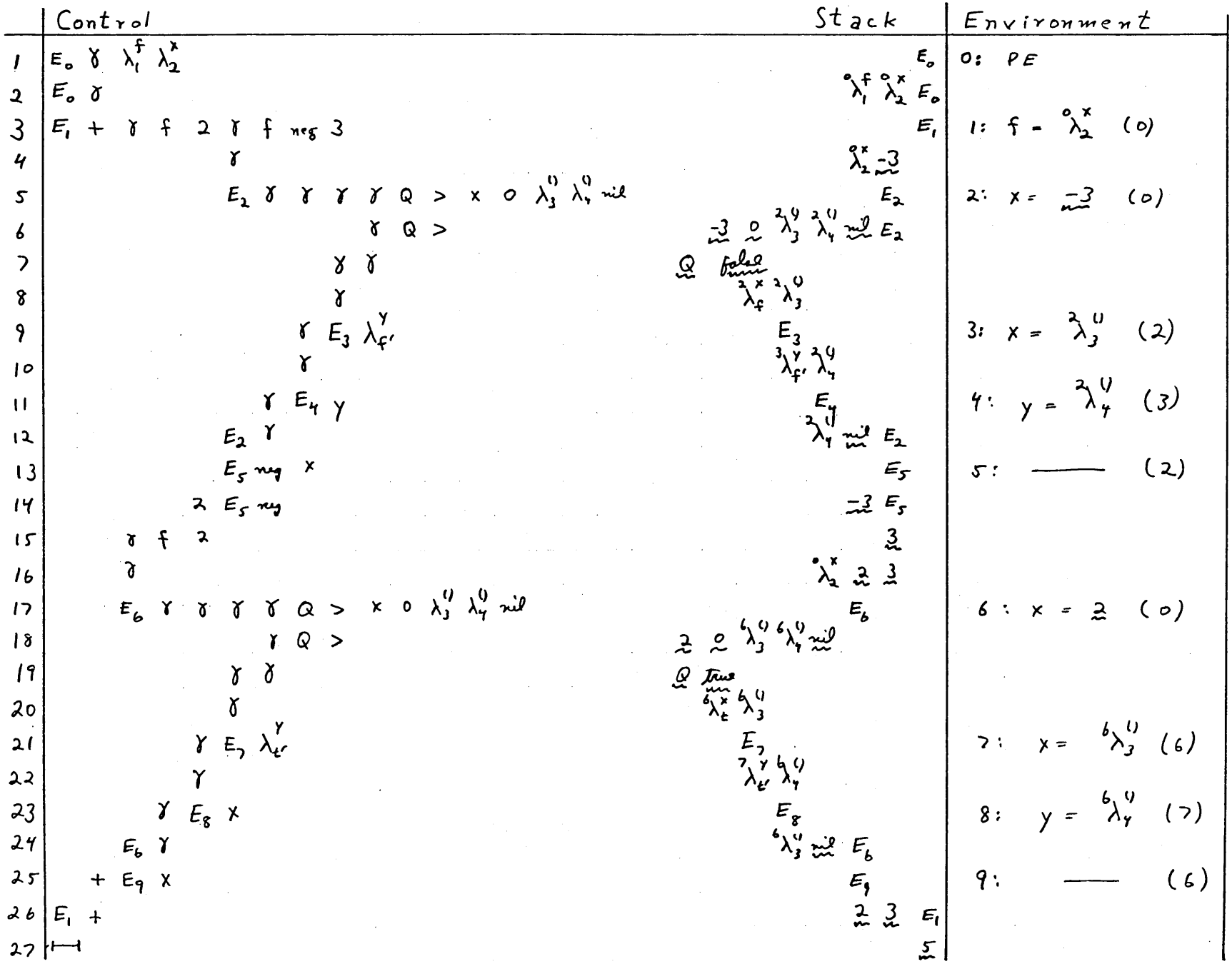


Figure 3.3.1: Evaluation of (3.3.1)

Instead of regarding the boolean expression

B -> M | N (3.3-13)

as sugaring for the combination shown in (3.3-4) and hence the tree shown in Figure 3.3-2, we introduce instead a new type of node -- β -- and regard

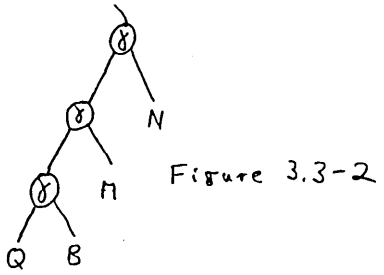


Figure 3.3-2

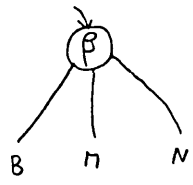


Figure 3.3-3

(3.3-13) as sugaring for the tree in Figure 3.3-3. The convention is that the left son of a β node is the boolean expression, the center son is the true arm and the right son is the false arm. Needed is a control sequence for this. Now note that the selection of an arm in Figure 3.3-1 was facilitated by the fact that each of the two arms of the conditional was represented in the control by a single control item: $\lambda_3^{()}$ and $\lambda_4^{()}$. We thus decide to use δ's to abbreviate the arms, so that the control corresponding to the conditional expression in (3.3-9b) which under the earlier method gave rise to (3.3-10) is now

$$\delta_3\ \delta_4\ \beta\ >\ x\ 0$$

The blackboard rule is then:

> When β is the top item in the control, the top item in the stack must be a truthvalue. If it is true, then the second item in the control is discarded, and evaluation continues with the (old) third item as the top of the control. If the top of the stack is false, then the third item of the control is discarded and the (old) second item is used. In either case, the β and the top stack item are discarded.

Thus we represent the PAL program (3.3-9a) and hence the AE (3.3-9b) by the following control structure:

$$\begin{aligned} &\gamma\ \lambda_1^f\ \lambda_2^x \\ &\delta_1 = +\ \gamma\ f\ 2\ \gamma\ f\ \text{neg}\ 3 \\ &\delta_2 = \delta_3\ \delta_4\ \beta\ >\ x\ 0 \\ &\delta_3 = x \\ &\delta_4 = \text{neg}\ x \end{aligned} \qquad (3.3\text{-}14)$$

The convention guarantees that whenever β occupies the top of the control, then the second and third items in the control will each be δ's. Figure 3.3-4 shows evaluation of the control structure. The transitions on lines 7 and 13 are the interesting ones. On line 7, the top of the control is β and the top of the stack is false, so we select δ_4 (as shown on line 8), the second element of the control, discarding the δ_3 as well as the β and false. (In this example and in all subsequent ones, we aid the reader by underlining the δ selected by a β.

	Control	Stack	Environment
1	$E_0\ \gamma\ \lambda_1^f\ \lambda_2^x$	E_0	0: PE
2	$E_0\ \gamma$	$^\circ\lambda_1^f\ ^\circ\lambda_2^x\ E_0$	
3	$E_1\ +\ \gamma\ f\ 2\ \gamma\ f\ neg\ 3$	E_1	1: f = $^\circ\lambda_2^x$ (0)
4	γ	$^\circ\lambda_2^x\ \underline{-3}\ E_1$	
5	$E_2\ \delta_3\ \delta_4\ \beta\ >\ x\ 0$	E_2	2: x = $\underline{-3}$ (0)
6	$\beta\ >$	$\underline{-3}\ \underline{0}\ E_2$	
7	$E_2\ \delta_3\ \underline{\delta_4}\ \beta$	$\underline{false}\ E_2$	
8	$E_2\ \delta_4$	E_2	
9	$\gamma\ f\ 2\ E_2\ neg\ x$	E_2	
10	γ	$^\circ\lambda_2^x\ \underline{2}\ \underline{3}$	
11	$E_3\ \delta_3\ \delta_4\ \beta\ >\ x\ 0$	E_3	3: x = $\underline{2}$ (0)
12	$\beta\ >$	$\underline{2}\ \underline{0}\ E_3$	
13	$E_3\ \underline{\delta_3}\ \delta_4\ \beta$	$\underline{true}\ E_3$	
14	$+\ E_3\ x$	E_3	
15	$E_1\ +$	$\underline{2}\ \underline{3}\ E_1$	
16	$\vdash$	$\underline{5}$	

$\gamma\ \lambda_1^f\ \lambda_2^x$

$\delta_1 = +\ \gamma\ f\ 2\ \gamma\ f\ neg\ 3$

$\delta_2 = \delta_3\ \delta_4\ \beta\ >\ x\ 0$

$\delta_3 = x$

$\delta_4 = neg\ x$

Figure 3.3-4: Evaluation of (3.3-14)

Thus δ_4 is underlined in line 7.) On line 13 the top of the stack is true, so we select δ_3. (The line corresponding to line 8 is elided after line 13.) It should be clear to the reader that this mechanism has the same effect as that shown in Figure 3.3-1, but that the present mechanism is distinctly more efficient.

Let us look at one more example of evaluation of a conditional in the blackboard mechanism. We consider the PAL program

```
let f x y  =
     (y -> Stem | Stern) x
in
f 'ab' false
```
(3.3-15a)

Here we have used a conditional expression as the rator of a combination. The AE corresponding to this PAL program is

$$(\lambda_1 f.\ f\ \text{'ab'}\ \text{false})\ [\lambda_2 x.\ \lambda_3 y.\ (y \to_4 \text{Stem} \mid_5 \text{Stern})\ x\ ] \qquad (3.3\text{-}15b)$$

and the resultant control sequence is

$$\begin{aligned} &\gamma\ \lambda_1^f\ \lambda_2^x \\ \delta_1 &= \gamma\ \gamma\ f\ \text{'ab'}\ \text{false} \\ \delta_2 &= \lambda_3^y \\ \delta_3 &= \gamma\ \delta_4\ \delta_5\ \beta\ y\ x \\ \delta_4 &= \text{Stem} \\ \delta_5 &= \text{Stern} \end{aligned} \qquad (3.3\text{-}15c)$$

Evaluation of this control sequence is shown in Figure 3.3-5.

Control	Stack	Environment
$E_0\ \gamma\ \lambda_1^f\ \lambda_2^x$	E_0	
$E_0\ \gamma$	$^0\lambda_1^f\ ^0\lambda_2^x\ E_0$	
$E_1\ \gamma\ \gamma$ f 'ab' false	E_1	1: f = $^0\lambda_2^x$ (0)
γ	$^0\lambda_2^x$ 'ab' false	
$\gamma\ E_2\ \lambda_3^y$	E_2	2: x = 'ab' (0)
$E_1\ \gamma$	$^2\lambda_3^y$ false E_1	
$E_3\ \gamma\ \delta_4\ \delta_5\ \beta\ y\ x$	E_3	3: y = false (2)
$\delta_4\ \delta_5\ \beta$	false 'ab'	
γ Stern	'ab'	
$E_3\ \gamma$	Stern 'ab' E_3	
$\vdash$	'b'	

Figure 3.3-5: Evaluation of (3.3-15)

Summary: Which of the two methods to select for handling conditionals in the blackboard evaluator is primarily a matter of objective. To prove results such as the Church-Rosser theorem it is advantageous to minimize the diversity of constructs and hence the complexity of case analysis, so the first method should be selected. Method two, in addition to the demonstrated gain in

efficiency, has also the advantage of capturing clearly the essence of conditionals in programming: the selection of one of two ways to proceed. We thus use method two in all subsequent evaluations, and we base the gedanken evaluator on it in section 3.5.

3.4 Recursion

We have already remarked in conjunction with (3.2-15) that if Z is the fixed-point generating λ-expression defined by

$$Z \equiv \lambda G.\ (\lambda u.\ u\ u)[\lambda v.\ G\ (v\ v)] \tag{3.4-1}$$

then no expression of the form

$$Z\ F\ a1\ a2\ \ldots\ ak \tag{3.4-2}$$

terminates under blackboard evaluation. Thus the decision to gain efficiency by departing from normal order forces us to devise alternate methods for accommodating recursion.

As we did in the previous section, we investigate here two approaches. The first approach involves exploiting our ability to control order of evaluation: By inserting semantically-irrelevant λ's, we can modify the definition of Z in such a way that (3.4-2) does terminate for certain important special cases of expressions E. The second approach is derived from the fundamental identity

$$Y\ F \simeq F(Y\ F) \tag{3.4-3}$$

and involves making changes to the blackboard evaluator itself. Neither approach, however, succeeds in regaining the full semantic power of the λ-calculus forfeited by departure from normal order.

Applicative Modification of Y

In order to see how (3.4-1) can be modified to yield termination of expressions such as (3.4-2), we show in Figure 3.4-1 part of the evaluation of the AE

$$(\lambda u.\ u\ u)\ (\lambda v.\ F(v\ v)) \tag{3.4-4a}$$

and hence of the control structure

$$\begin{aligned} &\gamma\ \lambda_1^u\ \lambda_2^v \\ &\delta_1 = \gamma\ u\ u \\ &\delta_2 = \gamma\ F\ \gamma\ v\ v \end{aligned} \tag{3.4-4b}$$

in an environment in which E is known. It is clear from lines 6 and 8 that the evaluation is in a loop and can never terminate. Now consider normal order evaluation of Z E, for the Z of (3.4-1) and assuming that E abbreviates some -expression. We have

$$\begin{aligned} Z\ F &\equiv [\lambda G.\ (\lambda u.\ u\ u)\ (\lambda v.\ G(v\ v))]\ F \\ &\xrightarrow{\beta} (\lambda u.\ u\ u)\ (\lambda v.\ F(v\ v)) \\ &\xrightarrow{\beta} [\lambda v.\ F(v\ v)]\ [\lambda v.\ F(v\ v)] \\ &\xrightarrow{\beta} F\ \{[\lambda v.\ F(v\ v)]\ [\lambda v.\ F(v\ v)]\} \\ &\simeq \ldots \end{aligned} \tag{3.4-5}$$

	Control	Stack	Environment
1	E_0 ⎵⎵	E_0	0: PE
2	$E_1\ \gamma\ \lambda_1^u\ \lambda_2^v$	E_1	1: F = ⎵⎵ (0)
3	$E_1\ \gamma$	$'\lambda_1^u\ '\lambda_2^v\ E_1$	
4	$E_2\ \gamma\ u\ u$	E_2	2: u = $'\lambda_2^v$ (1)
5	$E_2\ \gamma$	$'\lambda_2^v\ '\lambda_2^v\ E_2$	
6	$E_3\ \gamma\ F\ \gamma\ v\ v$	E_3	3: v = $'\lambda_2^v$ (1)
7	γ	$'\lambda_2^v\ '\lambda_2^v\ E_3$	
8	$E_4\ \gamma\ F\ \gamma\ v\ v$	E_4	4: v = $'\lambda_2^v$ (1)

Figure 3.4-1: Evaluation of (3.4-4).

The two evaluations are analogous up to the point where a rand is a combination. From this point on, however, the two procedures diverge: In normal order E is applied to the unevaluated rand in braces, whereas the blackboard machine attempts to evaluate the rand and becomes trapped in the loop

$$\begin{aligned} & F\ \{[\lambda v.\ F\ (v\ v)][\lambda v.\ F\ (v\ v)]\} \\ \xrightarrow{\beta}\ & F\ \{F\ \{[\lambda v.\ F\ (v\ v)][\lambda v.\ F\ (v\ v)]\}\} \\ & \cdots \end{aligned} \tag{3.4-6}$$

Looping is avoided if in lieu of Z we use

$$Z' \equiv \lambda G.\ (\lambda u.\ u\ u)\ [\lambda v.\ G(\lambda x.\ v\ v\ x)] \tag{3.4-7}$$

as the fixed point operator, since then

$$\begin{aligned} Z'\ F & \xrightarrow{\beta} (\lambda u.\ u\ u)\ [\lambda v.\ F(\lambda x.\ v\ v\ x)] \\ & \xrightarrow{\beta} [\lambda v.\ F(\lambda x.\ v\ v\ x)][\lambda v.\ F(\lambda x.\ v\ v\ x)] \\ & \xrightarrow{\beta} F\{\lambda x.\ [\lambda v.\ F(\lambda x.\ v\ v\ x)][\lambda v.\ F(\lambda x.\ v\ v\ x)]\ x\} \end{aligned} \tag{3.4-8}$$

Here the loop-inducing combination is tucked away as the body of $\{\lambda x.$ ⎵⎵$\}$. Since evaluation of a λ-body is deferred until application of the closure of which it is a part, the next step in blackboard evaluation of (3.4-8) is application of E to $\{\lambda x.$ ⎵⎵$\}$, which again accords with normal order.

The semantic irrelevance of replacing (v v) in (3.4-1) by (λx. v v x) in (3.4-7) follows from the observation that

$$(\lambda x.\ v\ v\ x)\ Q \xrightarrow{\beta} v\ v\ Q \tag{3.4-9}$$

for every applicative expression Q. Thus the effect of the replacement is solely to modify the order in which subexpressions are evaluated. Some presentations of the λ-calculus use

$$\lambda x.\ M\ x \xrightarrow{\eta} M \tag{3.4-10}$$

as another axiom, called η-conversion, in addition to the ones presented in Section 2.3. Equation (3.4-9) is of course a special case of this axiom.

The function Z' defined in (3.4-7) suffices to handle many important instances of recursion, but it suffers from two disadvantages: First, its use in blackboard evaluation is tediously inefficient; and second, as we see later, there are certain constructs which still cannot be accommodated.

To see both that Z' works and also just how inefficient it is, we consider the PAL program

```
let rec f n = n eq 0 -> 1 | n * f(n-1)
in                                          (3.4-11)
f 3
```

and hence the AE

$$(\lambda_1 f.\ f\ 3)\ \{\ [\lambda_2 G.\ (\lambda_3 u.\ u\ u)(\lambda_4 v.\ G(\lambda_5 x.\ v\ v\ x))]\ [\lambda_6 f.\ \lambda_7 n.\ n\ \text{eq}\ 0 \rightarrow_8 1 \mid_9 n*f(n-1)]\ \} \qquad (3.4\text{-}12a)$$

and hence the control structure

$$\begin{aligned}
&\gamma\ \lambda_1^f\ \gamma\ \lambda_2^G\ \lambda_6^f \\
\delta_1 &= \gamma\ f\ 3 \\
\delta_2 &= \gamma\ \lambda_3^u\ \lambda_4^v \\
\delta_3 &= \gamma\ u\ u \\
\delta_4 &= \gamma\ G\ \lambda_5^x \\
\delta_5 &= \gamma\ \gamma\ v\ v\ x \\
\delta_6 &= \lambda_7^n \\
\delta_7 &= \delta_8\ \delta_9\ \beta\ \text{eq}\ n\ 0 \\
\delta_8 &= 1 \\
\delta_9 &= *\ n\ \gamma\ f\ -\ n\ 1
\end{aligned} \qquad (3.4\text{-}12b)$$

The first part of the blackboard evaluation of this control is shown in Figure 3.4-2. Lines 1 through 9 serve to set things up in preparation for the recursion, and lines 18 to 23 lead from one recursive step to the next. It is clear that the overhead is high and that a blackboard method for recursion that reduces or eliminates this overhead is desirable. The method about to be presented reduces the overhead noticeably, and we see later in this section a method that reduces it still further.

The Y - η Method: We saw in section 2.4 (in (2.4-12d) on page 2.4-84, for example) that AE's involving Y can be evaluated by β-reduction and ρ-reduction, the latter being derived from the fundamental identity

$$Y\ F \simeq F(Y\ F) \qquad (3.4\text{-}13)$$

The idea implicit in ρ-reduction is that, although we do not know the value produced by applying Y to E, we do know that it is the same as E applied to that value. That is, we know that the value is a fixed point of E. We exploit this idea now to do recursion in the blackboard evaluator. We consider again the PAL program (3.4-11), but now we regard it as sugaring for the AE

$$(\lambda_1 f.\ f\ 3)\ [Y\ (\lambda_2 f.\ \lambda_3 n.\ n\ \text{eq}\ 0 \rightarrow_4 1 \mid_5 n*f(n-1))] \qquad (3.4\text{-}14a)$$

	Control	Stack	Environment
1	$E_0\ \gamma\ \lambda_1^f\ \gamma\ \lambda_2^G\ \lambda_6^f$	E_0	0: PE
2	γ	${}^0\lambda_2^G\ {}^0\lambda_6^f\ E_0$	
3	$E_1\ \gamma\ \lambda_3^u\ \lambda_4^v$	E_1	1: $G = {}^0\lambda_6^f$ (0)
4	$E_1\ \gamma$	${}^1\lambda_3^u\ {}^1\lambda_4^v\ E_1$	
5	$E_2\ \gamma\ u\ u$	E_2	2: $u = {}^1\lambda_4^v$ (1)
6	$E_2\ \gamma$	${}^1\lambda_4^v\ {}^1\lambda_4^v\ E_2$	
7	$E_3\ \gamma\ G\ \lambda_5^x$	E_3	3: $v = {}^1\lambda_4^v$ (1)
8	$E_3\ \gamma$	${}^0\lambda_6^f\ {}^3\lambda_5^x\ E_3$	
9	$\gamma\ \lambda_1^f\ E_4\ \lambda_7^n$	E_4	4: $f = {}^3\lambda_5^x$ (0)
10	$E_0\ \gamma$	${}^0\lambda_1^f\ {}^4\lambda_7^n\ E_0$	
11	$E_5\ \gamma\ f\ 3$	E_5	5: $f = {}^4\lambda_7^n$ (0)
12	$E_5\ \gamma$	${}^4\lambda_7^n\ \underline{\underline{3}}\ E_5$	
13	$E_6\ \delta_8\ \delta_9\ \beta\ eq\ n\ 0$	E_6	6: $n = \underline{\underline{3}}$ (4)
14	$\beta\ eq$	$\underline{\underline{3}}\ \underline{\underline{0}}\ E_6$	
15	$E_6\ \delta_8\ \underline{\delta_9}\ \beta$	$\underline{\underline{\text{false}}}\ E_6$	
16	$E_6\ *\ n\ \gamma\ f\ -\ n\ 1$	E_6	
17	$\gamma\ f\ -$	$\underline{\underline{3}}\ \underline{\underline{1}}\ E_6$	
18	γ	${}^3\lambda_5^x\ \underline{\underline{2}}$	
19	$E_7\ \gamma\ \gamma\ v\ v\ x$	E_7	7: $x = \underline{\underline{2}}$ (3)
20	γ	${}^1\lambda_4^v\ {}^1\lambda_4^v\ \underline{\underline{2}}\ E_7$	
21	$E_8\ \gamma\ G\ \lambda_5^x$	E_8	8: $v = {}^1\lambda_4^v$ (1)
22	$E_8\ \gamma$	${}^0\lambda_6^f\ {}^8\lambda_5^x\ E_8$	
23	$\gamma\ E_9\ \lambda_7^n$	E_9	9: $f = {}^8\lambda_5^x$ (0)
24	$E_7\ \gamma$	${}^9\lambda_7^n\ \underline{\underline{2}}\ E_7$	
25	$E_6\ *\ n\ E_{10}\ \delta_8\ \delta_9\ \beta\ eq\ n\ 0$	$E_{10}\ E_6$	10: $n = \underline{\underline{2}}$ (9)

Figure 3.4-2: Partial Evaluation of (3.4-12)

and hence the control structure

$$\begin{aligned} &\gamma\ \lambda_1^f\ \gamma\ Y\ \lambda_2^f \\ \delta_1 &= \gamma\ f\ 3 \\ \delta_2 &= \lambda_3^n \\ \delta_3 &= \delta_4\ \delta_5\ \beta\ \text{eq n 0} \\ \delta_4 &= 1 \\ \delta_5 &= *\ n\ \gamma\ f\ -\ n\ 1 \end{aligned} \qquad (3.4\text{-}14b)$$

Just as we introduced $\underline{Q}$ as a special identifier in section 3.3, so now we introduce $\underline{Y}$. Blackboard evaluation of (3.4-14b) proceeds to

$$\ldots\ \gamma \qquad \underline{Y}\ {}^{\circ}\lambda_2^f\ \ldots \qquad (3.4\text{-}15a)$$

and we are faced with the problem of deciding what $\underline{Y}$ is to do when applied to a closure. Presumably the next step leaves some value in the stack, so we give that value a name: η. Now let $\underline{F}$ abbreviate ${}^{\circ}\lambda_2^f$. Since we have said that the value of $\underline{Y}\ \underline{F}$ is η, we can substitute η for $\underline{Y}\ \underline{F}$ in (3.4-13) to get

$$\eta \simeq F\ \eta$$

That is, the unknown value η produced by applying $\underline{Y}$ to the closure ${}^{\circ}\lambda_2^f$ is equivalent to the value that would be produced by applying ${}^{\circ}\lambda_2^f$ to η. Thus we decree that the next blackboard line after (3.4-15a) is to be

$$\ldots\ \gamma \qquad {}^{\circ}\lambda_2^f\ \eta\ \ldots \qquad (3.4\text{-}15b)$$

and we make a note that η is an abbreviation for the result of applying $\underline{Y}$ to ${}^{\circ}\lambda_2^f$. The rest of the evaluation is shown in Figure 3.4-3. In going from line 3 to line 4, we associate $\underline{f}$ in the environment with the as-yet-unknown quantity denoted by η. We make a note in the extra column at the right as to what η is an abbreviation for. (In a more complicated evaluation involving several recursive functions, we could use subscripted η's, one for each recursive function being defined.)

On line 13 is the first need for the value of η. Although we do not know this value we do know what it is an abbreviation for, so on line 14 we have replaced η by the application of $\underline{Y}$ to ${}^{\circ}\lambda_2^f$. Thus line 14 repeats the situation of line 2, and the evaluation proceeds. On line 33 the recursion terminates and starts to unwind. Note that each of the three "n"s in the control in line 33 is eventually looked up in a different environment, a fact that is essential to make the evaluation come out correctly.

Note the improvement in efficiency over the evaluation in Figure 3.4-2: Initialization here takes 4 lines, whereas it took 9 lines in the earlier method. The first recursive call used four lines for overhead (lines 13 to 16) and the other two each use three, since the line corresponding to line 14 is elided. Thus the overhead of a recursion has been reduced from six lines to three. No very solid quantitative conclusions on efficiency improvements can be inferred from this argument, since counting lines in blackboard evaluation is rather meaningless. It is safe though to conclude that efficiency <u>has</u> been improved.

	Control	Stack	Environment	Defs
1	$E_0\ \gamma\ \lambda_1^f\ \gamma\ Y\ \lambda_2^f$	E_0	0: PE	
2	γ	$\underline{Y}\ {}^{\circ}\lambda_2^f\ E_0$		
3	γ	${}^{\circ}\lambda_2^f\ \eta$		$\eta \sim \underline{Y}\ {}^{\circ}\lambda_2^f$
4	$\gamma\ \lambda_1^f\ E_1\ \lambda_3^\eta$	E_1	1: f = η (0)	
5	$E_0\ \gamma$	${}^{\circ}\lambda_1^f\ {}'\lambda_3^\eta\ E_0$		
6	$E_2\ \gamma\ f\ 3$	E_2	2: f = ${}'\lambda_3^\eta$ (0)	
7	$E_2\ \gamma$	${}'\lambda_3^\eta\ \underline{3}\ E_2$		
8	$E_3\ \delta_4\ \delta_5\ \beta\ eq\ \eta\ 0$	E_3	3: η = $\underline{3}$ (1)	
9	$\beta\ eq$	$\underline{3}\ \underline{0}\ E_3$		
10	$E_3\ \delta_4\ \underline{\delta_5}\ \beta$	false E_3		
11	$E_3\ *\ \eta\ \gamma\ f\ -\ \eta\ 1$	E_3		
12	$\gamma\ f\ -$	$\underline{3}\ \underline{1}\ E_3$		
13	γ	$\eta\ \underline{2}$		
14	$\gamma\ \gamma$	$Y\ {}^{\circ}\lambda_2^f$		
15	γ	${}^{\circ}\lambda_2^f\ \eta$		
16	$\gamma\ E_4\ \lambda_3^\eta$	E_4	4: f = η (0)	
17	γ	${}'\lambda_3^\eta\ \underline{2}$		
18	$E_5\ \delta_4\ \delta_5\ \beta\ eq\ \eta\ 0$	E_5	5: η = $\underline{2}$ (4)	
19	$\beta\ eq$	$\underline{2}\ \underline{0}\ E_5$		
20	$E_5\ \delta_4\ \underline{\delta_5}\ \beta$	false E_5		
21	$E_5\ *\ \eta\ \gamma\ f\ -\ \eta\ 1$	E_5		
22	γ	$\eta\ \underline{1}\ E_5$		
23	$\gamma\ \gamma$	${}^{\circ}\lambda_2^f\ \eta$		
24	$\gamma\ E_6\ \lambda_3^\eta$	E_6	6: f = η (0)	
25	γ	${}^{6}\lambda_3^\eta\ \underline{1}$		
26	$E_7\ \delta_4\ \underline{\delta_5}\ \beta\ eq\ \eta\ 0$	E_7	7: η = $\underline{1}$ (6)	
27	$E_7\ *\ \eta\ \gamma\ f\ -\ \eta\ 1$	E_7		
28	γ	$\eta\ \underline{0}\ E_7$		
29	$\gamma\ \gamma$	${}^{\circ}\lambda_2^f\ \eta$		
30	$\gamma\ E_8\ \lambda_3^\eta$	E_8	8: f = η (0)	
31	γ	${}^{8}\lambda_3^\eta\ \underline{0}$		
32	$E_9\ \underline{\delta_4}\ \delta_5\ \beta\ eq\ \eta\ 0$	E_9	9: η = $\underline{0}$ (8)	
33	$E_3\ *\ \eta\ E_5\ *\ \eta\ E_7\ *\ \eta\ E_9\ 1$	$E_9\ E_7\ E_5\ E_3$		
34	$E_7\ *\ \eta$	$\underline{1}\ E_7$		
35	$\eta\ E_7\ *$	$\underline{1}\ \underline{1}\ E_7$		
36	$E_5\ *\ \eta$	$\underline{1}\ E_5$		
37	$\eta\ E_5\ *$	$\underline{2}\ \underline{1}\ E_5$		
38	$E_3\ *\ \eta$	$\underline{2}\ E_3$		
39	$E_3\ *$	$\underline{3}\ \underline{2}\ E_3$		
40	$\vdash$	$\underline{6}$		

The Nature of the Problem

We have seen that building a new mechanism into the blackboard evaluator permits us to reduce noticeably the overhead for recursion, both for the initialization and for each recursive call. Since recursion is so important in our study of programming languages, we would like to reduce this overhead further if possible. While the next improvement to be made requires use of the imperative features of PAL and we must wait for Chapter 4 to see its details, it is appropriate here to investigate further the nature of the problem.

Insight may be achieved by study of evaluation of the non-recursive PAL program

```
let f n = n eq 0 -> 1 | n*f(n-1)
in                                        (3.4-16a)
f 2
```

and the corresponding AE

$$(\lambda_1 f.\ f\ 2)\ [\lambda_2 n.\ n\ \text{eq}\ 0 \rightarrow_3 1 \mid_4 n*f(n-1)] \qquad (3.4\text{-}16b)$$

and the control sequence

$$\begin{aligned} &\gamma\ \lambda_1^f\ \lambda_2^n \\ &\delta_1 = \gamma\ f\ 2 \\ &\delta_2 = \delta_3\ \delta_4\ \beta\ \text{eq}\ n\ 0 \\ &\delta_3 = 1 \\ &\delta_4 = *\ n\ \gamma\ f\ -\ n\ 1 \end{aligned} \qquad (3.4\text{-}16c)$$

Blackboard evaluation of (3.4-16c) relative to an environment in which the value of f is the function Succ is carried out (with many uninteresting steps elided)

	Control	Stack	Environment
1	E_0 ⋯	⋯ E_0	0: PE
2	$E_1\ \gamma\ \lambda_1^f\ \lambda_2^n$	E_1	1: f = Succ (0)
3	$E_1\ \gamma$	$'\lambda_1^f\ '\lambda_2^n\ E_1$	
4	$E_2\ \gamma\ f\ 2$	E_2	2: f = $'\lambda_2^n$ (1)
5	$E_2\ \gamma$	$'\lambda_2^n\ \underline{2}\ E_2$	
6	$E_3\ \delta_3\ \delta_4\ \beta\ \text{eq}\ n\ 0$	E_3	3: n = $\underline{2}$ (1)
7	$E_3\ *\ n\ \gamma\ f\ -\ n\ 1$	E_3	
8	$*\ n\ \gamma$	Succ $\underline{1}$ E_3	
9	$E_3\ *$	$\underline{2}\ \underline{2}$	
10	⊢	$\underline{4}$	

Figure 3.4-4: Evaluation of (3.4-16)

in Figure 3.4-4. Note that the value coupled with f when environment layer 2 is laid down is the λ-closure $'\lambda_2^n$. It follows that application of f (the transition from line 5 to line 6) creates a new environment layer (labelled 3)

which is appended to layer 1. Accordingly, the body δ_2 of $^1\lambda^n_2$ is evaluated in an environment in which the value of f is determined by environment layer 1. Thus the f in δ_2 references Succ.

In order to effect a self-referential definition of f, it is sufficient that matters be altered so that the body of λ^n_2 is evaluated instead relative to an environment in which the value of f is determined by the (new) definition of environment layer 2. This would be accomplished if, somehow, the value of f in environment layer 2 were $^2\lambda^n_2$ instead of $^1\lambda^n_2$. Then the application of f in line 5 would result in environment layer 3 being linked to 2 instead of to 1, and then the lookup of f in line 7 would result in a recursive call instead of a call to Succ. To see this more clearly, consider evaluation of the PAL program (3.4-11) using the desugaring (3.4-14a). Since we plan to use a method other than the Y-η method, we replace Y by Y" to emphasize the difference. Thus we get the control structure

$$
\begin{array}{l}
\gamma\ \lambda^f_1\ \gamma\ Y''\ \lambda^f_2 \\
\delta_1 = \gamma\ f\ 3 \\
\delta_2 = \lambda^n_3 \\
\delta_3 = \delta_4\ \delta_5\ \beta\ \text{eq n 0} \\
\delta_4 = 1 \\
\delta_5 = *\ n\ \gamma\ f - n\ 1
\end{array}
\qquad (3.4\text{-}17)
$$

Examine now blackboard evaluation of this control structure in Figure 3.4-5. Assume for the moment that application of Y" to $^1\lambda^f_2$ on line 2, the details of which are elided, leaves the result shown on line 4. The key fact is that the λ-closure which is the value of f in environment layer 2 is linked to layer 2, rather than to layer 1 as it was in Figure 3.4-4. In consequence, initial application of f (the transition from line 8 to line 9) creates new environment layer 4 which is appended to layer 2 rather than to layer 1. It follows that any previous definition of f (such as Succ) is superceded by the new definition when f is encountered within the body of λ^n_3, as in line 11. The same effect recurs until the premise "n eq 0" is true, at which point the procedure unwinds. Thus evaluation of (3.4-17) produces the factorial of 3. Note particularly that there was no overhead at all in successive steps of the recursion, and that the initial overhead is in the elided steps. Clearly we have met our objective of improved efficiency, provided only that we can specify a Y" that does what is wanted.

The Function Y": We now explain in detail a blackboard mechanism for Y" that works as suggested above. Although the mechanism is rather ad hoc and not very well grounded in theory, it is worth studying because it captures the essence of what is needed to do recursion. We defer until Chapter 4 presentation of a PAL program that corresponds to Y", since such a program requires use of the assignment statement, a PAL language feature whose explanation requires concepts which we have yet to see.

We proceed as follows: We assume that the ob that is the value of Y" is written (in the stack) as Y". Because of the nature of the desugaring rules, Y"

	Control	Stack	Environment
1	$E_1\ \gamma\ \lambda_1^f\ \gamma\ Y^*\ \lambda_2^f$	E_1	1: f = Succ (0)
2	γ	$Y^*\ \lambda_2^f\ E_1$	
3	$E_2\ \delta_2$	E_x	
4	$\lambda_1^f\ E_2$	${}^2\lambda_3^n\ E_2$	2: f = ${}^2\lambda_3^n$ (1)
5	$\gamma\ \lambda_1^f$	${}^2\lambda_3^n$	
6	$E_1\ \gamma$	${}^0\lambda_1^f\ {}^2\lambda_3^n\ E_1$	
7	$E_3\ \gamma\ f\ 3$	E_3	3: f = ${}^2\lambda_3^n$ (0)
8	$E_3\ \gamma$	${}^2\lambda_3^n\ \underline{3}\ E_3$	
9	$E_4\ \delta_4\ \underline{\delta_5}\ \beta\ \underline{=_0}\ n\ 0$	E_4	4: n = $\underline{3}$ (2)
10	$E_4\ *\ n\ \gamma\ f\ -\ n\ 1$	E_4	
11	γ	${}^2\lambda_3^n\ \underline{2}\ E_4$	
12	$E_5\ \delta_4\ \underline{\delta_5}\ \beta\ \underline{=_0}\ n\ 0$	E_5	5: n = $\underline{2}$ (2)
13	$E_5\ *\ n\ \gamma\ f\ -\ n\ 1$	E_5	
14	γ	${}^2\lambda_3^n\ \underline{1}\ E_5$	
15	$E_6\ \delta_4\ \underline{\delta_5}\ \beta\ \underline{=_0}\ n\ 1$	E_6	6: n = $\underline{1}$ (2)
16	$E_6\ *\ n\ \gamma\ f\ -\ n\ 1$	E_6	
17	γ	${}^2\lambda_3^n\ \underline{0}\ E_6$	
18	$E_7\ \underline{\delta_4}\ \delta_5\ \beta\ \underline{=_0}\ n\ 1$	E_7	
19	$E_4\ *\ n\ E_5\ *\ n\ E_6\ *\ n\ E_7\ 1$	$E_7\ E_6\ E_5\ E_4$	
20	$E_5\ *\ n\ E_6\ *$	$\underline{1}\ \underline{1}\ E_6$	
21	$*\ n\ E_5\ *$	$\underline{2}\ \underline{1}\ E_5$	
22	$E_4\ *$	$\underline{3}\ \underline{2}\ E_4$	
23	$\vdash$	$\underline{6}$	

Figure 3.4-5: Evaluation of (3.4-17).

can be used only as a rator in a combination whose rand is a λ-closure. Suppose then that we encounter

$$\ldots\ \gamma \qquad \underline{Y''}\ {}^{b}\lambda_i^f\ \ldots \tag{3.4-18}$$

in environment $\underline{a}$. We again exploit the fundamental identity. Study carefully the following steps:

(1) We want to know the value produced by applying $\underline{Y''}$ to ${}^{b}\lambda_i^f$. Call that value $\underline{\theta}$, and let $\underline{F}$ abbreviate ${}^{b}\lambda_i^f$.

(2) Since $\underline{Y''}$ is a fixed point operator, it must satisfy the fundamental identity.

(3) Since $\underline{\theta}$ is $\underline{Y''\ F}$ it follows from (2) that $\underline{\theta} \simeq \underline{F\ \theta}$.

(4) Therefore we can evaluate $\underline{\theta}$ by evaluating $\underline{F\ \theta}$. That is, we can proceed from (3.4-18) by applying ${}^{b}\lambda_i^f$ to $\underline{\theta}$.

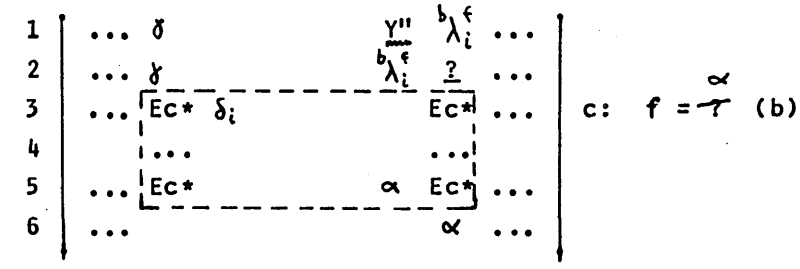
(5) We would like to use the blackboard evaluator to do this, but we can do so only if we know $\underline{\theta}$. But that is what we wanted in the first place.

(6) Not knowing the value of $\underline{\theta}$, we assume temporarily a value for it which we call $\underline{?}$. When we eventually get a value for $\underline{\theta}$, we will replace the $\underline{?}$ by that value.

(7) Thus we replace the application of $\underline{Y''}$ to ${}^{b}\lambda_i^f$ in (3.4-18) by application of ${}^{b}\lambda_i^f$ to $\underline{?}$. This leads to the subproblem of evaluating δ_i in an environment in which $\underline{f}$ is associated with $\underline{?}$. Call that environment $\underline{c}$.

(8) After a while the subproblem will terminate with a value in the stack: Call it $\underline{\alpha}$. Since $\underline{\alpha}$ is the value of $\underline{F\ \theta}$, it follows from (3) that this is the value we wanted and is what we should have used instead of $\underline{?}$. Thus in environment $\underline{c}$ we replace the $\underline{?}$ by $\underline{\alpha}$.

All of this can be summarized:



(3.4-19)

There are several points to note about this:

- The $\underline{?}$ in environment layer $\underline{c}$ is shown as being crossed out and replaced by $\underline{\alpha}$. This is done when the evaluation reaches line 5.
- The environment markers for layer $\underline{c}$ are shown as $\underline{Ec*}$ instead of $\underline{Ec}$ as usual. The purpose is to serve notice that something out of the

ordinary is to be done on completion of the subproblem; specifically, that the variable in layer c is to be changed.

- The ? as the value of f in the subproblem serves notice that the value of f should not be used in the course of the subproblem.
- Evaluation of the identifier Y'' produces the ob Y'' in the stack.

The method is clearly ad hoc and has an air of magic about it. We present it at this time because we want next to compare the Y's we have seen with one another, and we need an exposition of each to do so. We have yet to give any argument that the method works. In fact it frequently does, and there are several examples in the rest of this section of its use. The PAL implementation uses a method very similar to this one.

Comparison of Y's

We have now discussed four different ways to handle recursion in blackboard evaluation:

- normal order evaluation, using the axioms of section 2.3
- the λ-expression Z', defined by (3.4-7)
- the Y-M method
- the function Y'' just described

We have already compared these methods in one way, having observed that each is noticeably more efficient than the previous one. We now want to consider situations in which, in a sense to be described, the methods "don't work". Consideration of PAL's within construction leads nicely to such problems.

As mentioned on page 3.2-121, the "within" construct plays an important role vis-a-vis computational efficiency. The role becomes particularly significant in connection with recursive functions. The PAL expression

$$\text{let rec } (c = P \text{ within } g\ n = Q) \text{ in } M \qquad (3.4\text{-}20)$$

exemplifies the definition of a recursive function g involving an own variable c. (Note that PAL's syntax requires the parentheses, since without them the parse would be different.) We desugar, first reducing the definition to the standard form

$$\text{let rec } g = (\lambda c.\ \lambda n.\ Q)\ P \text{ in } M \qquad (3.4\text{-}21)$$

and then reducing to the equivalent AE

$$(\lambda_1 g.\ M)\ \{Y\ [\lambda_2 g.\ (\lambda_2 c.\ \lambda_4 n.\ Q)\ P]\} \qquad (3.4\text{-}22)$$

Using Y'' for Y and (g 3) for M in (3.4-20) leads to the control structure

$$\begin{aligned} &\gamma\ \lambda_1^9\ \gamma\ Y''\ \lambda_2^9 \\ &\delta_1 = \gamma\ g\ 3 \\ &\delta_2 = \gamma\ \lambda_3^c\ P \\ &\delta_3 = \lambda_4^n \\ &\delta_4 = Q \end{aligned} \qquad (3.4\text{-}23)$$

Evaluation of this control is shown in Figure 3.4-6. By line 10 we are about to

	Control	Stack	Environment
1	$E_0\ \gamma\ \lambda_1^g\ \gamma\ Y''\ \lambda_2^g$	E_0	0: PE
2	γ	$Y''^0\lambda_2^g\ E_0$	$^2\lambda_4^n$
3	$E_1^*\ \gamma\ \lambda_3^c\ P$	E_1^*	1: g = ? (0)
4	$E_1^*\ \gamma$	$^1\lambda_3^c\ P\ E_1^*$	
5	$E_1^*\ E_2\ \lambda_4^n$	E_2	2: c = P (1)
6	$\gamma\ \lambda_1^g\ E_1^*$	$^2\lambda_4^n\ E_1^*$	
7	$E_0\ \gamma$	$^0\lambda_1^g\ ^2\lambda_4^n\ E_0$	
8	$E_3\ \gamma\ g\ 3$	E_3	3: g = $^2\lambda_4^n$ (0)
9	$E_3\ \gamma$	$^2\lambda_4^n\ 3\ E_3$	
10	$E_4\ Q$	E_4	4: n = 3 (2)

Figure 3.4-6: Evaluation of (3.4-23)

evaluate Q, in an environment in which n, c and g are known. Note that if g occurs free in Q (as it presumably does, else why the rec in (3.4-20)?), the value associated with it is the same as the value associated with g in M. P is not again evaluated, just as one would hope with the within construction. (Compare the PAL program

```
let rec g n = (Q where c = P)
```

with (3.4-20). Here P is evaluated each time g is called, either recursively or from M.) Note also that P is evaluated in an environment in which g is known. Thus it would appear that there may be free occurrences of g in P. But may there? This point is discussed in the next subsection.

Limitations of Y": Although Y" is clearly efficient and seems to handle properly programs such as (3.4-20), carefuly study uncovers a semantic deficiency. It seems clear that the scope of g is to include P. (This fact is obvious in (3.4-22), in which P appears in the body of [λg.⋯].) Thus free occurrences of g in P are to denote the recursive function being defined. But in Figure 3.4-6 the evaluation of P takes place at line 4, while g is still associated with ? in environment 1. What value of g can be used?

There is one kind of case in which there is no problem: when c is a function. For example, consider

```
let rec
    (       c k = P
            within
        g n = Q                                    (3.4-24)
    )
in
M
```

Since any free g's in P occur in the body of (λk.⋯⋯), they are not needed until c is applied in Q. By that time g is no longer associated with ?. That is, $c \sim {}^{1}\lambda_2^{K}$ so that when δ_a is evaluated it is in an environment linked to E1, in which g is then properly defined.

To continue with the case where c does not denote a function, consider first the PAL program

```
let rec
    (       c = 120
            within
        f n = n eq 0 -> 1 | n eq 5 -> c | n*f(n-1)     (3.4-25)
    )
in
f 6 + f 7 + f 8
```

This variation on the factorial is designed for greater efficiency than the usual definition for arguments 5 or greater. This is not yet an example of the problem -- there are no problems here with any method for recursion.

Evidently the writer of (3.4-25) knew that the factorial of five is 120. Suppose the function is to be changed, so that the cutoff is 10 rather than 5. Not knowing the factorial of 10 one might try

```
let rec
    (       c = f 10
            within
        f n = n eq 0 -> 1 | n eq 10 -> c | n*f(n-1)    (3.4-26)
    )
in
f 11 + f 12 + f 13
```

This program fails for all methods of doing recursion, since we must apply f to 10 to get a value of c, but doing so requires knowing c. (A proof that (3.4-26) cannot be evaluated by any Y would require only showing that normal order leads to a loop. Why is this sufficient? The reader is advised to carry out the proof.)

There is an obvious minor variation on (3.4-26) which avoids the problem just presented:

```
let rec
    (       c = f 10
            within
        f n = n eq 0 -> 1 | n eq 11 -> 11*c | n*f(n-1)      (3.4-27)
    )
in
f 11 + f 12 + f 13
```

Now there is hope, at least, since f 10 can be evaluated without knowing a value for c. But it is by no means clear which Y, if any, will work.

To find out, we simplify (3.4-27) slightly to an AE that is easier to work with. Let F abbreviate the AE

$$\lambda f.[\lambda c.\lambda n.n \text{ eq } 0 \text{ -> } 1 \mid n \text{ eq } 11 \text{ -> } 11*c \mid n*f(n-1)](f\ 10) \qquad (3.4\text{-}28a)$$

Then we concern ourselves with the AE

$$(\lambda f.\ f\ 11)\ (Y\ F) \qquad (3.4\text{-}28b)$$

We first investigate normal order reduction. We have

$$(\lambda f.\ f\ 11)\ (Y\ F) \xrightarrow{\beta} Y\ F\ 11$$

Let us write first Y F: We have

$$\begin{aligned} &Y\ F \\ &\xrightarrow{\rho} F\ (Y\ F) \\ &\xrightarrow{\beta} [\lambda c.\lambda n.n \text{ eq } 0 \text{ -> } 1 \mid n \text{ eq } 11 \text{ -> } 11*c \mid n*Y\ F\ (n-1)]\ (Y\ F\ 10) \\ &\xrightarrow{\beta} [\lambda n.n \text{ eq } 0 \text{ -> } 1 \mid n \text{ eq } 11 \text{ -> } 11*Y\ F\ 10 \mid n*Y\ F\ (n-1)] \end{aligned}$$

Now for any k such that $k \neq 0$ and $k \neq 11$, we have

$$\begin{aligned} &Y\ F\ k \\ &\simeq k \text{ eq } 0 \text{ -> } 1 \mid k \text{ eq } 11 \text{ -> } 11*Y\ F\ 10 \mid k*Y\ F\ (k-1) \\ &\simeq k*Y\ F\ (k-1) \end{aligned}$$

Also,

$$Y\ F\ 0 \simeq 1$$

Since

$$Y\ F\ 11 \simeq 11 * Y\ F\ 10$$

it follows that Y F is the factorial (as expected) and that there are no problems. Y F k has normal form to all non-negative integers k. (As might be expected, the efficiency gain of the within disappears under normal order.)

We now show that Z', the Y-η method and Y'' all fail to evaluate (3.4-28). Consider first Z'. Replacing Y in (3.4-28b) by the λ-expression (3.4-7), we get the control structure

$$
\begin{array}{l}
\gamma\ \lambda_1^f\ \gamma\ \lambda_9^F\ \lambda_2^f \\
\delta_1 = \gamma\ f\ 11 \\
\delta_2 = \gamma\ \lambda_3^c\ \gamma\ f\ 10 \\
\delta_3 = \lambda_4^n \\
\delta_4 = \delta_5\ \delta_6\ \beta\ \text{eq n 0} \\
\delta_5 = 1 \\
\delta_6 = \delta_7\ \delta_8\ \beta\ \text{eq n 11} \\
\delta_7 = *\ 11\ c \\
\delta_8 = *\ n\ \gamma\ f - n\ 1 \\
\delta_9 = \gamma\ \lambda_{10}^g\ \lambda_{11}^h \\
\delta_{10} = \gamma\ g\ g \\
\delta_{11} = \gamma\ F\ \lambda_{12}^x \\
\delta_{12} = \gamma\ \gamma\ h\ h\ x
\end{array}
\qquad (3.4\text{-}29)
$$

The beginning of an evaluation of this is shown in Figure 3.4-7. Line 13 repeats line 7, and we are in a loop.

	Control	Stack	Environment
1	$E_0\ \gamma\ \lambda_1^f\ \gamma\ \lambda_9^F\ \lambda_2^f$	E_0	0: PE
2	γ	${}^0\lambda_9^F\ {}^0\lambda_2^f\ E_0$	
3	$E_1\ \gamma\ \lambda_{10}^g\ \lambda_{11}^h$	E_1	1: $F = {}^0\lambda_2^f$ (0)
4	$E_1\ \gamma$	${}^1\lambda_{10}^g\ {}^1\lambda_{11}^h\ E_1$	
5	$E_2\ \gamma\ g\ g$	E_2	2: $g = {}^1\lambda_{11}^h$ (1)
6	$E_2\ \gamma$	${}^1\lambda_{11}^h\ {}^1\lambda_{11}^h\ E_2$	
7	$E_3\ \gamma\ F\ \lambda_{12}^x$	E_3	3: $h = {}^1\lambda_{11}^h$ (1)
8	$E_3\ \gamma$	${}^0\lambda_2^f\ {}^3\lambda_{12}^x\ E_3$	
9	$E_4\ \gamma\ \lambda_3^c\ \gamma\ f\ 10$	E_4	4: $f = {}^3\lambda_{12}^x$ (0)
10	γ	${}^3\lambda_{12}^x\ \underline{10}\ E_4$	
11	$E_5\ \gamma\ \gamma\ h\ h\ x$	E_5	5: $x = \underline{10}$ (3)
12	γ	${}^1\lambda_{11}^h\ {}^1\lambda_{11}^h\ \underline{10}\ E_5$	
13	$E_0\ \gamma\ \lambda_1^f\ E_4\ \gamma\ \lambda_3^c\ E_5\ \gamma\ E_6\ \gamma\ F\ \lambda_{12}^h$	$E_6\ \underline{10}\ E_5\ E_4\ E_0$	6: $h = {}^1\lambda_{11}^h$ (1)

Figure 3.4-7: Evaluation of (3.4-29) using the λ-expression Z'.

The Y-Y method, shown in Figure 3.4-8, fares little better. (Here we have replaced λ_9^f on the first line of (3.4-29) by Y, causing the last four lines to be unneeded.) The loop here comes from an attempt to evaluate f 10 to initialize c. The corresponding step in normal order after line 4 would be to apply λ_3^c to the unevaluated rand.

	Control	Stack	Environment	Defs
1	$E_0\ \gamma\ \lambda_1^f\ \gamma\ Y\ \lambda_2^f$	E_0	0: PE	
2	γ	$Y\ {}^o\lambda_2^f\ E_0$		
3	γ	${}^o\lambda_2^f\ \eta$		$\eta \sim {}^o\lambda_2^f$
4	$E_1\ \gamma\ \lambda_3^c\ \gamma\ f\ 10$	E_1	1: f = η (0)	
5	γ	$\eta\ \underline{10}\ E_1$		
6	$\gamma\ \gamma$	${}^o\lambda_2^f\ \eta$		
7	$E_0\ \gamma\ \lambda_1^f\ E_1\ \gamma\ \lambda_3^c\ \gamma\ E_2\ \gamma\ \lambda_3^c\ \gamma\ f\ 10$	$E_2\ \underline{10}\ E_1\ E_0$	2: f = η (0)	

Figure 3.4-8: Evaluation of (3.4-29) by the Y-η Method

Figure 3.4-9 shows evaluation using Y". Here we need the value of f while it is still ?.

	Control	Stack	Environment
1	$E_0\ \gamma\ \lambda_1^f\ \gamma\ Y''\ \lambda_2^f$	E_0	0: PE
2	γ	$Y''\ {}^o\lambda_2^f\ E_0$	
3	γ	${}^o\lambda_2^f\ ?$	
4	$E_1\ \gamma\ \lambda_3^c\ \gamma\ f\ 10$	E_1	1: f = ? (0)
5	$E_0\ \gamma\ \lambda_1^f\ E_1\ \gamma\ \lambda_3^c\ \gamma$	$?\ \underline{10}\ E_1\ E_0$	

Figure 3.4-9: Evaluation of (3.4-29) using Y".

We can draw two conclusions: First, although we can evaluate (3.4-28) in normal order, each deviation from normal order which we have tried leads to failure. Second, we developed the Y-η method and then Y" to improve efficiency, and we do see that each of these revealed the problem in fewer steps than did the previous one. One final comment: The PAL implementation, which is modelled closely after Y", also fails on this program.

A Family of Y's: A natural question is to ask whether there is any systematic departure from normal order that terminates for (3.4-28). It turns out that there is, and that there are some interesting observations to be made in studying the problem. We started this section by observing that if Z is defined by

$$\lambda F.\ (\lambda u.\ u\ u)\ (\lambda v.\ F\ (v\ v)) \qquad (3.4\text{-}30)$$

then no expression of the form

$$Z\ F\ a1\ a2\ \ldots \qquad (3.4\text{-}31)$$

terminates under blackboard evaluation. We derived

$$Z' \equiv \lambda F.\ (\lambda u.\ u\ u)\ (\lambda v.\ F\ (\lambda x.\ v\ v\ x)) \qquad (3.4\text{-}32)$$

and showed that, at least for some AE's E,

Z' F a1 a2 ...

terminates in blackboard order. Clearly the function F in (3.4-28b) is not such an F.

We derived Z' from Z by observing that the problem in order of evaluation could be circumvented by adding semantically irrelevant λ's. An obvious question now is whether we can find a modification of Z' -- say Z'' -- that works for (3.4-28). To aid our thinking, we consider the simpler PAL program

```
let rec
    (       c = g 0
            within
        g n = n eq 0 -> 1 | n+c                    (3.4-33)
    )
in
g 5
```

since this rather silly program captures the essence of the problem of (3.4-28) and is easier to work with. The corresponding AE is

$$(\lambda g.\ g\ 5)\ \{Y\ [\lambda g.\ (\lambda c.\ \ n.\ n\ eq\ 0\ \text{->}\ 1\ |\ n+c)\ (g\ 0)]\} \qquad (3.4\text{-}34)$$

and we are interested in a λ-expression for Y such that evaluation of this AE terminates in blackboard order. To gain insight, we look more carefully at what fails for Z'. We define the following abbreviations:

$$\begin{aligned} F &\equiv \lambda g.\ (\lambda c.\ \lambda n.\ n\ eq\ 0\ \text{->}\ 1\ |\ n+c)\ (g\ 0) \\ \lambda_1 &\equiv \lambda v.\ F(\lambda x.\ v\ v\ x) \\ \lambda_2 &\equiv \lambda x.\ \lambda_1\ \lambda_1\ x \end{aligned} \qquad (3.4\text{-}35)$$

Then we are concerned with the AE

$$[\lambda G.\ (\lambda u.\ u\ u)\ (\lambda v.\ G(\lambda x.\ v\ v\ x))]\ F\ 5 \qquad (3.4\text{-}36)$$

At first the evaluation proceeds identically in normal order and blackboard order:

$$\begin{aligned} & Z'\ F\ 5 \\ \equiv\ & [\lambda G.\ (\lambda u.\ u\ u)\ (\lambda v.\ G(\lambda x.\ v\ v\ x))]\ F\ 5 \\ \xrightarrow{\beta}\ & (\lambda u.\ u\ u)\ (\lambda v.\ F(\lambda x.\ v\ v\ x))\ 5 \\ \xrightarrow{\beta}\ & [\lambda v.\ F(\lambda x.\ v\ v\ x)]\ [\lambda v.\ F(\lambda x.\ v\ v\ x)]\ 5 \\ \xrightarrow{\beta}\ & F\ (\lambda x.\ \lambda_1\ \lambda_1\ x)\ 5 \\ \equiv\ & F\ \lambda_2\ 5 \\ \xrightarrow{\beta}\ & (\lambda c.\ \lambda n.\ n\ eq\ 0\ \text{->}\ 1\ |\ n+c)\ (\lambda_2\ 0)\ 5 \end{aligned} \qquad (3.4\text{-}37a)$$

The next step in blackboard order is to apply λ_2 to 0. But this leads to a loop

$$\begin{aligned} & \lambda_2\ 0 \\ \simeq\ & \lambda_1\ \lambda_1\ 0 \\ \simeq\ & F\ (\lambda x.\ \lambda_1\ \lambda_1\ x)\ 0 \\ \simeq\ & (\lambda c.\ \lambda n.\ n\ eq\ 0\ \text{->}\ 1\ |\ n*c)\ (\lambda_2\ 0)\ 0 \end{aligned} \qquad (3.4\text{-}37b)$$

since the next step (in blackboard order) is again to apply λ_2 to $\underline{0}$. There is of course no problem in normal order.

Compare this evaluation carefully with that in (3.4-8) on page 3.4-134. There we avoided a loop by tucking the loop-inducing combination away as the body of a λ-expression. This kept us out of trouble for the simple combination

$$[\lambda v.\ F(v\ v)]\ [\lambda v.\ F(v\ v)] \qquad (3.4\text{-}38a)$$

but fails for the "two-level" combination

$$[\lambda v.\ F(\lambda x.\ v\ v\ x)]\ [\lambda v.\ F(\lambda x.\ v\ v\ x)]\ 0 \qquad (3.4\text{-}38b)$$

because of the presence of the extra rand $\underline{0}$. We can solve this problem by using two levels of padding with semantically irrelevant λ's, by defining

$$Z'' \triangleq \lambda F.\ (\lambda u.\ u\ u)\ (\lambda v.\ F(\lambda x.\ \lambda y.\ v\ v\ x\ y)) \qquad (3.4\text{-}39)$$

Here we have once again used η-conversion, as in (3.4-10). The reader should satisfy himself that

$$Z''\ F\ 5$$

terminates under blackboard order, although $\underline{Z''}$ is even less efficient than $\underline{Z'}$.

It is not hard to see that $\underline{Z''}$ does not solve all our problems: Consider

```
let rec
    (       c = g a1 a2
            within
        g n1 n2 = Q                    (3.4-40)
    )
in
M
```

This loops using $\underline{Z''}$, but would terminate for

$$Z''' \equiv \lambda F.\ (\lambda u.\ u\ u)\ (\lambda v.\ F(\lambda x.\ \lambda y.\ \lambda z.\ v\ v\ x\ y\ z))$$

Indeed, for any PAL program of the form

```
let rec
    (       c = g a1 a2 ... ak
            within
        g n1 n2 ... nk  =  Q           (3.4-41)
    )
in
M
```

we can find an appropriate fixed point operator $\underline{Z^{(\kappa)}}$ to accommodate it. Conversely, however, given a design decision that always adopts any particular $\underline{Z^{(\kappa)}}$ in desugaring $\underline{rec}$, one can always write a program such as (3.4-41) that cannot be accommodated, even though it could be evaluated in normal order.

3.5 The Gedanken Evaluator

The blackboard methodology developed in preceding sections has been motivated primarily by human considerations. We have sought to gain insight into the structure of the environment tree built up in the course of a computation, and into how computational efficiency relates to order of evaluation. To these ends, we have sought to organize and display successive stages of an evolving computation in an easily comprehensible format.

It should be recognized, however, that our blackboard procedure has not been defined precisely. Indeed, precise definition is unnecessary for two reasons: First, we have been concerned exclusively with applicative expressions, the semantics of which are already established in terms of the λ-calculus and the postulates underlying the universe of discourse. And second, the procedure is to be carried out by people, who presumably can resolve uncertainties in accordance with what they understand to be the established semantic intent.

Although the λ-calculus suffices to define the semantics of the applicative subset of PAL, we have mentioned at the beginning of this chapter that semantic specification of PAL's imperatives involves additional concepts. Specifically, the introduction of imperatives into PAL corresponds to a change of perspective: Beforehand an evaluator is peripheral, in the sense that its operation mirrors the effect of evaluation via the rules of the λ-calculus. But hereafter the role of the evaluating mechanism becomes central, since the introduction of imperatives admits constructs whose semantics apparently cannot be expressed, at least in any natural way, in terms of the λ-calculus. Thus the semantics of imperatives cannot be specified without specifying the evaluator itself. Our purpose in this section is to develop an evaluator for applicative PAL. In chapters 4 and 5, we expand the capabilities of the evaluator so that it handles imperative PAL also.

Methodology and Objectives

The objective of the present section is to introduce a "gedanken evaluator" for the applicative subset of PAL which

(a) is isomorphic to our blackboard procedure, in the sense that both always produce the same result in substantially the same way, and which also

(b) can be easily extended in Chapters 4 and 5 to accommodate semantic definition of PAL's imperatives as well.

Our present task is to explain the gedanken evaluator. Since it is nothing other than a very complex algorithm, and since we have claimed (at the beginning of Chapter 2) that the purpose of a programming language is to serve as a set of conventions for communicating algorithms, we take the obvious path and choose to

exhibit the evaluator as a PAL program. It is the operation of this program that we have been simulating with the blackboard evaluator.

Given our decision to exhibit the algorithm in PAL, we are faced with the task of selecting representations of the various computational entities with which we deal, as well as of writing the detailed PAL programs. The usual way to write a program involving various variables such as x, y and z as well as functions such as F, G and H, is in the form

```
let x = ...
and y = ...
and z = ...
in
let F(...) = ...
in
let G(...) = ...
in
let H(...) = ...
in
...
```

Here x, y and z may be used in defining F, G and H; F may be used in G and H; G may be used in H; and all six may be used in the "main program" at the end. In the programs to be presented, we want for the sake of convenience of presentation to exhibit functions such as H above without showing at the same time the preceeding and following text. To this end we use the PAL feature def, which permits writing a single definition in isolation. Thus a writing such as

```
def H(...) = ...
```

defines the function H, but we do not attempt to ascribe semantics to this writing except in a "suitable context". As this section proceeds we present many functions defined by the def construct, providing the "suitable context" by explanatory text in English. In the appendix at the end of this section all of the functions are collected together in order, providing the proper context.

An important question to be decided on in writing any program as this one has to do with representation of the data objects with which the program deals. In the blackboard evaluator we know that "x" appearing in the control represents a variable, while "2" represents a constant. In order to write a PAL program that does the same thing we must decide how to represent constants and variables, as well as λ-expressions and other constructs, so that the necessary operations can be done. For the most part we ignore issues of representation in the early part of this section. Instead, we assume that the necessary predicates, such as "Is_variable" and "Is_constant" suggested above, can be written. All of the necessary details are provided eventually at the end of this section, but the main part of the section may be read without worrying about them. The earlier programs are not incorrect -- just incomplete.

Logical Bootstrapping: In a sense, use of PAL to describe the gedanken interpretor involves the circular logic of using a language to describe itself. Such a procedure is not unusual in the field of computation -- a classical exercise is to write in some language a compiler for the language, and then to claim that the program "explains" the language. The utility of the exercise is that one need then understand only a single program (presumably written in a subset of the language) in order to understand the whole language.

This advantage, of course, is one which we secure automatically. In addition, however, we achieve a stronger result: By writing the interpretor in terms of the applicative subset of PAL, we gain an "evaluating machine" whose operation is defined by axiom. The semantics of our program then is defined by the reduction axioms of the λ-calculus, the axioms defining PAL's universe of discourse, and the desugaring rules. Thus our specification of PAL involves logical bootstrapping rather than logical circularity. (A small lacuna in the argument remains by virtue of our decision to use PAL syntax in defining the interpretor. To desugar the interpretor program into a pure AE, however, is an intellectually vacuous exercise in virtuosity and perseverance.) This bootstrapping activity is continued in Chapters 4 and 5, wherein the full PAL interpretor is developed in two phases. The first phase presumes prior definition of PAL's applicative subset, called R-PAL, and extends to the language L-PAL which includes assignment, sequences, and structures. The second phase in turn presumes prior definition of L-PAL and completes extension of the language to J-PAL to include labels, goto's, and the constructs valof and res. J-PAL is the PAL of the PAL Manual.

Overview of this Section: Our objective in the rest of this section is to write in PAL the function Gedanken_evaluator which accepts a PAL program as input and whose value is the value of the program. We divide our labors into two parts: translation of the program into control structure, and evaluation of the results. Thus we write

```
def Gedanken_evaluator Program =
    let Control_structure = Translate Program
    in
    Evaluate( Control_structure, Empty_stack, PE )
```
(3.5-1)

The input to Translate is some representation of a PAL program, about which we have much to say later, and Control_structure is much akin to the control sequences we have been using as input to the blackboard machine.

The function Evaluate works very much as does the blackboard evaluator. Its three arguments are the initial control, stack and environment. PE is the primitive environment, and the stack is initially empty.

Our next goal is to specify the functions Translate and Evaluate. As part of doing so, we must specify the representations of the PAL program, the control structure, and the control, stack and environment of the evaluator. We proceed by defining first the representation of control structure and of the components C, S and E, defining next the workings of Evaluate, defining next represention

of PAL programs, and finally specifying the function Translate. In our initial discussion of Evaluate we ignore the possibility of simultaneous definitions. In discussing Translate we show the control structure which they give rise to, and then later we go back to Evaluate to show the changes needed to accomodate them.

Representation of the Control, Stack and Environment

As we have seen, the gedanken evaluator with which we are concerned has three components. We refer to the control, stack and environment as C, S and E, respectively. (In Chapter 4 we add a fourth component M, for memory.) We sometimes refer to the evaluater as the "CSE machine".

Historically, our CSE machine is a direct descendant of Landin's SECD evaluator, described in Landin (1964). We have permuted the letters to correspond to the way the components are written in blackboard evaluation, and we have subsumed Landin's dump (the "D") by using environment markers in the control and stack. The idea is the same.

Lists and their Representation: Our next task is to choose representations of the components C, S and E. C and S are each represented by a list, so it is to lists that we first turn our attention. As we have done in the past, we start with a structure definition:

> A list of objects is either
> empty, or it is
> non-empty, in which case it has a (3.5-2)
> top, which is an object, and a
> rest, which is a list of objects.

Note that "object" is, in a sense, a free variable in this definition. Thus we can use terminology such as "list of integers" to refer to a list in which each top is an integer. We choose to represent the empty list by "nil", and a non-empty list by a 2-tuple whose first component is the top and whose second component is the rest. Various functions useful when dealing with lists are defined by

```
def t(x, y)  =  x          // top of a list                (3.5-3)
and r(x, y)  =  y          // rest of a list
and Push(t, r)  =  t, r    // make a list

def rec Prefix(L1, L2)  =  // concatenate 2 lists
     Null L1 -> L2 |
     Push(t L1, Prefix(r L1, L2))

def 2d x  =  t(r x)        // second element
and 3d x  =  t(r(r x))     // third
and r2 x  =  r(r x)        // rest of rest
and r3 x  =  r(r(r x))     // rest of (rest of rest)
```

We frequently find it useful to represent lists graphically. An obvious way to represent a 2-tuple is shown in part (a) of Figure 3.5-1. Since in general the "rest" is also a list, we usually find it more convenient to use the

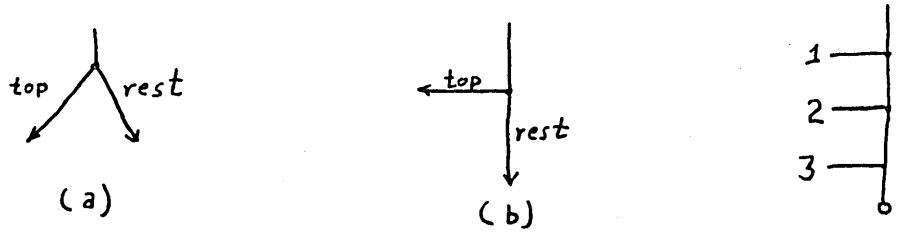


Figure 3.5-1: Graphical Representations

vertical display of part (b) of the figure. Thus the list which in PAL might be written as

(1, (2, (3, nil)) (3.5-4)

can be presented graphically as in part (c). The circle at the bottom stands for the empty list, nil. The various "tops" of the list are its components. We sometimes have lists some of whose components are lists.

Representation of Control: We start with three structure definitions:

A control structure is a
list of control items. (3.5-5a)

A control item is either a
constant, or a
variable, or a
λ-expression, or
BETA, or
AUG, or
RETURN, or
GAMMA. (3.5-5b)

A λ-expression has a
bv-part, which is a variable, and a
body, which is a control structure. (3.5-5c)

We later change the last definition, permitting structured bv-parts in λ-expressions so as to accomodate simultaneous definitions. RETURN corresponds to an environment marker in the control of the blackboard machine, signifying termination of a subproblem. The other control items are as in the blackboard machine.

We have stated previously that writing structure definitions such as these implies the existence of certain predicates, selectors and constructors. Any representation we might choose must permit us to write these functions, or the representation is not satisfactory. As suggested earlier, we do not choose to specify now the representation we use for each of the items just listed,

defering that until later. We claim though that we can write the predicates Is_constant, Is_variable and Is_lambda_exp; the selectors bV and Body (to be applied to λ-expressions); and the constructor Cons_lambda_exp. We also assume that variables named BETA, AUG, RETURN and GAMMA are in the environment with values which are in the domain of the functor eq. All of these entities are defined in the PAL programs at the end of this section.

As a convention, predicates written for use in the gedanken evaluator always have names that start with "Is_", such as "Is_variable", to distinguish them from built-in PAL names such as "Isstring". Recall that the underscore may be used as part of the name of a variable in PAL.

Conditionals, such as

B -> M | N

are accommodated (as in blackboard evaluation) by a control which

begins with the control corresponding to the premise B,
followed by the control symbol β,
followed by the control structure corresponding to N, the false arm,
followed by the control structure corresponding to M, the true arm,
followed by the remainder of the control.

Thus we have encompassed all cases of current interest.

An example illustrating the format of control structures is provided in Figure 3.5-2, which corresponds to the PAL program

(let a = 5 in (Zero a -> (b where b=3) | a) + 2) * 3 (3.5-6)

Of course, the structure definitions (3.5-2) and (3.5-5) and the example of Figure 3.5-2 do not suffice to specify completely what control structure corresponds to any given input PAL program: This correspondence is established formally by the function Translate. But for the time being we can rely upon the intuition already gained via blackboard evaluation.

Representation of the Stack: As with the control, we start with two structure definitions:

A stack is a list of stack items. (3.5-7a)

A stack item is either a
PAL constant, or a
PAL tuple, or a
λ-closure, which has a (3.5-7b)
bv-part, which is a variable, and a
body, which is a control structure, and an
environment, or it is
Y, or an
eta-object, or an
environment.

(let a = 5 in (zero a → (b where b = 3) | a) + 2) * 3

$[\lambda_1 a.\ (zero\ a \rightarrow_2 (\lambda_3 b.\ b)\ 3 \mid_4 a) + 2]\ 5 * 3$

$\gamma\ \gamma * \gamma\ \lambda_1^a\ 5\ 3$

$\delta_1 = \gamma\ \gamma + \delta_2\ \delta_4\ \beta\ \gamma\ zero\ a\ 2$

$\delta_2 = \gamma\ \lambda_3^b\ 3$

$\delta_3 = b$

$\delta_4 = a$

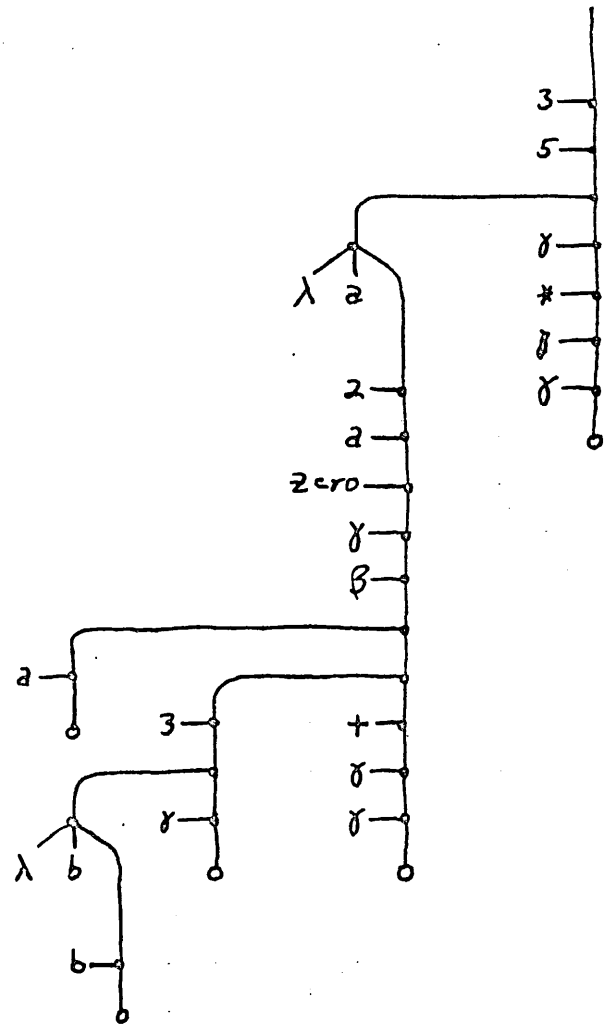


Figure 3.5-2: Control Structure for (3.5-6)

We again postpone discussion of the details of the representation. We assume that we can write the predicates Is_constant, Is_tuple (which differs from Istuple), Is_closure, Is_Y and Is_eta; the selectors bV, Body and Env (the first two of which can be applied to either a λ-expression or a λ-closure); and the constructor Cons_lambda_closure. Cons_lambda_closure takes two arguments: a λ-expression and an environment.

Representation of Environment: As in the blackboard evaluator, an environment provides a pairing between names and values, along with a link to another environment. The structure is

```
An environment is either
    empty, or it is
    non-empty, in which case it has a
        name, and a
        value, and an
        environment.
```
(3.5-8)

We represent the empty environment by nil, and a non-empty environment by a 3-tuple whose first component is the name, whose second component is the associated value, and whose third component is the next environment layer. Names in the environment are represented as are variables in the control, and values are represented as they are in the stack.

We assume initially (and exhibit later) a function Lookup, such that the call

Lookup(Var, Env)

returns the value associated with variable Var in environment Env.

The Evaluator

We have said that the gedanken evaluator has two parts: a translator and an evaluator. We discuss in this section the latter. The definition of the evaluator is given by the following PAL program:

```
def rec Evaluate(C, S, E) =
    Null C -> t S |                                          (3.5-9)
    (   let New_C, New_S, New_E = Transform(C, S, E)
        in
        Evaluate(New_C, New_S, New_E)
    )
```

The function Transform is applied to a CSE 3-tuple corresponding to the state on one line of the blackboard evaluator, and returns that CSE 3-tuple corresponding to the next line. Evaluate calls itself recursively until the control is empty, at which point it returns as the "answer" the top stack item. Our task in this section is to specify the workings of Transform.

Strategies in the Evaluator: Quite a few arbitrary decisions were made in development of the blackboard evaluator, and some of these require further discussion in connection with the gedanken evaluator. For example, we elect to use the Y-η method in the gedanken evaluator to accomodate recursion, since it seems to be the method which is most efficient, both in its operation and conceptually, of those that are available. (It is unfortunately not practical to express the algorithm of Y" in R-PAL, since updating of the "?" in the environment really requires an assignment command.) In the desugaring process to be specified in Translate, definitions of recursive functions transform into the application of the variable "Y#" to a suitable λ-expression. Since this name is not acceptable syntactically as a variable name in PAL, its use cannot conflict with any name used by the programmer. The primitive environment PE supplied to Evaluate associates Y# with a special built-in value which can be recognized in Transform so that requisite special processing can be done. Applying Y# to a closure results in an eta-object in the stack, which has the closure associated with it.

Tuples are accomodated by the control item AUG, and a tuple such as

3, 4, 5 (3.5-10a)

written by the programmer is desugared by Translate as if the programmer had written

nil aug 3 aug 4 aug 5 (3.5-10b)

When AUG appears at the top of the control, the effect is to build the appropriate tuple from the top two stack items.

The Function "Transform": Recall that Transform is applied to a CSE 3-tuple corresponding to a line of blackboard evaluation and returns that 3-tuple which represents the next line. It is defined as follows:

```
def Transform(C, S, E) =                                    (3.5-11)
    let  A = C, S, E   // to save writing later
    and x = t C        // the top control item
    in
       is_constant x     -> Eval_constant A
    |  is_variable x     -> Eval_variable A
    |  is_lambda_exp x   -> Eval_lambda_exp A
    |  x eq BETA         -> Do_conditional A
    |  x eq AUG          -> Do_aug A
    |  x eq RETURN       -> Do_return A
    |  x eq GAMMA
         -> ( let r = t S  // the rator
              in
                 is_closure r  -> Apply_closure A
              |  is_constant r -> Apply_constant A
              |  is_tuple r    -> Apply_tuple A
              |  is_Y r        -> Apply_Y A
              |  is_eta r      -> Apply_eta A
              |  error
            )
    |  error
```

There are several points to note about this program:

- The function Transform is not recursive, the recursion being done by Evaluate.
- It uses eleven functions which we have yet to write: three to do "Eval"s, three to do "Do"s and five to do "Apply"s.
- The control item RETURN corresponds to the presence in the blackboard evaluator of an environment marker at the top of the control. It signifies a sub-problem exit.
- The program uses the non-PAL reserved word "error", whose semantics should be obvious.
- It makes explicit use of the various predicates and selectors for control items and stack items which were discussed earlier.

In the next several subsections, the missing functions aluded to are provided.

<u>The "Eval" Functions</u>: There are three evaluating functions used by Transform, one for each type of control item which has a value: constants, variables and λ-expressions. As in the blackboard evaluator, the value of a constant is implicit in the constant (the details differ), variables are looked up in the environment, and λ-expressions are evaluated to form λ-closures by associating with them the current environment. The three functions are:

```
def Eval_constant(C, S, E) = // Evaluate a constant.
     let V = Value_of(t C)  // its value.
     in                                                    (3.5-12a)
     r C, Push(V, S), E

and Eval_variable(C, S, E) = // Evaluate a variable.
     let V = Lookup(t C, E)  // Look it up.
     in                                                    (3.5-12b)
     r C, Push(V, S), E

and Eval_lambda_exp(C, S, E) = // Evaluate a λ-expression.
     let V = Cons_closure(t C, E)
     in                                                    (3.5-12c)
     r C, Push(V, S), E
```

Note that each of these returns a C-S-E 3-tuple. Consider Eval_constant. The top item in the control when it is called is known to be a constant (or this function would not have been called by Transform), so the function Value_of is called to return the constant's value. (This function is defined later, since it depends on the representations yet to be specified for items in the control and the stack.) The 3-tuple returned by Eval_constant consists of the rest of the control after deletion of the constant, a stack with the new value pushed on top of the old stack, and the old environment.

The other two functions are similar, the second using Lookup to find the value of the variable in the current environment and the third forming a λ-closure by associating the current environment with the λ-expression.

Subproblems: In the CSE machine as in the blackboard evaluator, application of a λ-closure involves setting up a subproblem to evaluate the body in an appropriate environment, leaving enough information to get back to the main evaluation on completion of the subproblem. The new control is formed by pushing the control item RETURN onto the existing control, and then prefixing the body of the closure being applied on top of that. The new stack, as in the blackboard evaluator, contains information about the environment to become current on completion of the subproblem. (The blackboard machine puts an environment marker into the control, too, but this is not needed here.) What we put into the stack is not an environment marker but the environment itself. The relevant functions are:

```
def Apply_closure(C, S, E) =
     let Rator = t S  // What is being applied.
     in                                                    (3.5-12d)
     let New_C = Prefix(Body Rator, Push(RETURN, r C))
     and New_S = Push(E, r2 S)
     and New_E = Decompose(bV Rator, 2d S, Env Rator)
     in
     New_C, New_S, New_E
```

```
and Do_return(C, S, E) =
     r C, Push(t S, r2 S), 2d S                                    (3.5-12e)
```

In making New_E in Apply_closure we have used the function Decompose to make a new environment from a name, value and existing environment. This function, to be specified later, accomodates the structured bv-parts which are needed to handle simultaneous definitions. For the case in which the bv is a simple name, Decompose returns a 3-tuple consisting of its three arguments.

When RETURN is encountered, the top stack item is the "value" of the subproblem and the second item in the stack is the environment that was current just before entry to the subproblem. The rest of the stack is the stack to be reinstated on completion of the subproblem -- the stack that was there before subproblem entry.

Recursion: The reader would be well advised at this point to review the discussion starting on page 3.4-135, in which the Y-η method is explained, before continuing with this discussion. The following code used in the CSE machine formalizes the method:

```
def Apply_Y(C, S, E) =
     let V = (ETA, 2d S)  // ETA
     in
     let New_S = Push(2d S, Push(V, r2 S))                         (3.5-12f)
     in
     C, New_S, E  // Leave GAMMA in C.

and Apply_eta(C, S, E) =
     Push(GAMMA, C), Push(t S 2, S), E                             (3.5-12g)
```

The effect of Apply_Y is replacing (Y λ) by (λ η), where η is a (marked) copy of λ; and Apply_eta involves replacing η by the application of λ to η. Study carefully the new stack created by each of these functions. Note that Apply_Y leaves the GAMMA in the control, to be "reused", and that Apply_eta performs the unusual task of pushing a GAMMA onto the control.

Conditionals, Tuples and Basics: The remaining functions to be discussed are these:

```
def Do_conditional(C, S, E) =
     let Sw = Val_of(t S)  // The boolean arm.
     in
     let Selected_arm = (Sw -> 3d | 2d) C                          (3.5-12h)
     in
     Prefix(Selected_arm, r3 C), r S, E
```

```
and Do_aug(C, S, E) =
      let V = Augment_tuple (t S) (2d S)
      in
      r C, Push(V, r2 S), E                                    (3.5-12i)

and Apply_constant(C, S, E) =
      let V = Apply (t S) (2d S)
      in
      r C, Push(V, r2 S), E                                    (3.5-12j)

and Apply_tuple(C, S, E) =
      let V = Apply (t S) (2d S)
      in
      r C, Push(V, r2 S), E                                    (3.5-12k)
```

In Do_conditional the function Val_of is needed because objects on the stack of the CSE machine have identifying tags, which must be removed. The function Augment_tuple used by Do_aug is equally concerned with representation.

The Translator

We have now specified that part of the gedanken evaluator that performs evaluation of control structures, and have next to explain the function Translate that transforms PAL input into control structure. Translate does essentially the same task done by the average compiler for a language such as Fortran or PL/I, except that it produces control structure instead of machine code. Compilers are almost always organized into two or more phases, so that there is an initial activity of reading source text and parsing it according to the syntax of the language, followed by an activity concerned with creation of executable code. The interface between these two activities is often some sort of tree representation of the input text, wherein the nodes of the tree correspond to the syntactic categories of the language.

It would seem then that the function Translate should be similarly organized. However, we choose instead to bypass entirely the parsing activity, assuming as input to Translate a tree representation of the program to be analyzed. This decision is based mainly on the fact that syntactic analysis is very well understood, there being a large collection of published literature on it. Since our purpose in these notes is to illuminate certain aspects of programming languages, we choose to devote ourselves to those aspects of the problem which are less well understood. It is not that we regard syntactic analysis as uninteresting, but rather that we elect to study other subjects.

Our present task then is two-fold: We must specify exactly the form of the input to Translate, and we must specify Translate's algorithm. In the remaining parts of this section we first show three ways to describe syntax, the last of which leads us to the form used as input. We finally discuss in detail how Translate operates.

Tree Syntax: Heretofor we have expressed PAL's syntax using the BNF notation discussed in Section 1.2 of the PAL Manual. For example, the following equations taken from Appendix 2.1 of the PAL Manual define the syntax of arithmetic expressions, abbreviated "A":

```
A   ::= A + A1 | A - A1 | A1
A1  ::= A1 * A2 | A1 / A2 | A2
A2  ::= A3 ** A2 | A3
A3  ::= A3 % NAME R | R
R   ::= R R1 | R1
R1  ::= CONSTANT | VARIABLE | ( A )
```

(3.5-13)

(The definitions of the categories A, A3 and R1 have been simplified slightly.) We know that this syntax provides a unique parse for any string which is an instance of an A.

Now consider instead the following syntax:

```
A  ::= A OP A | A A | ( A ) | CONSTANT | VARIABLE
OP ::= + | - | * | / | ** | % NAME
```

(3.5-14)

This ambiguous syntax defines the same set of strings as does the syntax of (3.5-13), but it pays for being shorter by being ambiguous. For example, this syntax permits the string "a+b*c" to be parsed as either "a+(b*c)" or as "(a+b)*c".

Related closely to (3.5-14) is the tree syntax shown in Figure 3.5-3:

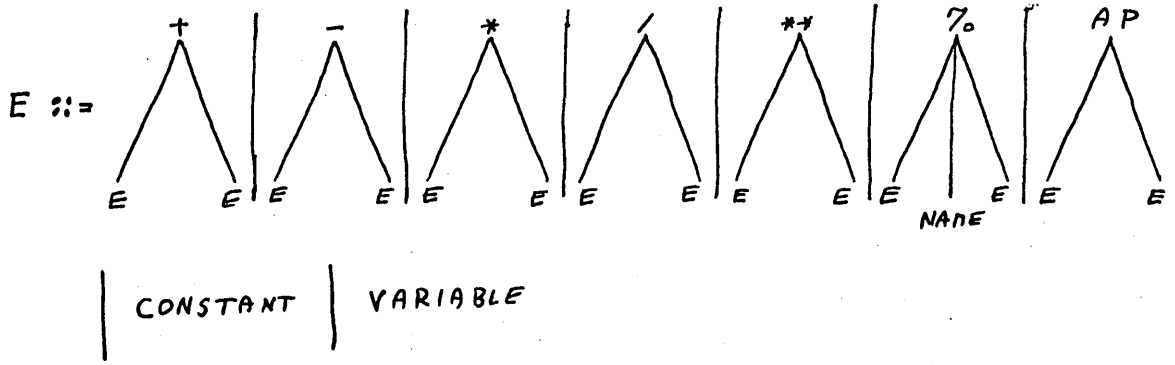


Figure 3.5-3: Tree Syntax for Arithmetic Expressions

The legal utterances in the language defined by this syntax are trees, such as either of the following:

Note that the possibility of ambiguity does not exist in this language, just as it did not exist in the prefix or postfix forms discussed at the beginning of this chapter. The input to Translate is the type of tree of which these are samples. Another example of input to Translate is given in Figure 3.5-4, which shows the tree form of the PAL program of (3.5-12h), the function Do_conditional in the CSE evaluator. (Some of the variable names have been shortened to ease the task of drawing the figure.)

The applicative subset of PAL is characterized precisely by the ambiguous syntax of Figure 3.5-5 and the tree syntax of Figure 3.5-6. As shown, each principle syntactic alternative of the ambiguous syntax leads to a node-type in the tree syntax. The overall syntax may then be viewed as specifying rules for piecing nodes together into syntax trees. For example, an AP node (corresponding to application) may be formed by connecting together two E's as the sons of an AP node, and so forth.

Two comments may prove helpful in conjunction with Figure 3.5-6. First, the decision as to what syntactic alternatives are to be accorded a node-type is somewhat arbitrary: All that is entailed is a division of responsibility between the syntactic anaysis done before inputting a program to Translate, and the subsequent processing done by Translate. The precise boundaries of the division are unimportant, the only substantive issue being to gain conceptual clarity by separating the jobs of tree generation and tree processing into distinct tasks. For example, we have chosen to avoid an overabundance of node-types by stipulating that infix and prefix operators be input as shown using node-type BINOP for all infix functors and UNOP for all prefix functors.

The second comment pertains to the absence of tags (i. e., labels on the nodes) in those subtrees that indicate the binding of variables: the left sons of LAMBDA and FF nodes. The use of tags in these subtrees is avoidable simply because bound-variables can not be "computed objects" in PAL; i.e. because expressions such as

```
let (x eq 0  ->  a  |  b) = M in N
```

are not allowed. The structure of such a node is implicit in its position in the overall tree, so no tagging information is needed. A similar remark applies to the right sons of BINOP and UNOP nodes, which are always functors.

The graphical representations of Figure 3.5-6 are easy to visualize, but do not accord directly with PAL's tuples. In particular, tuples do not include "tags". We introduce therefore two functions:

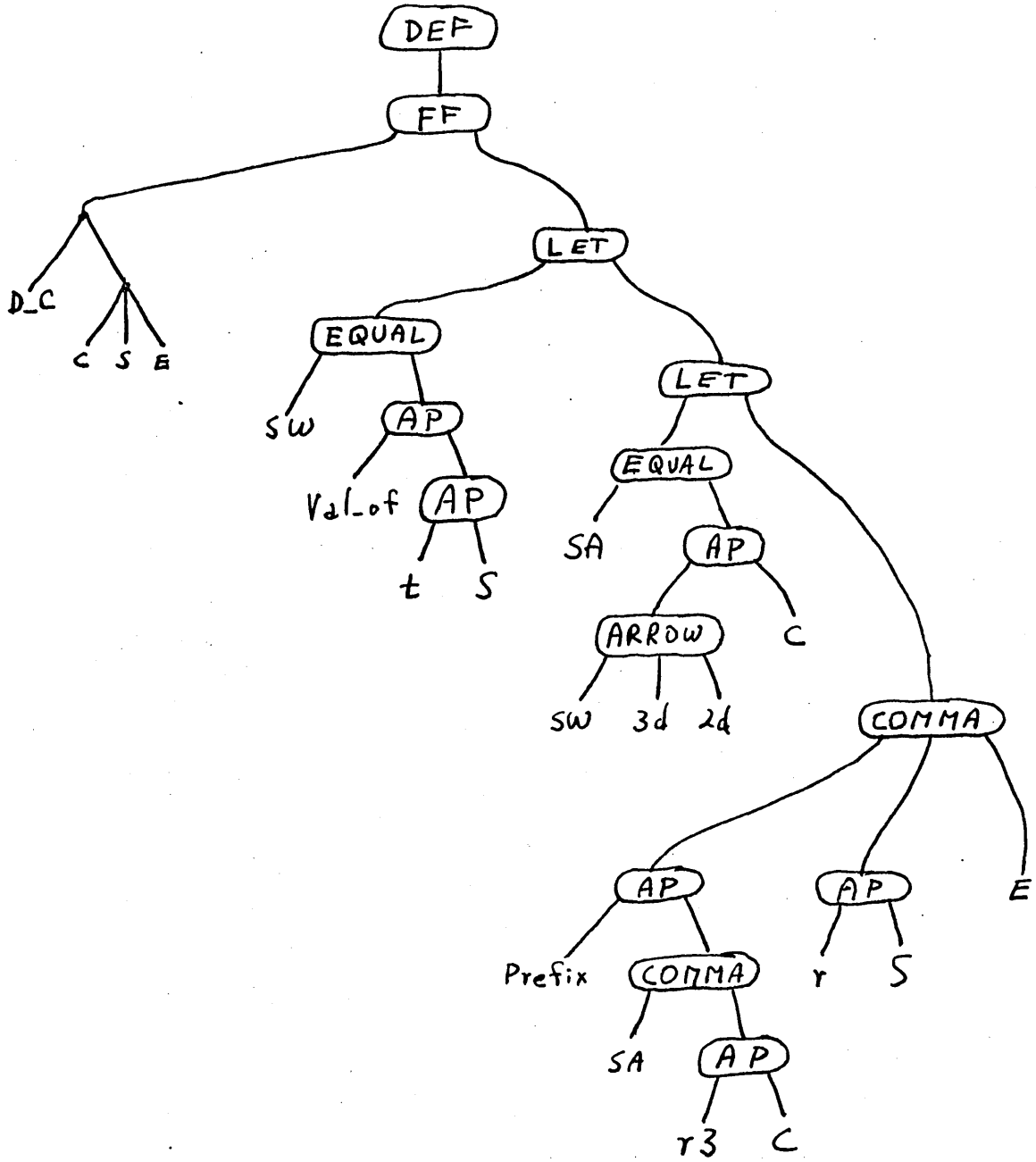


Figure 3.5-4:

Tree for (3.5-12 b)

```
E ::=     let D in E
        | E where D
        | fn V1 . E
        | E → E <bar> E
        | E {, E}∞1
        | E  BINOP E
        | UNOP  E
        | E E
        | NAME
        | CONSTANT
        | ( E )

D ::=     D within D
        | D { and D }∞1
        | rec D
        | NAMES = E
        | NAME V = E
        | ( D )

V ::= { V1 }∞1
V1 ::=  NAME | ( NAMES ) | ( )
NAMES ::=  NAME {, NAME }∞0

BINOP ::= + | - | * | / | ** | & | or | aug | %0 NAME | RL
UNOP  ::= + | - | not
RL    ::= < | > | ls | gr | eq | ne
```

Figure 3.5-5 Ambiguous Syntax for R-PAL

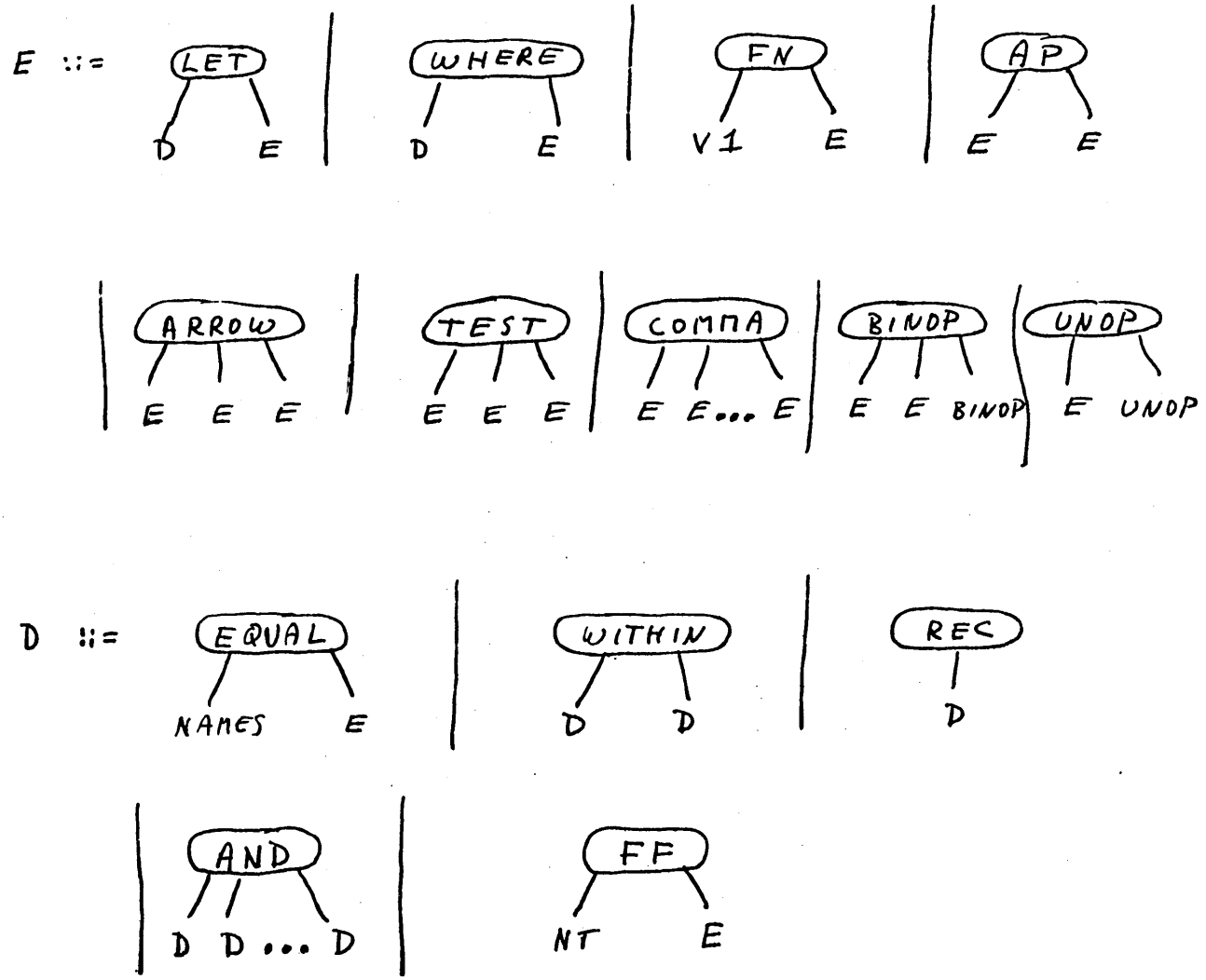


Figure 3.5-6: Tree Syntax for R-PAL.

```
def Tag n s      =  s aug n
and Is_tag s n   =  n eq s(Order s)
```
(3.5-15a)

which serve to "tag" a node and to "test its tag", respectively. If M and N are PAL expressions, it follows then that the ob denoted by

```
Tag GAMMA (M, N)
```

may be visualized as either of the following:

The left picture shows the form corresponding to the trees of Figure 3.5-6, while on the right is a form more in accord with the way we usually think of a 3-tuple. We hereafter choose the graphical form on the left as being more perspicuous. If w is any tuple tagged with GAMMA, then the PAL expression

```
Is_tag w GAMMA
```

denotes true, whereas

```
Is_tag w v
```

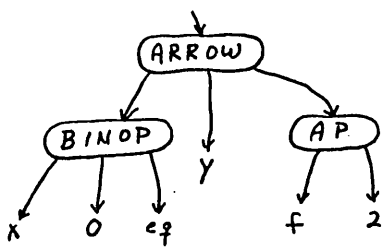
denotes false, whenever v denotes any ob other than GAMMA within the domain of "eq". Since tagging a tuple adds another component to it, we introduce also the following two functions:

```
def Get_tag s  =  s(Order s)
and Sons s     =  Order s  -  1
```
(3.5-15b)

With the functions now available, and assuming that variables such as AP, FN, LET, etc., are defined with suitable values (as we have been assuming all along about GAMMA and others) the PAL expression

```
x eq 0  ->  y  |  f 2
```
(3.5-16)

would be represented by the tree



and hence by a PAL expression something like

```
Tag ARROW [Tag BINOP (x, 0, eq), y, Tag AP (f, 2)]
```
(3.5-17)

(The details of the representation of names and functors have yet to be specified. The preceeding use of "eq" is of course not legal PAL.)

The Function "Translate": It proves expedient to break the job of translation of syntax trees to control structure into two parts: standardizing the syntax tree, and flattening the result into a control structure. Thus we have

```
def Translate Program =
    FF( ST Program, nil )                                    (3.5-18)
```

ST produces a standardized tree which is similar to the original tree but which includes only nodes of type GAMMA, BETA, LAMBDA and AUG. Thus ST's operation is much like desugaring PAL into pure AE's. FF processes standardized trees, flattening them into control structure.

The general rule for standardization is

```
To standardize a structure...
    first standardize its sons,                              (3.5-19)
    and then assemble the result into standard form.
```

The details of ST are shown graphically in Figure 3.5-7, which shows in the left columns each permissible node type (those listed in Figure 3.5-6) which may be input to Translate, and which shows in its right columns what ST does to it. The asterisks indicate standardization of the sons. For example, the first picture shows that an AP node may be standardized by first standardizing its sons, and then building a GAMMA node with those sons. Note the adherence to the principle enunciated in (3.5-19).

A standardized definition is always an EQUAL node, and the standardization process is much the same as the desugaring process which we have been doing. However, we now choose to face squarely the problem of desugaring simultaneous definitions, a problem which up to now we have been ignoring. To see the problem, consider the following PAL program, which is similar to that in (3.2-30a) on page 3.2-124.

```
let f = F
and (       w = W
            within
      g x = G
      and                                                    (3.5-20a)
      h y = H
    )
in
M
```

The desugaring and standardization processes are similar, each leading to

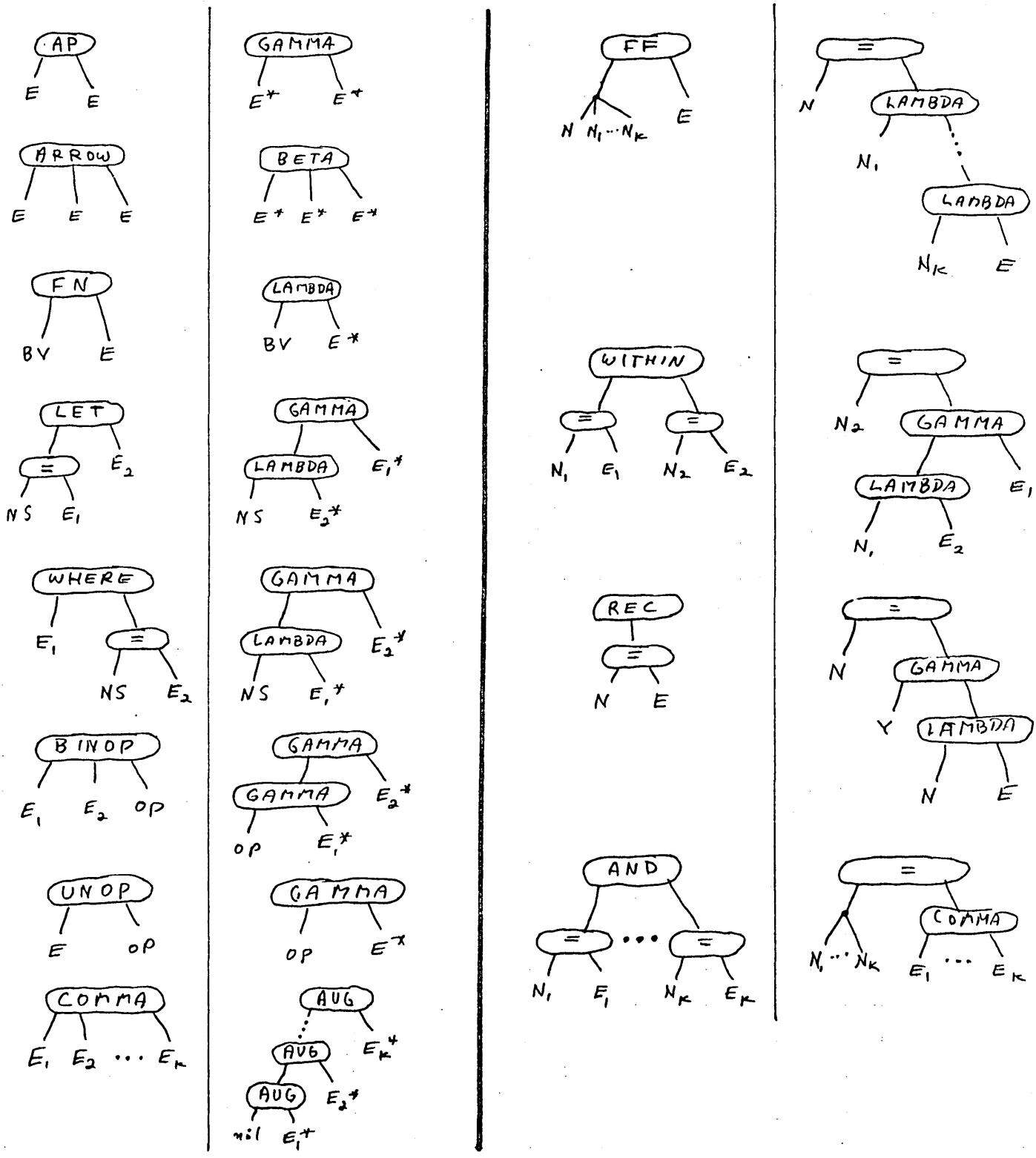


Figure 3.5-7 Standardization of Trees

```
let f = F
and g, h = [λw. (λx.G), (λy.H)] W
in                                                    (3.5-20b)
M
```

and then to

```
let f, (g, h) = F, [λw. (λx.G), (λy.H)] W
in                                                    (3.5-20c)
M
```

and finally

```
[λ(f, (g, h)).M] {F, [λw. (λx.G), (λy.H)]W}           (3.5-20d)
```

The tree forms of (3.5-20a), the original program, and (3.5-20c), in which the definition is standardized, are shown in Figure 3.5-8. Note that (3.5-20c) and

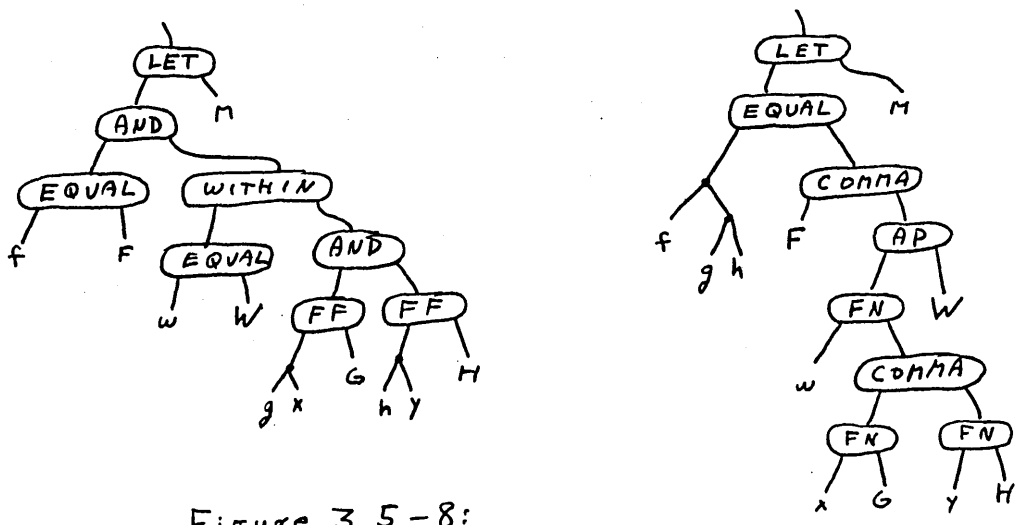


Figure 3.5-8:

(3.5-20d) are <u>not</u> legal PAL, since structured bv-parts are not permitted. However, it is clear that PAL's syntax permits definitional structures which, on standardization, produce arbitrarily complicated bv-parts. (This is the problem that was aluded to in the discussion of Apply_closure on page 3.5-162.) The decision to exclude structured bv-parts from PAL is an arbitrary one, reflecting the ideas of PAL's designers as to what constitutes "good human engineering". The language would not be changed in any important way were the decision to be reversed. Indeed, the standardization functions would be unchanged.

The functions D and ST which do the job of standardization are shown in Figures 3.5-9 and 3.5-10, respectively. These programs formalize the process shown graphically in Figure 3.5-7, the same process which is done in preparing input for blackboard evaluation.

```
def rec D x =  // Standardize a definition.
     let Type = is_tag x
     in
        Type EQUAL  ->  x  // Already OK.
     |  Type WITHIN
          -> ( let u, v = D(x 1), D(x 2)
               in
               EQUAL_ (v 1) ( AP_ (FN_ (u 1) (v 2)) (u 2) )
             )
     |  Type REC
          -> ( let w = D(x 1)
               in
               EQUAL_ (w 1) ( AP_ Y_VAR (FN_ (w 1) (w 2)) )
             )
     |  Type FF
          -> ( let rec Q k T =
                       k < 2  ->  T
                       | Q (k-1) (FN_ (x 1 k) T)
               in
               EQUAL_ (x 1 1) (Q (Order(x 1)) (x 2))
             )
     |  Type AND
          -> ( let rec Q k S T =
                       k > Sons x  ->  (S, T)
                       |  ( let w = D(x k)
                            in
                            Q (k+1) (S aug w 1) (T aug w 2)
                          )
               in
               let L, R = Q 1 nil nil
               in
               EQUAL_ L (Tag COMMA R)
             )
     |  error
```

Figure 3.5-9: The Function "D"

```
def rec ST x  =  // Standardize abstract syntax tree.

      let Type = Is_tag x
      in
         Is_identifier x  ->  x
      |  Type BETA or Type TEST or Type ARROW
           ->  BETA_ (ST(x 1)) (ST(x 2)) (ST(x 3))
      |  Type FN
           ->  LAMBDA_ (x 1) (ST(x 2))
      |  Type COMMA
           -> ( Q 1 NIL
                where rec Q k t =
                      k > Sons x  ->  t
                    |  Q (k+1) ( AUG_ t (ST(x k)) )
              )
      |  Type PERCENT
           -> GAMMA_ (x 2) ( AUG_ (AUG_ NIL (ST(x 1))) (ST(x 3)) )
      |  Type LET
           -> ( let w = D(x 1)  // Standardize the definition.
                in
                GAMMA_ ( LAMBDA_ (w 1) (ST(x 2)) ) (ST (w 2))
              )
      |  Type WHERE  ->  ST(LET_ (x 2) (x 1))
      |  Type AP  ->  GAMMA_ (ST(x 1)) (ST(x 2))
      |  Type BINOP
           ->  GAMMA_ ( GAMMA_ (CONSTANT, x 3) (ST(x 1)) ) (ST(x 2))
      |  Type UNOP
           ->  GAMMA_ (CONSTANT, x 2) (ST(x 1))
      |  Type AUG  ->  AUG_ (ST(x 1)) (ST(x 2))

      |  error
```

Figure 3.5-10: The Function "ST"

Flattening Standardized Trees: The final step in producing control structure for input to the CSE machine involves the flattening function FF(x, c). In visualizing the operation of this function, it helps to recognize that its principal effect is to extract a new control item from the (standardized) tree x and to push it onto the nascent control structure c. FF calls itself recursively until the original tree is exhausted. Since standardized trees are made up of only GAMMA, LAMBDA, BETA and AUG nodes, there are only four cases to consider. Figure 3.5-11 shows graphically the effect of FF, and Figure 3.5-12 shows the code.

Example of Translate: Consider the following PAL program:

```
let f x = x + (x > 0 -> 1 | -1)
in                                                    (3.5-21)
f 2 * f(-3)
```

The tree form of this program, along with the standardized tree and control structure, are shown in Figure 3.5-13.

Other Topics

We have one problem which we have so far bypassed -- simultaneous definitions -- and one possible defect which we have yet to mention, that of overspecification of order of evaluation. We consider these in turn, and then discuss some of the implications of our logical bootstrap.

Simultaneous Definitions: The only problem in R-PAL remaining to be discussed concerns simultaneous definitions. Recall from Figure 3.5-7 on page 3.5-171 that a standardized definition may have a left side which under transformation by the function ST is converted into a λ-expression having that same tree as its bound variable part. However, the structure of an Environment as given by (3.5-8), page 3.5-158, is a simple linking of (name, value) pairs. The task of unraveling a tree of bound variables in the application of a λ-closure has been relegated to the subsidiary function Decompose. As indicated in (3.5-12d) this function takes three arguments:

- a bound-variable or a tree of bound-variables,
- an ob, which may be a tree of obs, and
- an environment.

The function Decompose is defined as follows:

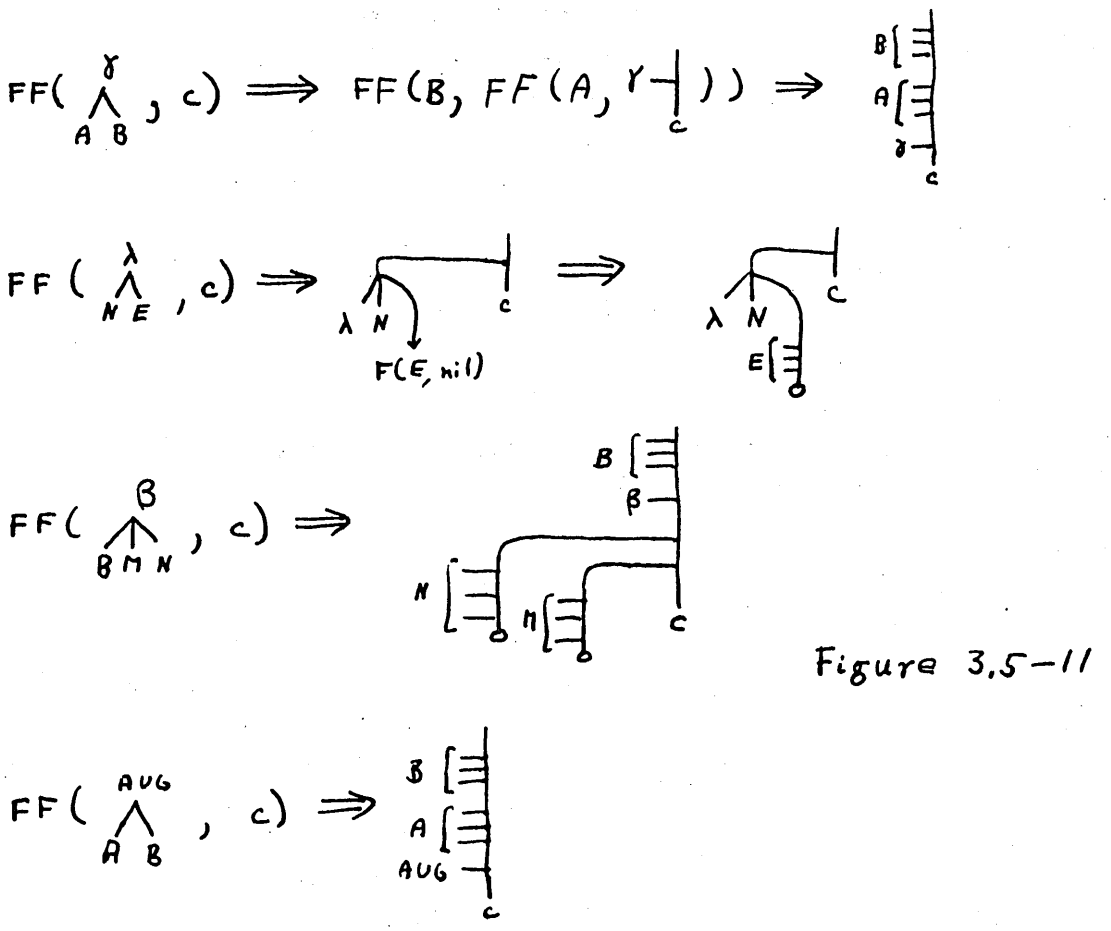


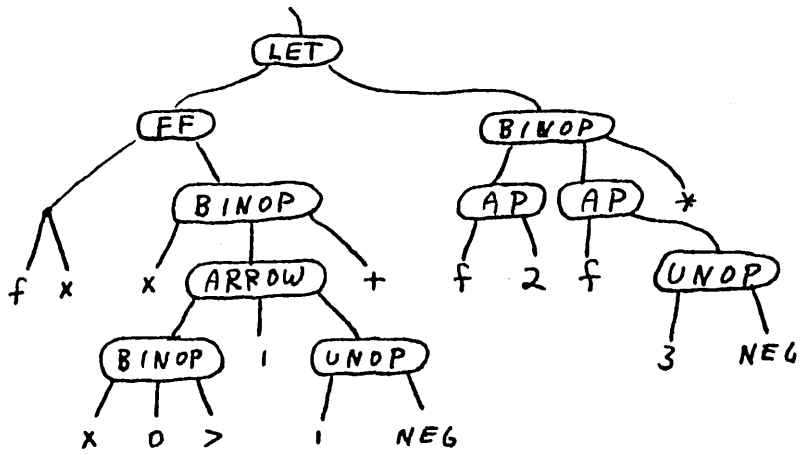
Figure 3.5-11


```
def rec FF(x, c)  =  // Flatten standardized tree x onto control c.

    let Type = Is_tag x
    in
       Is_identifier x -> (x, c)
    |  Type LAMBDA
         -> ( let Body = FF( x 2, nil )
              in
              Cons_lambda_exp(x 1, Body), c
            )
    |  Type BETA
         -> ( let TA = FF(x 2, nil)  // True arm.
              and FA = FF(x 3, nil)  // False arm.
              in
              FF( x 1, (BETA, (FA, (TA, c))) )
            )
    |  Type GAMMA -> FF( x 2, FF( x 1, (GAMMA, c) ) )
    |  Type AUG -> FF( x 2, FF( x 1, (AUG, c) ) )
    |  error
```

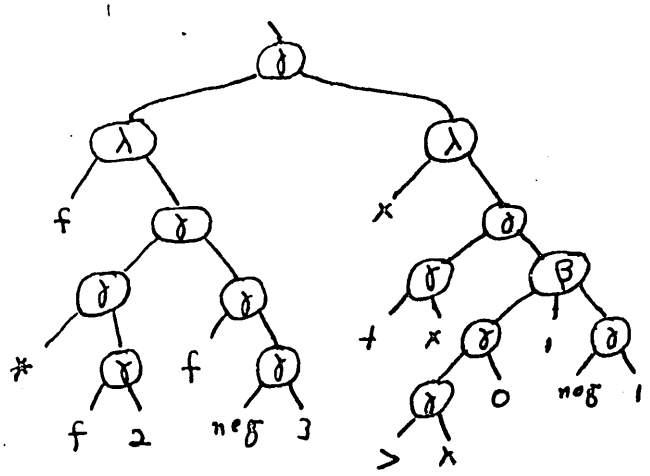
Figure 3.5-12

let f x = x + (x>0 → 1 | -1) in f 2 * f(-3)



[λ¹f.f 2 * f(-3)] [λ²x. x + (x>0 → ³1 | ⁴-1)]

γ λ¹f λ²x
δ1 = * γ f 2 γ f neg 3
δ2 = + x δ3 δ4 β > x 0
δ3 = 1
δ4 = neg 1



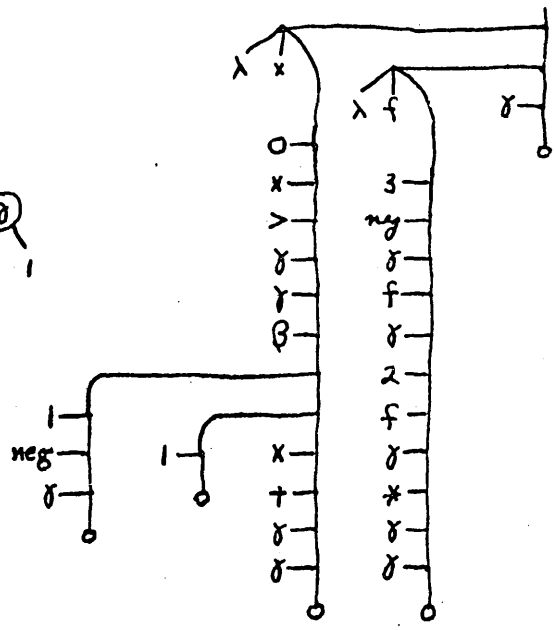


Figure 3.5-13:

```
def rec Decompose(Names, Values, Env)  =
     test  Is_variable Names  // Is it a single variable?
     ifso  (Names, Values, Env)  // Yes, add it to environment.
     ifnot  // Check conformality.
     test  Is_tuple Values
     ifnot error  // Tuple applied to scalar.
     ifso
     test  Order Names eq Order (Val_of Values)
     ifnot error  // Differing tuple lengths.
     ifso  // Process a multiple-bv part.
      (  Q 1 Env
         where rec Q n e =
            n > Order Names  ->  e  |
            Q (n+1) ( Decompose(Names n, (Val_of Values) n, e) )
      )
```

Figure 3.5-14: The function Decompose.

Note that if Names is a single bound variable, the effect of Decompose is to return an environment obtained by placing on top of environment Env a new layer consisting of the pair (Names, Values). If Names is a tuple and Values is conformal thereto -- i.e., has the same number of components -- a succession of layers is placed on Env. Should a component of Names also be a tuple, this tuple too is unraveled and its bound-variables paired with corresponding components of Values. Finally, if Values is not conformal to Names -- i.e., if at any level Names is a tuple and Values is a tuple of different Order -- the function aborts. These four cases are illustrated in Figure 3.5-15.

Order of Evaluation: One who specifies a programming language must decide what to say about the order of evaluation of the constituents of expressions such as

$$a + b \qquad (3.5\text{-}22)$$

There are several possibilities: Evaluation may be done from left to right, from right to left, or in some more complex order which is implicit in the implementation. The decision of PAL's designers, as documented in Section 3.3/S of the PAL Manual, is as follows:

> "No order of evaluation is to be inferred... The reader may find it helpful to think of the process in these terms: The choices are made at the time the expression is evaluated, and the choice made by the evaluating mechanism is dependent on the then-current weather forecast. The programmer is cautioned not to write a program whose successful evaluation depends on a particular order of evaluation of expressions."

Although the intent of the designers is clear, it is equally clear that the formal definition so far presented provides that evaluation is always from right to left. The intent and the result being out of argument, it behooves us to

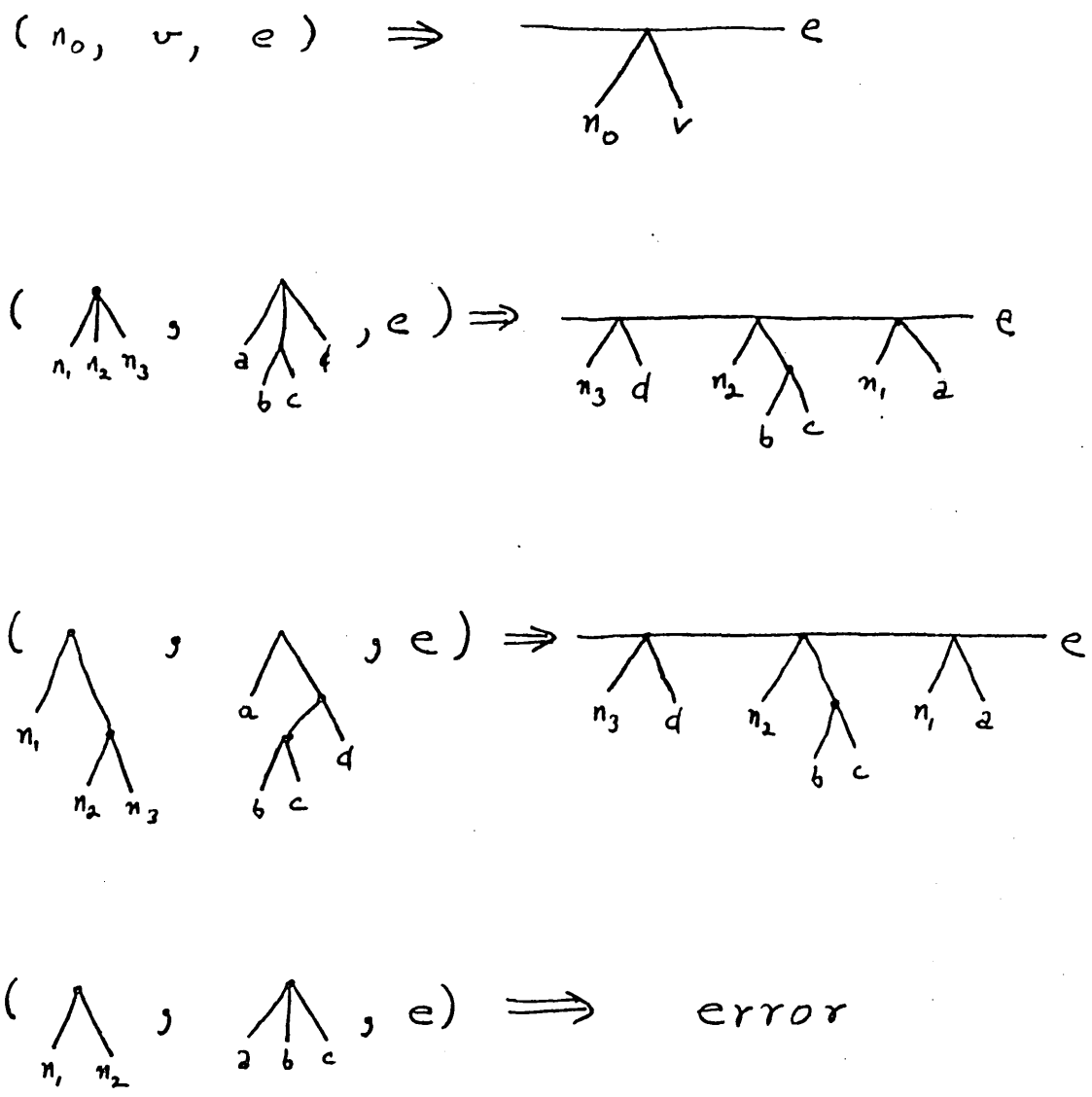


Figure 3.5-15: Processing done by the function Decompose

change one or the other. We elect to change the formal definition, and the reason is not what one might expect. As far as PAL is concerned, it seems to be unimportant which decision is adopted. We elect to leave order of evaluation unspecified so that we may have the opportunity to show how that effect might be achieved. Keep in mind that our objective is not just to define PAL but rather to illuminate various aspects of language definition. The following discussion is of points that can be made conveniently in no other context.

The problem arises from the fact that the rand of a combination is always evaluated before the rator. The standarized tree and control structure for "a + b" are shown in Figures 3.5-16(a) and (b):

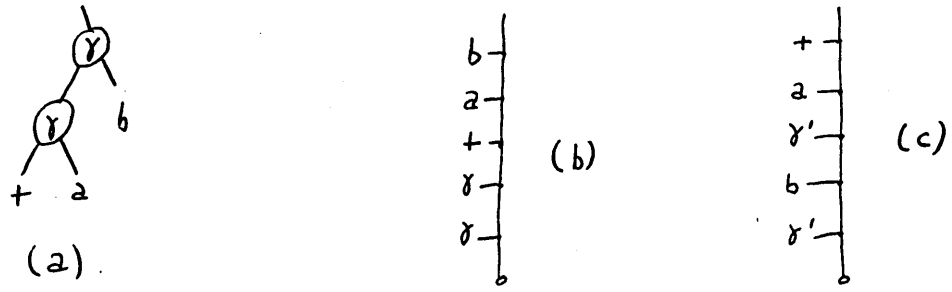


Figure 3.5-16: Trees for "a + b"

The ordering in (b) is required by the fact that the CSE machine, in processing the control item GAMMA, applies the top stack item to the second. Thus the rand must be evaluated first so that it will be "below" the rator on the stack.

Now, consider a new control item, AMMAG, whose effect is to apply the second stack element to the first, deleting both and leaving the result in the stack. Figure 3.6-16(c) shows the control for (3.5-22) using AMMAG (written γ') instead of GAMMA. Clearly the order is different, being from left to right, but it is no closer to the desired result of being unpredictable.

What we need is a way to use randomly either GAMMA or AMMAG. But this is easy. We first modify the CSE machine to handle both GAMMA and AMMAG, and then alter FF so that, on encountering a GAMMA mode in the standardized tree, it makes a random decision between GAMMA and AMMAG in the control structure, adjusting its output accordingly. As suggested in the quote from the PAL Manual, this choice could be based on the weather forecast, time of day, or some other phenomenon independent of the program being translated. The effect then is as desired, and the result of gedanken evaluation of any program which requires a particular order of evaluation is undefined, in the sense that we have not defined it. To implement this decision, we might call a function "Choice nil" in FF which randomly returns true or false.

Several more changes are needed to avoid over-specifying order of evaluation. Control item GUA should be selected randomly instead of AUG to make undefined the order of evaluation of operands of aug. Finally, there is the problem, alluded to on page 3.2-120, of definitions such as

```
let (x, x = 1, 2) in M                                    (3.5-23)
```

The mechanism so far described specifies that bv-lists are processed from left to right, because of the way we have written the recursive function Q which is part of Decompose. (See Figure 3.5-14 on page 3.5-178.) To circumvent this problem, we could provide randomness in Decompose. We leave the details as an exercise for the reader.

The previous decision raises an interesting question, arising from the fact that subexpressions in R-PAL exhibit referential transparency: it is just not possible to write an R-PAL program whose value is dependent on order of evaluation. So as better to see this, suppose there were a function Next of no arguments which, on each call, returns an integer one larger than its value on the previous call. (For the sake of concreteness, suppose it returns 0 on its "first" call.) Then the value of the expression

```
2 * Next nil  +  3 * Next nil                             (3.5-24)
```

depends very much on the order of evaluation of the operands of "+". If the order is left to right, we would have (2*0 + 3*1) or 3, whereas otherwise it would be (2*1 + 3*0) or 2. Now let us see why Next cannot be written in R-PAL. If it could, we would have, in some context, the fragment

```
let Next() = N in M                                       (3.5-25a)
```

which would desugar as

```
(λ Next. M) [λ(). N]                                      (3.5-25b)
```

Since each application of Next in M results in evaluation of N in the same environment (the one existing when this fragment is encountered), the value returned is always the same. There is just nothing Next can do on one call that will have an effect on its next call. What is needed of course is an assignment statement: In L-PAL we could write

```
def          k = 0
             within
     Next() =                                             (3.5-25c)
          k := k + 1;
          k - 1
```

to get the desired effect. The semantics of programs such as this are discussed in Chapter 4.

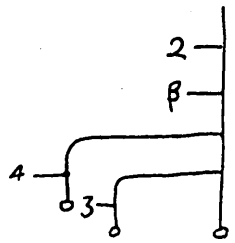
Since we have just shown that it is not possible to write in R-PAL a program whose value is dependent on the order of evaluation used by the evaluating mechanism, one might question the need for the previous discussion. Our purpose in introducing the issue of overspecification of order of evaluation has been to lay the groundwork for further discussion in connection with the L-PAL evaluator, where it is unquestionably relevant. We do not pursue this point further here.

The Logical Bootstrap: Our purpose in this section has been to define formally the semantics of R-PAL by showing a translator and evaluator for R-PAL. Let us reexamine our objectives with an eye to assessing our success.

The intent is that a reader be able to deduce the meaning of any R-PAL program from study of Translate and Evaluate. To make this decision more concrete, let us ask how one might deduce the semantics of the R-PAL expression

```
2 -> 3 | 4                                    (3.5-26)
```

The user must himself himself render this expression into form suitable as input to Translate, and then apply Translate to it. This latter can be done in one of three ways: hand simulate the operation of Translate, use normal order reduction or run the program on a computer. The first is of course what the reader has been doing in preparing input to the blackboard evaluator, the second is (as already mentioned) an intellectually vacuous exercise in virtuosity and perseverance, and the third seems a pleasant alternative if available. By hand simulation, we get



which we can submit to Evaluate. We again have the same three methods to choose from, and we assume we hand simulate.

The interesting question here is the interpretation of the conditional in which the boolean arm is not a truthvalue. The problem shows up in Do_conditional (3.5-12h) where we must evaluate

```
2 -> 3d | 2d                                  (3.5-27)
```

We seem to be in a logical loop: We have found that the semantics of 2 used as the Boolean arm of a conditional is that of 2 used as the Boolean arm of a conditional, a not-helpful result. This is typical of the flaw in attempts to "explain" a language by writing its interpretor in itself.

In the PAL case we are able to break the loop by appeal to the axiomatization of the universe of discourse. Using the interpretation of (3.3-3) and (3.3-4), we see that (3.5-27) is sugaring for

```
Q 2 3d 2d                                     (3.5-28)
```

Since Q does not include 2 in its domain of definition, we can conclude that (3.5-28) is undefined. Hence (3.5-27) is undefined, and so we conclude that our original problem, (3.5-26), is undefined.

The word "undefined" as used in this sort of discussion has interesting philosophical implications: What does it mean to say, of a particular PAL construct, that its semantics is "undefined"? If the decision about meaning of the construct is made by appeal to the definition of the universe of discourse, it is apparently the case that the designers of the language, who after all are the ones who designed the universe of discourse, have deliberately chosen to leave the construct without meaning. Presumably then an implementation of the language would give a "run error" or similar diagnostic in the event of an attempt to use the construct. (An attempt to execute a program including (3.5-26) would cause a "run error" diagnostic in any existing PAL implementation.)

But there is another aspect to "undefined". We have claimed that any program whose succesful execution depends on a particular order of evaluation is "undefined". It is clear that an attempt to run such a program, either on some PAL implementation or through the gedanken evaluator, would produce some answer, although we have shown how to build the gedanken evaluator so that the answer cannot be predicted. But it seems most unlikely that any implementor could detect this problem and provide a diagnostic. Thus there is a distinction between saying that a construct has no definition and refraining from saying what the definition is. We thus conclude the following: When we say that something is undefined, we mean no more or less than that we have not defined it. We make no claims about what will happen to the user who uses it. Perhaps he will get a diagnostic, perhaps he will just get wrong answers, perhaps the computer will blow up. The language specifier can say no more.

Detailed Description of the Gedanken Evaluator

The many details of the programs presented which have been omitted for the sake of expository convenience are presented in this section. The resulting programs are given in enough detail that they will run on a PAL implementation. (They do.) Our purpose is to be sure that we have completly specified the gedanken evaluator, and providing enough detail to permit execution on a computer helps to insure this. We leave undefined only the function Error, whose effect when applied is to terminate the evaluation of the program of which it is a part. Discrepencies between the programs shown here and those shown earlier in this chapter should be resolved in favor of those shown here, since these are listings of programs that have run on a computer. In some cases the programs shown earlier have been simplified somewhat for expository convenience.

It is now the time to say a few words about the representational issues which heretofor we have ignored. Let us consider the PAL expression

```
x eq 0 -> y | f 2                    (3.5-29)
```

which we considered briefly in (3.5-16) and (3.5-17). To facilitate writing PAL structures to represent the tree form of expressions such as this one, we introduce a set of tagging functions. For the three tags needed by this

example, we have

```
def ARROW_ x y z  =  Tag ARROW (x, y, z)
and BINOP_ x y z  =  Tag BINOP (x, y, z)          (3.5-30)
and AP_ x y       =  Tag AP (x, y)
```

The convention is that we use names made of upper case letters for tags, and append an underscore to get the name of the corresponding tagging function. Note that the taggers are curried. In terms of the taggers just defined, (3.5-29) can be rewritten as

```
ARROW_ (BINOP_ x 0 eq) y (AP_ f 2)                (3.5-31)
```

which is certainly easier to read than is (3.5-17). The complete set of tagging functions is listed in the appendix to this section.

(3.5-31) is still not correct PAL, and to correct it we must decide how to represent constants, variables and functors in syntax trees input to Translate. We make the following arbitrary decisions: Constants are represented by 2-tuples whose first component is the tag CONSTANT and whose second component is the value of the constant. Variables are represented by 2-tuples whose first component is the tag VARIABLE and whose second component is the name of the variable as a string. Since functors may appear in input only as the right son of a BINOP or UNOP node, they need no tags and we represent them by themselves. In writing PAL input to translate, we would for example represent "eq" by

```
fn x. fn y. x eq y
```

We now show a correct representation of (3.5-29). We assume that our task is to write the definition of a variable Data whose value is the tree form of (3.5-29), and we have

```
def Data =
   ARROW_
   (  BINOP_
      (  VARIABLE, 'x'  )
      (  CONSTANT, 0  )
      (  fn x. fn y. x eq y  )                    (3.5-32)
   )
   (  VARIABLE, 'y'  )
   (  AP_
      (  VARIABLE, 'f'  )
      (  CONSTANT, 2  )
   )
```

Note the consistent indenting used, in which each right parenthesis is either on the same line as the left parenthesis which it matches, or else is vertically beneath it. Another way to write this is

```
def Data =
   ARROW_
   (  EQ_ x 0_  )
   y
   (  AP_ f 2_  )

   where
      (     0_ =  CONSTANT,  0
        and 2_ =  CONSTANT,  2
        and f  =  VARIABLE, 'f'
        and x  =  VARIABLE, 'x'
        and EQ_ x y = BINOP_ x y (fn s. fn t. s eq t)
      )
```

(3.5-33)

This latter form is particularly advantageous for expressing long programs.

```
//                    PRELIMINARY DEFINITIONS

//  Preliminary definitions for the evaluator.


//  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *

//  Selectors and constructors for the stack and control.

def t(x, y) = x  // Top of stack or control.

and r(x, y) = y  // Rest of stack or control.

and Push(x, s) = x, s  // Put new item on stack or control.

def rec Prefix(x, y) =  // Put control x at top of control y.
        Null x -> y
     |  Push(t x, Prefix(r x, y) )


def r2 x = r(r x)  // Rest of (rest of (stack or control)).

and r3 x = r(r(r x))  // Rest of (rest of rest).

and 2d x = t(r x)  // Second element of stack or control.

and 3d x = t(r(r x))  // Third...

def Empty_stack = nil  // The empty stack.


//  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *

//  Tagger and tag-checkers for structures.

def Tag n s  =  s aug n  //  Tag structure s with tag n.

and Is_tag s n  =  //  Does structure s have tag n?
     Istuple s  ->  n eq s(Order s)  |  false

and Get_tag s  =  s(Order s)  //  Return the tag of s.

and Sons s  =  Order s - 1  //  Return number of sons of s.
```

```
//  Selectors, predicates and constructors for lambda-expressions
//  and lambda-closures.

def LAMBDA = '_lambda'  // Tag for lambda-expressions and closures.

def bV x = x 2       //  Select bv-part of a lambda-exp or closure.
and Body x = x 3     //  Select body part...
and Env x = x 4      //  Select environment part...

def       Test(x, n) =
               Istuple x
               -> Order x eq n
                  -> Isstring(x 1)
                     -> x 1 eq LAMBDA
                     |  false
                  |  false
               |  false
          within

     Is_lambda_exp x = Test(x, 3)
and  Is_closure x = Test(x, 4)

def Cons_lambda_exp(bV, Body)  =  //  Construct a lambda-expression.
     LAMBDA, bV, Body

and Cons_closure(L_exp, Env)   =  //  Construct a lambda-closure.
     LAMBDA, bV L_exp, Body L_exp, Env
```

```
//  Items and predicates for control structure and stack.

def GAMMA       = '_gamma'
and BETA        = '_beta'
and CONSTANT    = '_constant'
and VARIABLE    = '_variable'
and AUG         = '_aug'
and TUPLE       = '_tuple'      // Used only in stack.
and ETA         = '_eta'        // Used in stack for recursion.
and RETURN      = '_return'


def       Test(x, y) =
             Istuple x
             -> Order x eq 2
                -> Isstring(x 1)
                   -> x 1 eq y
                   |  false
                |  false
             |  false
          within

     Is_constant x  = Test(x, CONSTANT)
 and Is_variable x  = Test(x, VARIABLE)
 and Is_eta x       = Test(x, ETA)

 and Is_tuple x =
        Test(x, TUPLE)  ->  true  // Is it a constructed tuple?
     |  Test(x, CONSTANT)  ->  Null(x 2)  // Is it nil?
     |  false  // Neither.

 and Is_identifier x =  // Is x a constant or a variable?
     Test(x, CONSTANT) or Test(x, VARIABLE)


def Same_var(x, y)  =  // Are x and y the same variable?
     x 2 eq y 2

//  Call for Y_VAR is produced in Translate for rec-defs.

def Y_NAME = 'Y#'  // The name of "Y".

def Y_VAR  =
     VARIABLE, Y_NAME

and NIL  =  // Used in ST for COMMAs.
     CONSTANT, nil

def Is_Y x =
     Isstring x -> x eq Y_NAME | false
```

```
//  Tags for abstract syntax tree.

def TEST      = '_test'       // test ... ifso ... ifnot ...
and ARROW     = '_arrow'      // ... -> ... | ...
and AP        = '_ap'         // functional application
and FN        = '_fn'         // lambda
and EQUAL     = '_equal'      // definition
and WITHIN    = '_within'
and REC       = '_rec'
and FF        = '_ff'         // function form definition
and AND       = '_and'        // 'and' definition
and COMMA     = '_comma'      // tuple maker
and LET       = '_let'
and WHERE     = '_where'
and BINOP     = '_binop'      // infix binary operator
and UNOP      = '_unop'       // prefix unary operator
and PERCENT   = '_percent'


//  Taggers for tags in abstract syntax tree.

def ARROW_ x y z     = Tag ARROW (x, y, z)
and AP_ x y          = Tag AP (x, y)
and FN_ x y          = Tag FN (x, y)
and LET_ x y         = Tag LET (x, y)
and WHERE_ x y       = Tag WHERE (x, y)
and EQUAL_ x y       = Tag EQUAL (x, y)
and WITHIN_ x y      = Tag WITHIN (x, y)
and REC_ x           = Tag REC (nil aug x)
and FF_ x y          = Tag FF (x, y)
and AUG_ x y         = Tag AUG (x, y)
and BINOP_ x y z     = Tag BINOP (x, y, z)
and UNOP_ x y        = Tag UNOP (x, y)
and PERCENT_ x y z   = Tag PERCENT (x, y, z)

def                  // subsudiary function for n-ary taggers
          rec Q k T f =
               k eq 0 -> f T
               | (fn x. Q (k-1) (T aug x) f)
          within
     COMMA_ n  =  Q n nil (Tag COMMA)
 and
     AND_   n  =  Q n nil (Tag AND)


//  Taggers for standardized syntax tree.

def GAMMA_ x y       = Tag GAMMA (x, y)
and BETA_ x y z      = Tag BETA (x, y, z)
and LAMBDA_ x y      = Tag LAMBDA (x, y)
```

```
//  Some useful functions for Transform.

def Value_of x  =  // Evaluate a control element, to put it on stack.
     x

and Val_of x  =  // De-tag a stack element, to get its value.
     x 2

def Apply x y  =
     let t = (Val_of x) (Val_of y)
     in
     CONSTANT, t

and Augment_tuple x y  =  //  Augment x with y.
     Is_tuple x  ->  (TUPLE, Val_of x aug y)
     |  Error 'first argument of Aug not a tuple'

//
```

```
//                          E N V I R O N M E N T

//  The following function is used in applying a lambda-closure.
//  The names on the (possibly structured) bv-part 'Names' are
//  added to the environment 'Env', associated with the corres-
//  ponding part of 'Values'.  The new environment is returned as
//  the value of the function.

def rec Decompose(Names, Values, Env)  =
     test  Is_variable Names  // Is it a single variable?
     ifso  (Names, Values, Env)  // Yes, so add it to environment.
     ifnot  // Check conformality.
     test  Is_tuple Values
     ifnot Error 'conformality failure'  // Tuple applied to scalar.
     ifso
     test  Order Names eq Order (Val_of Values)
     ifnot Error 'conformality failure.'  // Differing tuple lengths.
     ifso  // Process a multiple-bv part.
       (  Q 1 Env
          where rec Q n e =
                  n > Order Names  ->  e
                | Q (n+1) ( Decompose(Names n, (Val_of Values) n, e) )
       )



//  Define primitive environment, and provide function to look
//  up variables in the environment.

def PE =  // The primitive environment.
     Y_VAR, Y_NAME,  // for recursion
     nil

and Lookup(Var, Env) =  // Look up a variable in the environment.
     L Env  // Start looking in Env.
     where rec L e =
             Null e  ->  Error 'variable not found in environment'
           | Same_var(Var, e 1)  ->  e 2  // Found.
           | L (e 3)  // Keep looking.
```

```
def rec D x =  // Standardize a definition.
     let Type = Is_tag x
     in
        Type EQUAL  ->  x  // Already OK.
     |  Type WITHIN
          -> ( let u, v = D(x 1), D(x 2)
               in
               EQUAL_ (v 1) ( AP_ (FN_ (u 1) (v 2)) (u 2) )
             )
     |  Type REC
          -> ( let w = D(x 1)
               in
               EQUAL_ (w 1) ( AP_ Y_VAR (FN_ (w 1) (w 2)) )
             )
     |  Type FF
          -> ( let rec Q k T =
                       k < 2  ->  T
                       | Q (k-1) (FN_ (x 1 k) T)
               in
               EQUAL_ (x 1 1) (Q (Order(x 1)) (x 2))
             )
     |  Type AND
          -> ( let rec Q k S T =
                       k > Sons x  ->  (S, T)
                       |  ( let w = D(x k)
                            in
                            Q (k+1) (S aug w 1) (T aug w 2)
                          )
               in
               let L, R = Q 1 nil nil
               in
               EQUAL_ L (Tag COMMA R)
             )
     |  Error 'improper node found in D'
```

```
def rec ST x  =  // Standardize abstract syntax tree.

     let Type = Is_tag x
     in
        Is_identifier x  ->  x
     |  Type BETA or Type TEST or Type ARROW
          ->  BETA_ (ST(x 1)) (ST(x 2)) (ST(x 3))
     |  Type FN
          ->  LAMBDA_ (x 1) (ST(x 2))
     |  Type COMMA
          -> ( Q 1 NIL
               where rec Q k t =
                     k > Sons x  ->  t
                   |  Q (k+1) ( AUG_ t (ST(x k)) )
             )
     |  Type PERCENT
          -> GAMMA_ (x 2) ( AUG_ (AUG_ NIL (ST(x 1))) (ST(x 3)) )
     |  Type LET
          -> ( let w = D(x 1)  // Standardize the definition.
               in
               GAMMA_ ( LAMBDA_ (w 1) (ST(x 2)) ) (ST (w 2))
             )
     |  Type WHERE  ->  ST(LET_ (x 2) (x 1))
     |  Type AP  ->  GAMMA_ (ST(x 1)) (ST(x 2))
     |  Type BINOP
          ->  GAMMA_ ( GAMMA_ (CONSTANT, x 3) (ST(x 1)) ) (ST(x 2))
     |  Type UNOP
          ->  GAMMA_ (CONSTANT, x 2) (ST(x 1))
     |  Type AUG  ->  AUG_ (ST(x 1)) (ST(x 2))

     |  Error 'improper node found in ST'
```

```
//  The function FF flattens a standardized tree into a
//  control structure.

def rec FF(x, c)  =  // Flatten standardized tree x onto control c.


     let Type = Is_tag x
     in
        Is_identifier x -> (x, c)
     |  Type LAMBDA
          -> ( let Body = FF( x 2, nil )
               in
               Cons_lambda_exp(x 1, Body), c
             )
     |  Type BETA
          -> ( let TA = FF(x 2, nil)  // True arm.
               and FA = FF(x 3, nil)  // False arm.
               in
               FF( x 1, (BETA, (FA, (TA, c))) )
             )
     |  Type GAMMA -> FF( x 2, FF( x 1, (GAMMA, c) ) )
     |  Type AUG -> FF( x 2, FF( x 1, (AUG, c) ) )
     |  Error 'improper node found in FF'


//  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *

def Translate Program  =  // The routine that does all the work.
     FF ( ST Program, nil )
```

```
//  State transformations for the right-hand evaluator.


def Do_return(C, S, E) =
     r C, Push(t S, r2 S), 2d S

and Eval_constant(C, S, E) =
     r C, Push(Value_of(t C), S), E

and Eval_variable(C, S, E) =
     r C, Push( Lookup(t C, E), S ), E

and Eval_lambda_exp(C, S, E) =
     let New_S = Cons_closure(t C, E)
     in
     r C, Push(New_S, S), E

and Do_conditional(C, S, E) =
     let Selected_arm = ( Val_of(t S) -> 3d | 2d ) C
     in
     Prefix(Selected_arm, r3 C), r S, E

and Do_aug(C, S, E) =
     let New_S = Augment_tuple (t S) (2d S)
     in
     r C, Push(New_S, r2 S), E

and Apply_closure(C, S, E) =
     let Rator = t S  // The closure being applied.
     in
     let New_C = Prefix(Body Rator, Push(RETURN, r C))
     and New_S = Push(E, r2 S)
     and New_E = Decompose(bV Rator, 2d S, Env Rator)
     in
     New_C, New_S, New_E

and Apply_constant(C, S, E) =
     let V = Apply (t S) (2d S)
     in
     r C, Push(V, r2 S), E

and Apply_tuple(C, S, E) =
     let New_S = Apply (t S) (2d S)
     in
     r C, Push(V, r2 S), E

and Apply_Y(C, S, E) =
     let V = ETA, 2d S
     in
     let New_S = Push(2d S,  Push(V, r2 S) )
     in
     C, New_S, E

and Apply_eta(C, S, E) =
     Push(GAMMA, C), Push(t S 2, S), E
```

```
//  Main program.  Transform does one step in the evaluation,
//  and Evaluate is the driver for it.


def Transform(C, S, E)  =  //  Do a single step.
     let A = C, S, E
     and x = t C  // Top of control.
     in
        Is_constant x     -> Eval_constant A
     |  Is_variable x     -> Eval_variable A
     |  Is_lambda_exp x   -> Eval_lambda_exp A
     |  x eq BETA         -> Do_conditional A
     |  x eq AUG          -> Do_aug A
     |  x eq RETURN       -> Do_return A
     |  x eq GAMMA
          -> ( let r = t S  // The rator.
               in
                  Is_closure r  ->  Apply_closure A
               |  Is_constant r ->  Apply_constant A
               |  Is_tuple r    ->  Apply_tuple A
               |  Is_Y r        ->  Apply_Y A
               |  Is_eta r      ->  Apply_eta A
               |  Error 'improper rator'
             )
     |  Error 'bad control'


def rec Evaluate(C, S, E) =
     Null C  ->  t S  |
     Evaluate(Transform(C, S, E))


def Gedanken_evaluator Program  =
     let Control_structure = Translate Program
     in
     Evaluate(Control_structure, Empty_stack, PE)
```

Chapter 4

ASSIGNMENT, STRUCTURES and SHARING

Heretofore we have been concerned almost exlcusively with functions and with techniques for defining and manipulating them. As we have seen, λ-calculus provides a natural and perspicuous mathematical model for such a study.

From a conceptual point of view, the most important attribute of functions in mathematics is independence of the result of functional application on order of evaluation of the function and its argument. This independence has led to the informal dictum

> to evaluate a functional application, first evaluate the function and its argument (in either order), and then apply the one to the other.

More formally, the Church-Rosser theorem guarantees of the λ-calculus the (weaker) result that all orders of evaluation which produce any value will produce the same value. The pragmatic import is that a programmer need not usually be concerned with irrelevant details of evaluation order when dealing with applicative expressions.

But it is obviously not true that the functional approach to programming is sacrosanct; indeed, pure LISP is the only common programming language in which applicative ideas are fundamental. Most languages (PL/I, for example) are predicated instead on the concept of a computational procedure, by which we mean specification of a computation in terms of a sequence of steps, each of which is to be executed in an order explicitly decreed by the programmer. We use the word imperatives to refer to those linguistic facilities (such as assignments and jumps) which are peculiar to the specification of procedures. The essence of applicative programming is that subexpressions are for the most part of interest for their value, while in imperative programming execution of subexpressions is frequently for effect. Thus we speak of the evaluation of an applicative expression, and of obeying an imperative. A command is a part of a program such as an assignment statement which is of interest primarily for its effect.

Clearly, there is no theoretical basis on which to decide that procedures are less fundamental than functions from a linguistic point of view. Indeed, to be practical it must be conceded that imperatives mesh better with the current state of evolution of actual computers. Why then have we chosen in these notes to treat functions first? Simply stated, the only real justification stems from our belief that applicative expressions provide a better springboard for the study of imperatives than vice versa. In this chapter and the next we use this springboard to deal with imperatives. We concern ourselves with two specific imperatives: the assignment command, and the goto command. The effect of obeying an assignment is to change the value of a variable, and the effect of obeying a goto is to alter what would otherwise be the normal order of execution

of commands. Adding these constructs to PAL, in all of their generality, has profound implications. In this chapter we study in depth the implications of assignment, and we consider goto in Chapter 5.

4.1 New Linguistic Concepts

To gain preliminary insight into imperatives, consider the three following programs, each of which defines the factorial function:

```
def f n =
     g (1, 0)
     where rec g (r, k) =
          k eq n  ->  r  |  g( r*(k+1), k+1 )
```
(4.1-1a)

```
def f n =
     let r, k = 1, 0
     in
     until k eq n do
       (  k := k + 1;
          r := r * k
       );
     r
```
(4.1-1b)

```
def f n =
     let r, k = 1, 0
     in
     L:   if k eq n do goto M;
          k := k + 1;
          r := r * k;
          goto L;
     M:   r
```
(4.1-1c)

The first definition, which is applicative, differs from the one which we have been studying in that it uses a subsidiary recursive function to do the work. It clearly computes factorial, and it is written entirely in R_PAL. By contrast, the second and third definitions each involve a sequence of assignments. The intent when k is not equal to n is first to increment k by one, then to replace the value associated with r by the value of "r*k", and finally to reiterate these steps until k does equal n, at which time r is returned as the value. The colon is a syntactic device which signifies that the identifiers "L" and "M" are labels. It is clear that the order in which the assignments are carried out is vital, whereas in evaluating "r*k" it makes no difference whether "r" or "k" is evaluated first. The program in (4.1-1b) is written in L-PAL, while that in (4.1-1c) uses linguistic facilities available only in J-PAL. In this chapter we explain L-PAL, covering J-PAL in the next.

Our task then is to provide an explanation of the imperative features of PAL. Before proceeding, we observe that the λ-calculus is inadequate to explain these features. Consider

```
let x = 2 in x := x + 3; x                    (4.1-2a)
```

The semantic intent here is to define a variable x initialized to 2, to increment x by 3, and then to return the value of x, 5. Desugaring according to our rules leads to

```
(λx. x := x + 3; x) 2                         (4.1-2b)
```

We develop later a way to interpret expressions such as this one, but we point out here that our existing procedure is deficient. Naive use of λ-reduction on (4.1-2b) would lead to

```
2 := 2 + 3; 2                                 (4.1-2c)
```

which is manifestly absurd. Let us look ahead a bit to see how one might interpret expressions such as (4.1-2b). We would have proceeded in the R-PAL blackboard evaluator (or the R-PAL gedanken evaluator) by evaluating the body in an environment in which x is associated with the ob 2. In the L-PAL evaluator to be described we accomplish the evaluation by finding a place to store a value of x, initializing that place to hold 2, and then evaluating the body in an environment in which x is associated with that place. The assignment statement then changes the contents of that place. The place of course is a cell in a memory.

Chapter Outline: In Chapter 3 we introduced a gedanken evaluator (the CSE machine) for evaluating programs written in the applicative subset of PAL, which subset we now call R-PAL. This machine, which hereafter we call the right-hand evaluator or the R-machine, was defined in terms of a program which (after conceptually straightforward desugaring) was meaningful in terms of a normal order λ-calculus reduction mechanism. In this chapter and the next we introduce two new machines, the left-hand (or L) machine and the jumping (or J) machine. The L-machine, defined in terms of a program meaningful to the R-machine, accommodates assignments and sequences. In turn, the J-machine is defined in terms of a program meaningful to the L-machine, and accommodates goto's and labels.

The advantage of building up a hierarchy of machines in this way is threefold:

- We isolate related sub-classes of important linguistic facilities in such a way that they can be studied separately.
- Each level of the hierarchy can be treated efficiently by exploiting concepts and tools which have already been established.
- The complete set of PAL semantics is reducible to intuitively satisfying axioms.

A disadvantage of the procedure rests in the length of the train of reasoning; we need to remember the substance of each phase of the development in order to treat the next. In balance, however, the approach seems consistent with our definition of programming linguistics as the science of building large semantic structures out of smaller ones.

Generalized Tuples

The gedanken evaluator of Chapter 3, the R-machine, accepts as input a syntax tree representation of a PAL program, produces therefrom a control structure via application of Translate, and then executes a series of CSE state transitions which depend on that control structure. Clearly, each CSE state is itself a structure. It seems fair to conclude that in large part the study of programming linguistics entails the study of structures and their manipulation.

Now, the class of structures and the facilities for manipulating them which we have encountered to date are rather primitive. In particular, the postulates for tuples which we have used thus far encompass only structures that are representable as trees. For example, the structures

(4.1-3a)

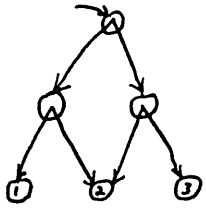
are naturally represented in terms of Augment, the function postulated in Section 2.1, or Aug, its curried version. The structure on the left could be written as

(1, 2, (3, 4)) (4.1-3b)

which we have been regarding as sugaring for

Aug [Aug (Aug nil 1) 2] [Aug (Aug nil 3) 4]

It is a moot point whether or not the structure

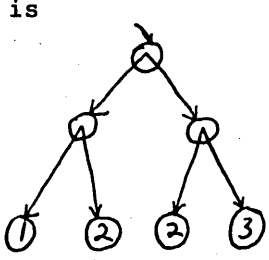


(4.1-4a)

can be specified in R-PAL. Does the expression

(1, x), (x, 3) where x = 2 (4.1-4b)

represent this tree, or is



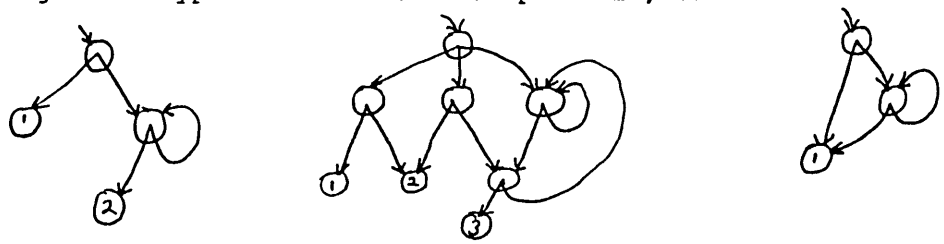
(4.1-4c)

a more appropriate drawing of (4.1-4b)? Nothing we have said so far answers this question, not even the programs in Chapter 3 for the CSE machine. In fact, no R-PAL program can distinguish between (4.1-4a) and (4.1-4c), although

they can be easily distinguished by an L-PAL program. As we see later, the L-PAL expression (4.1-4b) denotes a structure such as that shown in (4.1-4a), and (4.1-4c) might be written as

(1, 2), (2, 3) (4.1-4d)

More general types of structure are possible, such as



(4.1-5)

Such reentrant structures are not just pathological cases but arise naturally in many cases of interest. (We see such structures in the output of the J-PAL version of Translate.) In order to accomodate these more general structures, we introduce into PAL the infix functor aug. Informally, if E denotes a k-tuple, then the PAL expression

E aug F

denotes an object that transforms precisely as does

Aug E F

However, whereas the specification of Aug leaves unanswered such questions as that raised by consideration of (4.1-4), we specify aug so as to answer them.

Our objectives in introducing the functor aug are two-fold: first to enrich the class of structures with which we can deal, and second to increase the efficiency with which we can manipulate them. The problem of efficiency in pure λ-calculus manipulation of structures is intimately related to the "copying" implicit in the reduction of λ-expressions. Consider for example the reduction

$$(\lambda x.\ \sim x \sim x \sim)\ S \xrightarrow{\beta} \sim S \sim S \sim \qquad (4.1\text{-}6)$$

in which S may be an arbitrary AE. We have already pointed out (on page 3.2-117) the enormous gain in efficiency afforded by evaluating S before substituting for the free occurrences of x within the body of the rator. But still another problem remains, namely the need for providing as many copies of the value of S as there are free occurrences of x. If the value of S is represented by only a few bits of information, as in the case where S denotes an integer, then the cost of inserting several distinct copies of this representation when carrying out the reduction is unimportant. On the other hand, when S denotes, say, a 500-tuple, then the cost of maintaining multiple copies cannot be ignored. Thus we are motivated to substitute not a representation of the value of S itself, but instead concise information telling us how to access that representation. In common parlance, such information is called a "pointer": One elects to keep only one copy of the representation, stored away somewhere, and replicate as many copies of pointers to the place of storage as may be necessary.

Addresses and Contents: The possibility of "storing things away somewhere" leads us to the question of the properties of the place where we might do the storing. Such a place to store information we call a memory. We use the term address for the name of a location in a memory where an object may be stored, and we call the storage place itself a cell. We must distinguish carefully between addresses and obs. However, once this distinction is made, a whole new panoply of manipulatory procedures becomes immediately available. The situation is illustrated in Figure 4.1-1, in which we use circles to indicate addresses and dots with directed arcs emanating therefrom to indicate tuples whose components are addresses. Clearly, from Figure 4.1-1a we can obtain

Figure 4.1-1b by changing the contents of x to be the contents of (x 1); or

Figure 4.1-1c by changing the contents of (x 3) to be the contents of (x 1); or

Figure 4.1-1d by changing the contents of (x 2) to be the one-tuple "nil aug (x 3)"; or

Figure 4.1-1e by changing the contents of (x 2) to be the one-tuple "(nil aug x)".

In L-PAL these transformations may be effected respectively by the assignment commands

```
x   := x 1                    (4.1-6a)
x 3 := x 1                    (4.1-6b)
x 2 := nil aug (x 3)          (4.1-6c)
x 2 := nil aug x              (4.1-6d)
```

It is clear from Figure 4.1-1 that the concept of addresses not only affects efficiency but also introduces the notion of sharing in structures: Specifically, we say that two constituents of a structure share if and only if they designate the same address. In Figure 4.1-1 we rely simply on location relative to the sheet of paper to indicate an address. That is, a location on the page corresponds to an address in a memory.

The flexibility afforded by addresses, however, is not gained without cost in intellectual overhead. Study of the assignment commands in (4.1-6), for example, indicates a forfeiture of contextual independence (cf page 2.3-67). For example, in equation (4.1-6b) the expression "x 3" is to be interpreted as the address which is the third component of x, whereas the (syntactically equivalent) expression "x 1" is to be interpreted as the contents of the cell whose address is the first component of x. Thus the meaning of such expressions depends on the context in which they occur, in this case on whether they occur to the left or to the right of the assignment operator ":=". This contextual dependence gives rise to the nomenclature "L-value" and "R-value" to indicate whether one means an address or its contents, respectively. But the contextual dependence in PAL is not always so simple, as careful study of (4.1-6c) indicates: Here comparison with Figure 4.1-1d reveals that both "x 2" and "x 3"

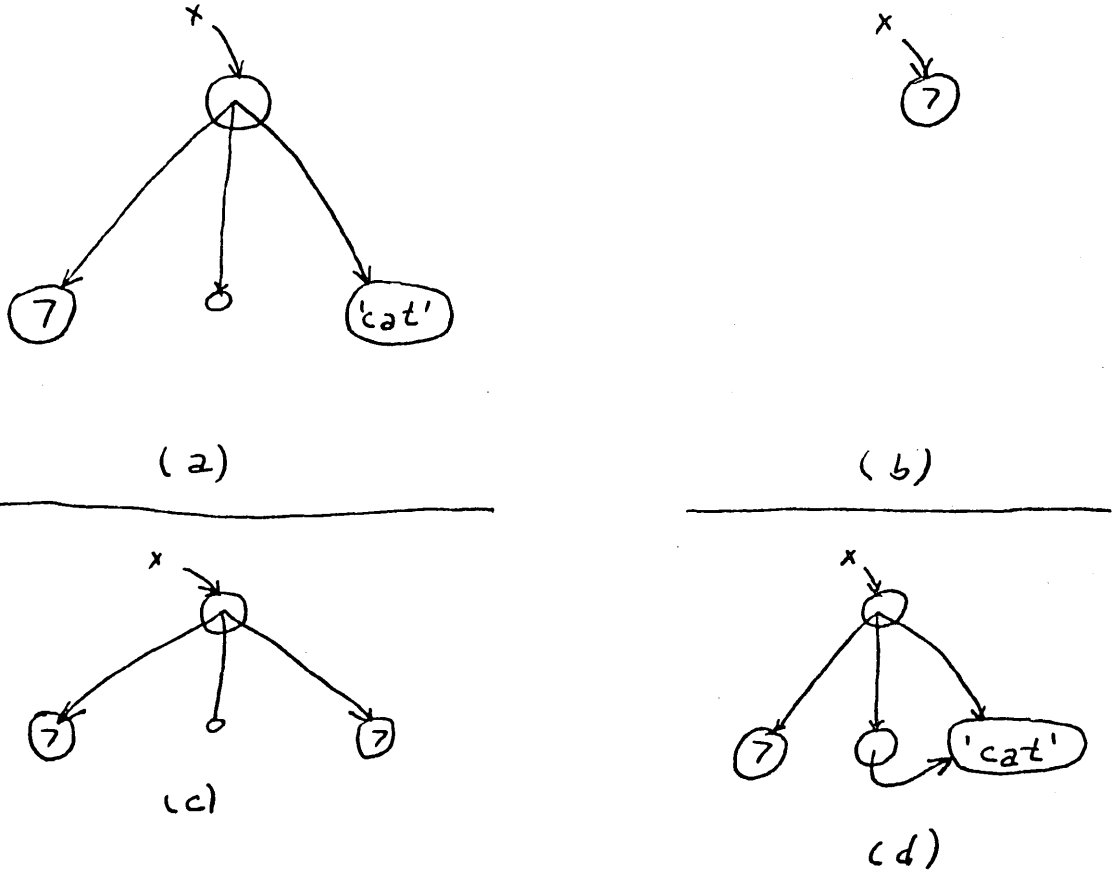


Figure 4.1-1

are to be interpreted as addresses, even though they occur on opposite sides of the ":=". We see later that this fact hinges on a design decision concerning the semantics of the infix operator "aug" -- it is not true in PAL that "aug" is just an infix version of the curried function "Aug".

The Functors "aug" and "$": The semantics of PAL's infix functor aug is as follows: If E denotes a k-tuple and F is any expression, then the expression

E aug F

denotes that (k+1)-tuple whose first k components share with the k components of E, and whose (k+1)-st component shares with F. This is not the only way we might have defined aug, and other possibilities are suggested by Figure 4.1-2. Consider the two tuples t and s shown in Figure 4.1-2a, and the problem of extending t to have s as a third component. The simplest procedure would be to draw a third arc, as indicated by the (crossed-out) dashed line; but this operation has the undesirable effect of altering structure t, so that if "aug" were defined this way the value of

[Order t eq 2 -> t 1 1 | t 1 2] + [(t aug s) 3 1]

would be 6 or 5, depending upon whether "t aug s" were evaluated before or after "Order t". This violation of invariance to order of evaluation seems unwarranted, and was rejected in the design of PAL.

Alternatively, as illustrated in Figure 4.1-2b, we could make complete copies of t and s, which is presumably the operation implied by the function "Aug". This leaves t and s uneffected, but implies possible monumental inefficiencies which we have already discussed. Of course copying structures such as those of (4.1-5) is a distinctly non-trivial task.

An example of the use of the infix operator "aug" in PAL is illustrated in Figure 4.1-2c. Assume that t and s are each associated with addresses, and that t's address references a memory cell containing a k-vector of addresses. Then, to evaluate "t aug s":

1. Form a (k+1)-vector of addresses. (4.1-7)
2. Copy the k addresses specified by t into the first k components of that vector.
3. Copy the address of s into the (k+1)-st component.

We say that "aug" evaluates its left argument in R-mode, and its right argument in L-mode.

It may happen that the left argument of "aug" is already an R-value, as in

nil aug s

In this case step 2 of (4.1-7) is to be omitted. Alternatively, it may happen that the right argument of "aug" is an R-value, as in

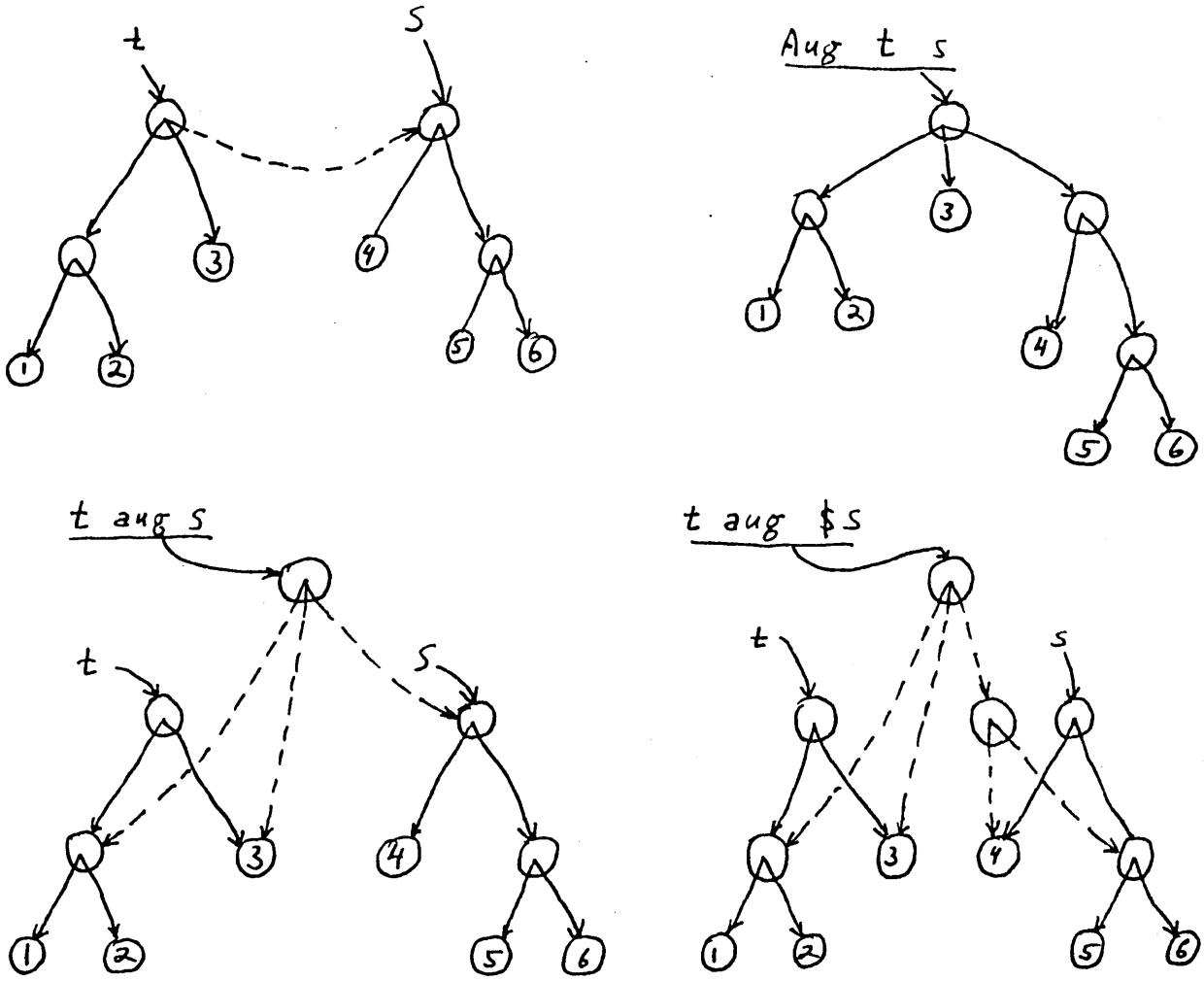


Figure 4.1-2

t aug 7

In this case we are to obtain a new address, store the given R-value into it, and use this address as the (k+1)-st component in step 3 of (4.1-7). In defining the L-machine we must be careful that transfer functions from L-values to R-values or from R-values to L-values are executed automatically whenever the context demands them.

The sharing effects of "aug" are evident in Figure 4.1-2c: The first two components of "t aug s" share with the corresponding components of t, and the third component shares with s. But it may happen that sharing with s is undesirable. In this case, a programmer may invoke PAL's "unsharing" operator "$", as illustrated in Figure 4.1-2d. "$" is a prefix operator which, when applied to an address, extracts its contents (in this case, a 2-vector of addresses). Thus "$" is simply a transfer function from an L-value to an R-value, defined to be nugatory if its argument is already an R-value. Alternatively, we may think of "$" as making a one-level copy of its argument. (The mark "$" is mnemonic for a crossed-out "S", standing for "unshare".)

The distinction between an address and its contents (i.e. between L-values and R-values) is fundamental to the concept of structures in almost every programming language which deals with them. The design decision in PAL is to associate an address implicitly with every node of a structure: The effect is to permit every component of a tuple to be updated by an assignment command. An alternative design decision would have been to associate a node with an address only under explicit instruction by the programmer, in which case only nodes so designated would be updatable. Although in some sense less flexible, this alternative appears to have an advantage in the readability of programs which do not entail pervasive updating. With either alternative, a little thought should make it clear that PAL's "aug" and the postulated function "Aug" are indistinguishable in the absence of the assignment command.

Memories

A major part of the task of specifying L-PAL's semantics is explication of the effects of sharing. To do this we find it useful to add a memory to our evaluating mechanism, as a fourth component. Although for the purposes of blackboard evaluation we can be fairly casual about the properties of memories, we must be more formal when it comes to specification of the L-PAL gedanken evaluator. Our objective now is to specify the memory to be used in the gedanken evaluator.

There are two obvious ways to proceed: We can either postulate memories as abstract objects in the universe of discourse, or we can represent them by obs already available to us. Before investigating each of these possibilities in turn we consider intuitively what properties we want. We think of a memory as a collection of cells. Each cell has an address which identifies it and a contents which is any ob. Given any memory, we can always find a second memory just like it but with another cell. (This property is rather a departure from

the real world.) We can find the contents of any cell in a memory, and given any memory we can create another memory just like it except that one cell has a different contents.

Postulated Memories: Although memories are rather different sorts of things from strings or integers, there is no reason why we cannot use the same sort of technique to put memories into the universe of discourse that we used earlier with strings. The fact that we start with less intuition about memories than we had about strings makes axiomatizing memories all the more useful: We are forced to be precise.

In choosing the postulates, we are guided more by a desire to model the characteristics of a sheet of paper than by a need to mimic the characteristics of an actual computer memory. Specifically, we think of the paper as being marked off in squares, each of which (like a cell) may contain an ob. Since each square of paper is distinct, we do not envision the possibility that a square might contain another square, although it might contain the name (i.e., address) of one. By analogy, we do not permit a cell to contain another cell. In order to accommodate tuples, however, we do elect to permit a cell to contain a vector of addresses. These considerations lead us to adopt the following postulates for memories. We assume a universe of discourse $\mathcal{U}$ and a set $\mathcal{A}$ of addresses, of which we require only that they be distinguishable. We then have the

Definition: A memory system $\{\mathcal{M}, \mathcal{A}, \text{Contents}, \text{Extend}, \text{Update}\}$ over a set $\mathcal{U}$ is composed of a set $\mathcal{M}$ of memories and a set of $\mathcal{A}$ of addresses, along with the three functions Contents, Extend and Update, such that the following hold:

(a) There is a function Contents: $\mathcal{M} \otimes \mathcal{A} \rightarrow \mathcal{U}$. The address σ is said to be encompassed by memory M if

$$\text{Contents}(M, \sigma)$$

is defined.

(b) There is a memory M# that encompasses no addresses at all. It is called the empty memory.

(c) There is a function Extend: $(\mathcal{M} \otimes \mathcal{U}) \rightarrow (\mathcal{M} \otimes \mathcal{A})$ such that if $M \in \mathcal{M}$ and $x \in \mathcal{U}$, and if

$$\text{Extend}(M, x) = (M', \sigma)$$

then σ is not encompassed by M and, for any $b \in \mathcal{A}$,

$$\text{Contents}(M', b) = \begin{cases} \text{Contents}(M, b) & \text{if } b \neq \sigma \\ x & \text{if } b = \sigma \end{cases}$$

(d) There is a function Update: $\mathcal{M} \otimes \mathcal{A} \otimes \mathcal{U} \rightarrow \mathcal{M}$. such that if M is a memory, σ an address encompassed by it, and $x \in \mathcal{U}$, then the memory M' denoted by

Update(M, σ, x)

satisfies, for any address b,

$$\text{Contents}(M', b) = \begin{cases} \text{Contents}(M, b) & \text{if } b \neq \sigma \\ x & \text{if } b = \sigma \end{cases}$$

(e) The set $\mathfrak{M}$ of memories is the closure of the function Extend over the empty memory M# and the universe of discourse Ω.

Clearly the set of addresses is arbitrarily large, since we can use Extend as often as we like and each use must yield a "new" address. The essence of Extend is that it can always find still another address, distinct from any already encompassed by the memory supplied to it. These memories differ in another important way from those of real computers, in that the contents of a cell can be any ob whatsoever. In real computers, a cell is usually some specified size, such as 32 bits, and it is just not possible to store a 20-tuple in one. A problem to be solved by any implementer of PAL is to represent memories such as those just described in the memory of a real computer.

Note the implications of the fact that the operation Update specifies one memory in terms of another one, just as the operation Succ specifies one number in terms of another. But Succ does not "destroy" its operand, and neither does Update. In this regard also our class of abstract memories differs significantly from the memory of an actual computer. In the latter case, a change in the contents of a memory cell destroys the previous contents of that cell: the transformational analogy is to erasing a square of paper and rewriting in it, with only one piece of paper ever being of interest. By contrast, in the abstract case the transformational analogy is to having a sheaf of papers each with slightly different inscriptions, and to specifying one sheet in terms of another by specifying how it differs in a particular square.

In our L-machine, the role of memories is meta-linguistic. That is to say, we elect not to include memories as objects in the domain of discourse of the programmer. Instead, we use memories in the evaluator to represent objects (such as tuples and programmer-defined functions) which the programmer is allowed to manipulate. The rationale behind this philosophical decision is simple: By constraining the programmer, we bound him away from certain well-defined but disastrous transformations which would otherwise be legal. For example, in defining the L-machine we discipline ourselves to reflect reality by never requiring access to more than one memory at a time. Specifically, we discipline ourselves not to write expressions such as

```
Contents(m, a) + Contents(n, a)
where n = Update(m, a, 5)
```
(4.1-8)

in which "a" is an address, and "m" and "n" denote different memories. Such expressions are meaningful in the abstract, but are inconsistent with a picture of the real world wherein each instantaneous memory configuration represents a single distinct abstract memory. In terms of our analogy between memories and sheets of paper, we always "throw away" one sheet of paper as soon as we have

defined another.

Definition by Representation: We are now in a position to define L-PAL's tuples. Formally, we replace the tuple axioms of section 2.2 by the simple definition

A tuple is a vector of addresses in a memory.

Thus tuples are now defined indirectly by representation in terms of memories, which in turn are defined by axiom. Rather than being stated explicitly, the properties of tuples follow implicitly from the properties of memories and the operations to which memories are amenable. Consider for example the structure shown in Figure 4.1-1e. We can describe the associated tuple as being stored in the memory cell with address σ_1, where

$$\begin{aligned} \sigma_1 &\sim (\sigma_2, \sigma_3, \sigma_4) \\ \sigma_2 &\sim 7 \\ \sigma_3 &\sim \text{nil aug } \sigma_1 \\ \sigma_4 &\sim \text{'cat'} \end{aligned} \qquad (4.1\text{-}9a)$$

Here our writing indicates that the cell with address σ_1 contains the 3-vector shown, cell σ_2 contains 7, etc. A possible PAL program denoting this structure is

```
let x = 7, nil, 'cat'
in
x 2  :=  nil aug x;
x
```
(4.1-9b)

There are of course other possible PAL programs.

It is important to realize that definition by representation is equally as valid as definition by axiom. Indeed, the semantics of most programming languages is defined by representation, in the sense that the implementation in an actual computer is the principle definition. Moreover, even in principle, recourse to definition by representation entails no loss of power: It is straightforward to show that the natural numbers are representable by λ-expressions with no free variables, and conversely Gödel has shown that all recursively definable functions are representable by the natural numbers. It follows that one's choice of definitional procedure must rest on questions of convenience and taste, rather than on more fundamental questions of mathematical validity.

There is, however, one fundamental question which deserves comment, namely the distinction between strong and weak representations. Consider the natural numbers: Peano's axioms enumerate properties that must be evidenced by any set if it is to qualify as a representation of the "natural numbers". Although various sets of λ-expressions can be concocted which evidence these properties, such λ-expressions also have other properties as well: For example, in the absence of free variables any λ-expression can be applied to any other λ-expression, so that the representation of 2 can be applied to the

representation of 3 even though this operation is not meaningful in arithmetic. Any representation that has properties beyond those required of it is called "weak", whereas a representation that has only the necessary and sufficient properties is called "strong". A problem inherent in definition by representation is avoiding the encroachment of unwanted additional properties.

Represented Memories: The preceeding discussion suggests that we could have defined memories by representation rather than by postulate. We now show two different ways to define them via representation by obs already in our universe of discourse. Our first approach is to regard a memory as a function, and to represent addresses by integers. To implement Extend we arrange that "cell 0" always contains the next free address. We have then

```
def Empty_memory = fn x. x eq 0 -> 0 | error

and Contents(M, a) = M a

and Extend(M, v) =
     let a = 1 + M 0  // next free cell                    (4.1-10)
     in
     (fn x. x eq 0 -> a | x eq a -> v | M x), a

and Update(M, a, v) =
     fn x. x eq a -> v | M x
```

This representation is weak for several reasons: Representation of addresses by integers gives addresses unwanted properties. For example, we can add two "addresses", which we could not do with postulated addresses. A second weakness is that postulated memories cannot be applied (i.e., they have empty domain), whereas these representations can be applied to some integers. On the other hand, we have the desired property of an arbitrarily large set of distinguishable addresses. Of course, there is the very practical objection to use of this representation in that it is disasterously inefficient if much updating is done.

An interesting aspect of the preceeding discussion is that memories, which provide the essence of what distinguishes L-PAL from R-PAL, can be represented by purely applicative functions. It turns out that we could instead represent memories by R-PAL structures. Consider the structure definitions

```
A memory has two components: a
     nextcell, which is an integer, and a
     mem.

A mem is either
     empty, or it is                                      (4.1-11)
     non-empty and has three parts: an
          address, and a
          value, and a
          mem.
```

Thus a mem is much like an environment in the R-machine. (See (3.5-8) on page 3.5-158.) We select the rather obvious choice of representing a memory by a 2-tuple and a non-empty mem by a 3-tuple, and we have

```
def Empty_memory =
     0, nil

and Contents(Memory, Address) =
     Look(Memory 2)
     where rec Look Mem =
          Address eq Mem 1 -> Mem 2 | Look(Mem 3)

and Update(Memory, Address, Value) =                    (4.1-12)
     Memory 1, (Address, Value, Memory 2)

and Extend(Memory, Value) =
     let NC = 1 + Memory 1  // next cell
     in
     let NextMemory = NC, (NC, Value, Memory 2)
     in
     NextMemory, NC
```

As in the functional representation, a memory that has been updated very often leads to serious inefficiencies in both time and space. This representation is of course also weak. It is the representation actually used in the L-PAL gedanken evaluator.

The inefficiency of each of these representations arises from the fact that a structure in R-PAL, once created, can never be changed. One would expect that an L-PAL representation of memories in which assignment statements could be used would be more efficient, so we show that this is indeed the case. (Of course, there is no value to this particular definition as part of a formal definition, since we do not permit ourselves to define L-PAL via an L-PAL program.) Since we are writing in L-PAL we can exploit the fact that functions can be executed for their effect only, and we refrain from supplying the memory as a parameter or returning a memory as part of the result. We then have

```
def
        M = nil  // the memory
        within
    Initialize_memory() =
        M := nil
and
    Contents A =
        M A
and                                                    (4.1-13)
    Extend V =
        M := M aug $V; Order M
and
    Update(A, V) =
        M A := V
```

Here there is only one memory rather than a set of them. Initialize_memory and Update are executed only for effect, Contents has a useful value and no effect, and Extend has both a useful value and an effect.

New Linguistic Constructs

Our purpose in developing abstract memories has been to provide a conceptual base in terms of which the ideas of sequence and assignment can be made precise. We now turn to this task, that is, to the formal definition of L-PAL. In defining the L-Machine, we continue (as with the R-Machine) to assume that the program to be evaluated has already been represented in the form of an abstract syntax tree. The new node-types incorporated into L-PAL are illustrated in Figure 4.1-3. The semantic intent of "aug" and of "$" has already been discussed, but we have still to discuss assignment and the variations on the conditional. First though we make an arbitrary decision converning the value of a sequence, say "E1; E2". Presumably, the expression E1 is to be executed for its effect only, so its value is of no interest. Accordingly, we take the value of any such sequence to be the value of E2.

Assignment: A second arbitrary decision concerns the value of an assignment command. Thus far we have established the side-effect of an assignment: to update the value associated with a variable or with a component of a tuple. But Figure 4.1-3 implies that an assignment command can occur anywhere that an expression can occur -- and expressions have values.

Two alternatives seem reasonable. The first is to introduce a new and distinct ob into our universe of discourse, say dummy, and adopt it as the value of a ":=" node; the second is to assume that the value of a ":=" node is the value of E2. Efficiency of computation slightly favors the latter alternative, since not infrequently a programmer may wish to re-use this value immediately. But the former alternative offers a correspondingly slight advantage in program explicitness, hence readability, and also eases the task of program debugging, since it makes it less convenient to write programs such as

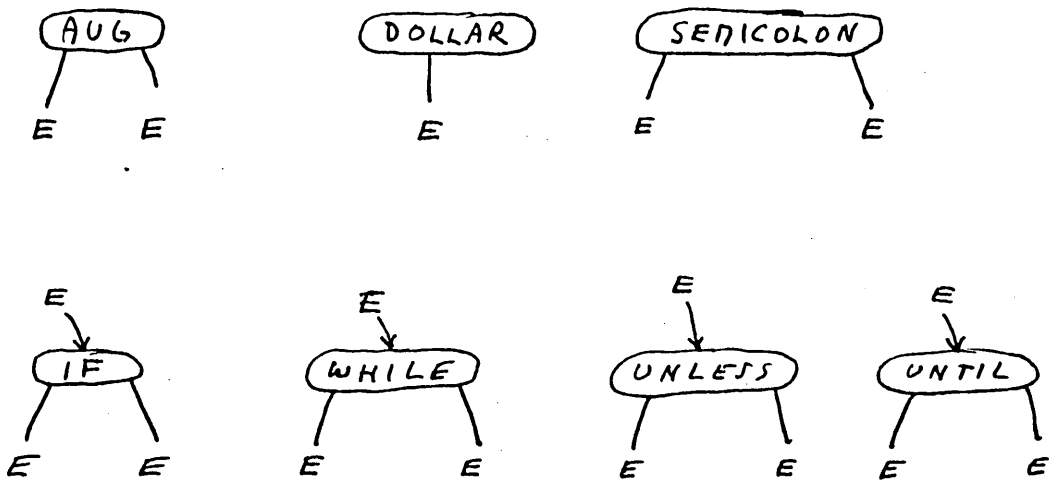

AUG
E E
DOLLAR
E
SEMICOLON
E E
E
IF
E E
E
WHILE
E E
E
UNLESS
E E
E
UNTIL
E E

Figure 4.1-3

```
x + (x := E2)                                    (4.1-14a)
```

whose value is so obviously dependent on the order of execution. PAL opts in favor of the concept of dummy, and incorporates the identifier "dummy" to denote this ob and the predicate "Isdummy" to test for it. In order to retain the value of E2 with this decision, the programmer need only write code such as

```
x + (x := E2; x)                                 (4.1-14b)
```

Of course, the semantics of this is also undefined because of its dependence on order of evaluation.

Figure 4.1-3 shows that the left son of a ":=" node may be any expression, but the examples used so far show only variables or components of a tuple used in that position. The usual semantic intent is that the left side be evaluated to yield the location into which to do the store. Thus if T denotes a tuple, obeying

```
T k := E                                         (4.1-15a)
```

updates the k-th component of T (providing that it exists). Similarly, obeying

```
(x > y -> x | y) := 0                            (4.1-15b)
```

sets to zero whichever of x or y was previously greater. An exception to this general rule occurs if the left son of the ":=" node is a COMMA node, in that then it is the components of the tuple that are updated. Thus obeying

```
x, y := x+1, y+1                                 (4.1-16a)
```

causes both x and y to be changed. The rule is that, if there are k sons of the COMMA node which is the left son of the ":=" node, then the expression on the right must denote a k-tuple. The semantics is that the k R-values on the right are all evaluated before any updating is done. Thus obeying

```
x, y := y, x                                     (4.1-16b)
```

causes the values of x and y to be interchanged. We later explain such simultaneous assignment in terms of simple assignment, regarding (4.1-16b) as sugaring for

```
Assign# (x, y) (y, x)                            (4.1-16c)
```

where Assign# is an identifier already in the environment which denotes a function that does simultaneous assignment by iteration of simple assignment. This function is given in Section 4.2 under the heading "Library Functions".

A second use of dummy in PAL involves the function Print. The intent of Print, of course, is to provide a side-effect on the world outside the computer itself, specifically to output (say, onto a piece of paper) a written representation of the argument to which Print is applied. As with assignments, the value of "Print E" is taken to be dummy for any expression E.

Variations on the Conditional: As do most other programming languages, PAL provides several syntactic variations on the conditional. Each of

```
test (B) ifso (E1) ifnot (E2)
test (B) ifnot (E2) ifso (E1)                    (4.1-17)
     (B) -> (E1) | (E2)
```

has identical semantics, although they have different parsing rules. (See the PAL Manual.)

Because of the L-PAL possibility of execution for effect, it is often useful to have a one-armed conditional. The intent of the construct

```
if E1 do E2                                      (4.1-18a)
```

is to execute E2 when and only when the value of E1 is true. We take (4.1-18a) as syntactic sugaring for

```
test E1 ifso E2 ifnot dummy                      (4.1-18b)
```

Similarly, the construct

```
unless E1 do E2                                  (4.1-18c)
```

is syntactic sugaring for

```
test E1 ifnot E2 ifso dummy                      (4.1-18d)
```

That is, (4.1-18c) is equivalent to

```
if not (E1) do E2                                (4.1-18e)
```

but it is more convenient to use where it is appropriate. Note that the value of an "if" or "unless" node is either that of E2 or dummy, depending on the value of E1.

PAL also provides a feature somewhat akin to the iteration statements of other languages. The intent of the construct

```
while E1 do E2                                   (4.1-19a)
```

is to execute E2 repeatedly, so long as the value of E1 remains true. We take the value of the overall construct to be dummy. Using labels (as in J-PAL) we can say that (4.1-19a) is sugaring for the program

```
L: if E1 do (E2; goto L)                         (4.1-19b)
```

where L is some identifier not otherwise used. However, this explanation, while perhaps helpful to the reader, cannot be used at this time as part of the formal definition of "while", since we must restrict our explanations to use only R-PAL. A conceptually simple (but somewhat inefficient) way to formalize the semantics of (4.1-19a) makes recourse to the auxiliary function Loop# defined by the PAL program

```
def rec Loop# x y =
    if x nil do (y nil; Loop# x y)               (4.1-20)
```

Assuming that the function Loop# is known, we encapsulate the intent of (4.1-19a) by considering it as sugaring for the combination

Loop# [λ(). E1] [λ(). E2]

in which the "λ(). ~~~" construct is used to defer evaluation of E1 and E2 until (each) application of Loop#. Similarly,

until E1 do E2 (4.1-21a)

is equivalent to the J-PAL program

L: unless E1 do (E2; goto L) (4.1-21b)

but is regarded for now as sugaring for

Loop# [λ(). not (E1)] [λ(). E2] (4.1-21c)

Note that for both "while" and "until" the boolean is evaluated before the first execution of the body, so that there is the possibility of executing the body zero times.

It should be clear that the "if", "unless", "while" and "until" constructs add no new power to the language. On the other hand, as syntactic devices they can contribute significantly to program readability. In particular, "while" and "until" correspond to a specialized form of recursion (called iteration). We see in conjunction with J-PAL that the special nature of such iterations makes it possible to accommodate them efficiently without recourse to recursion.

4.2 Mechanical Evaluation of L-PAL Programs

Just as the formal definition of R-PAL is provided by a gedanken evaluator whose algorithm is represented by a PAL program, so also the formal definition of L-PAL is provided by a gedanken evaluator whose algorithm is represented in R-PAL. And just as we gained familiarity with the R-PAL evaluator by simulating its operation with blackboard evaluator conventions, so also we develop an L-PAL blackboard evaluator. The strategy is a bit different for L-PAL, however, since certain aspects of the language (such as simultaneous assignment) are easier to explain in terms of the gedanken evaluator than to carry out in the blackboard evaluator. We adopt in each case the more expedient mechanism for explanation.

Blackboard Evaluation

The principle distinction between L-PAL and R-PAL involves the use of memories to specify sharing. In addition to the three components (Control, Stack and Environment) of the CSE machine, we now introduce a fourth component, M, for memory. Thus the gedanken evaluator for L-PAL is a CSEM machine.

With L-PAL as with R-PAL, blackboard simulation provides insight into the evaluation of programs. In this section we extend the informal blackboard bookkeeping rules of Chapter 3 to accommodate the new constructs (":=", ";", and "$") which occur in the control structure produced by L-PAL's "Translate" function, as well as show the control structure produced by the variations on the conditional. We must of course change the handling of "aug".

Memory Conventions: For blackboard purposes it is convenient to use a separate column to keep track of the state of the memory, like this:

Control	Stack	Environment	Memory
E0	E0	0: PE	0

Figure 4.2-1 Empty Blackboard Evaluator

The memory we are using is similar to that of (4.1-12), page 4.1-211, in which PAL structures are used to represent memories. For use by Extend, the left column of the memory contains the index of the last cell used. Addresses as represented by σ_1, σ_2, ..., when they appear in the stack or elsewhere. (σ may be thought of as mnemonic for storage.)

Any representation of memories must permit us to realize the three functions we need. Figure 4.2-2 shows a memory that has been extended and updated several times. On the first extension 12 was put into cell σ_1. Then cell σ_2 was created holding 14, and then σ_1 was updated to hold 10. The rules are as follows:

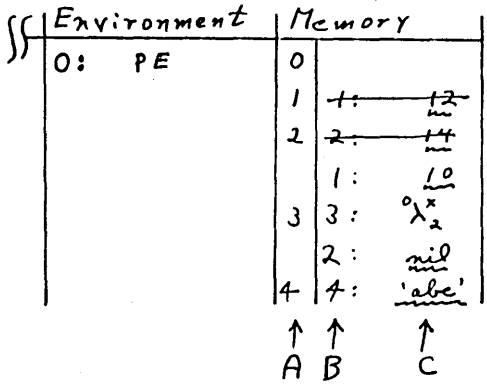


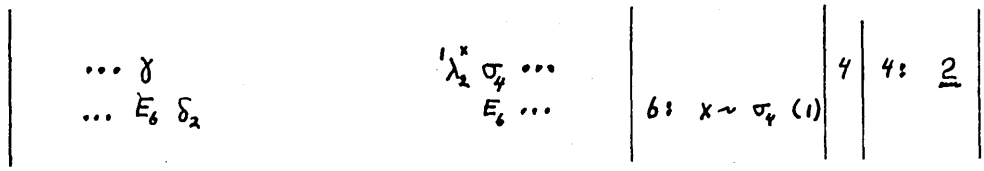
Figure 4.2-2:

Memory for the L-PAL Blackboard Machine

- To extend the memory, note the last integer used in column A. Write the next integer on the current line in column A, and make an entry for that cell in columns B and C.
- To update a cell, make a new entry for it in columns B and C. Find the last entry for that cell in column B, and cross out columns B and C on that line.
- To find the contents of a cell, look for an uncrossed-out instance of its address in column B. The corresponding entry found in column C is the value.

If the rules are followed, column B can never contain more than one uncrossed-out instance of any given address. The purpose of the crossing out is to decrease the possibility of human error in the lookup operation.

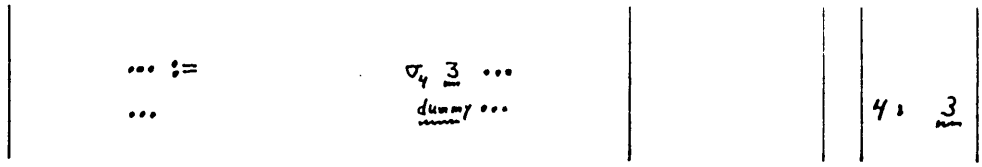
We have already noted the PAL design decision that all programmer-defined names should be variables, by which we mean that they should be updatable. The implication of this is that the environment should couple a name with an address rather than with a value. Thus we have the transition



Here cell 4 contains the value of x, shown as 2. Subsequent execution of the control corresponding to the PAL statement

x := 3

would lead to the transition



At the same time this last line is obeyed, we would also cross out the earlier memory pairing of address 4 with 2.

As an example, consider the L-PAL program

```
let x = 2
in                                    (4.2-1a)
x := x + 3;
x
```

which was also shown in (4.1-2). This desugars to the AE

$$(\lambda_1 x.\ x := x+3;_2\ x)\ 2 \qquad (4.2\text{-}1b)$$

and to the control structure

$$\begin{aligned} &\gamma\ \lambda_1^x\ 2 \\ &\delta_1 \ =\ \delta_2\ ;\ :=\ x\ +\ x\ 3 \\ &\delta_2 \ =\ x \end{aligned} \qquad (4.2\text{-}1c)$$

Note that the PAL sequence "E1; E2" leads to a control sequence consisting of the control for E2 followed by a semicolon followed by the control for E1. We consistently abbreviate with a δ that piece of control following a semicolon. Execution in an L-PAL blackboard evaluator of this control is shown in Figure 4.2-3. There are many new ideas in this evaluation, which we mention briefly here and explain in more detail in the next subsection. On line 2 we must apply a closure. Since we have agreed that environments associate names with addresses, we must extend the memory (as in line 3) and replace the 2 in the stack by σ_1, the address of a cell that holds 2. Then in line 4 we make a new environment in which x is associated with σ_1. In line 5, + is to be applied to σ_1 and 3. Since + needs R-values, we replace σ_1 by the contents of that cell, so that + can be applied to 2 and 3 on line 6. On line 7 we update cell 1 to hold 5. The memory entry on line 3 is crossed out at this time. The answer shown is σ_1 as the stack item left on completion of the evaluation. Of course the value we are interested in is 5, the contents of that cell.

Control Items: The control items new to the L-PAL evaluator are the following four items:

;	statement separator
:=	assignment functor
$	unshare functor
aug	tuple extender

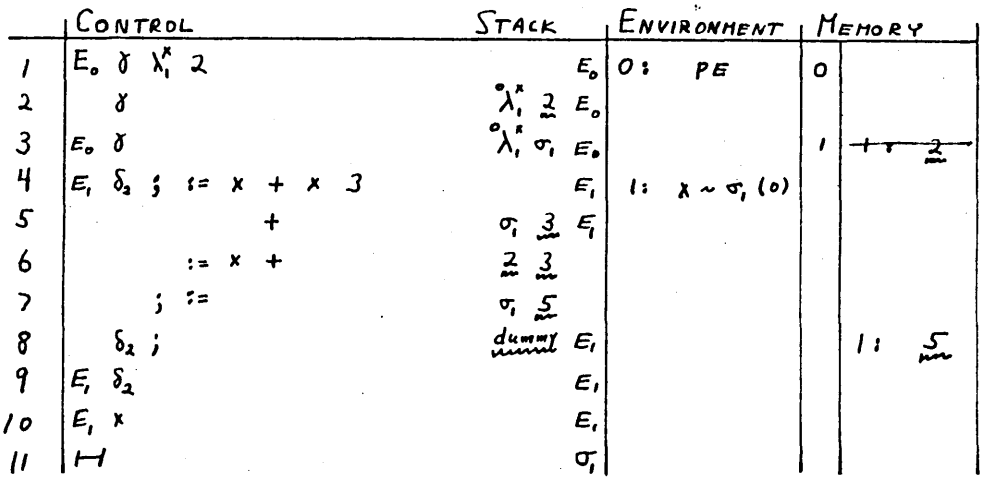


Figure 4.2-3: L-PAL Evaluation of (4.2-1)

Much of the processing required for these is implied by the discussion of Section 4.1 and the example just given. For example, we have agreed that the value of the sequence

E1; E2

is to be that of E2 and that the value of E1 is to be discarded. Thus the rule for a semicolon in the control is simple: Discard the top stack item. This rule was used on line 8 of Figure 4.2-3.

If := is the top control item, the top of the stack should be an address and the second stack item should be an ob. The effect of obeying the := is that the cell whose address is on top of the stack is to be updated so that its contents is the ob which is the second stack item. In going from line 7 to line 8 of Figure 4.2-3 cell 1 is updated to hold 5. Our blackboard convention is to write a new "memory layer" showing the new address-contents pairing for that cell, and to line out the last such pairing so as to reduce the possibility of human error.

The unsharing functor "$" is of use when an address is the top stack item. Its effect is merely to replace the address on the stack by its contents.

When aug is executed the top stack item is to be a vector of length n (where n is perhaps zero) of addresses, and the second stack item is to be an address. The effect is to remove these items and to leave in the stack a vector of length n+1 of addresses, the first n of those being the addresses in the previous top stack item and the n+1-st being the old second item.

Context Rules: Let the term mode, when used to refer to a stack item, refer to whether the item is an R-value or an L-value. The discussion of the previous subsection is predicated on the assumption that the top one or two stack items already have the correct mode before the control item under discussion is encountered. That this fortuitous situation does not always come about in practice is shown in several places in Figure 4.2-3 -- on lines 2 and 5 for example. The evaluation rule is simple: When the mode of a stack item is not as needed, fix it, using either Contents or Extend as appropriate. (Contents was used on line 5 and Extend on line 2 of Figure 4.2-3.) Thus for example we require that, if the top of the control is γ and the top stack item is a λ-closure, the second stack item be an L-value. This is one instance of what we call a context rule, one which can be stated in English by saying that closures are always applied to L-values. All of L-PAL's context rules are summarized in the following table:

Top of Control	Top Stack Item	2nd Stack Item
γ	R	--
γ	λ-closure	L
γ	tuple	R
γ	basic function	R
aug	R	L
β	R	--
:=	L	R
$	R	--
;	--	
τ_k	L	L...L

Table 4.2-1 Context Rules

Here R and L indicate respectively need for an R-value or an L-value, and "--" indicates "don't care". Whenever a conflict with a context rule is detected in the course of a blackboard evaluation, the necessary "transfer function" is to be invoked. Some additional comments about this table seem noteworthy:

- It must be remembered that although an address is an L-value, a vector of addresses is an R-value.
- The context rules make nugatory all statements such as

  5 := 3

  The effect of obeying such a statement is to get a new cell containing 5, to update the contents of that new cell to 3, and then to return the value dummy. Since no reference to the address endures, the computational effect of the assignment is equivalent to that of

dummy

In the L-PAL gedanken evaluator, this situation is detected (as it presumably would be by any alert blackboard evaluator) and the useless memory extension is not done. Of course the evaluation of the right side may have side effects.

- The design decision that requires an L-value as rand in an application whose rator is a λ-closure implies that x and y share during evaluation of the body (E) in program segments such as

  ... (let y = x in E) ...

  Thus an assignment to either of x or y in E also updates the other. If such sharing is not desired, the programmer should write

  ... (let y = $ x in E) ...

  Note also that x and y share in

  let F x = P in ... F y ...

Neglect of these sharing effects can lead to programming errors that are quite hard to diagnose, so the programmer is advised to note carefully how he binds any variable which he plans to update.

This use of PAL's context rules is a process which, in the current literature, is often referred to as coercion. The idea is that the programming system, noting that the program has produced one sort of value in a context in which another sort is required, automatically invokes a suitable transfer function. PAL's only coercion is between R-values and L-values. A coercion common to many programming languages is to coerce the left operand of "+" in expressions such as

3 + 7.2

to be of type rational rather than integer. That is, a transfer function such as PAL's ItoR is invoked, so that the above is treated as sugaring for

(ItoR 3) + 7.2

An elaborate coercion scheme can be very advantageous to the programmer whose desires match closely those of the designers of the system which he is using. To the extent that this match fails in a system in which the design decisions cannot be overridden, the user suffers. PAL's designers have opted for minimal coercion, a decision that seems to be consistent with our pedagogical objectives.

Other Matters: Recursion in the L-PAL blackboard machine is to be handled just as in the R-PAL machine, using the Y-η method. We permit τ_K as a blackboard control item, although for the gedanken evaluator we desugar tuples to usages of aug, as in R-PAL. Simultaneous assignment is not handled in the blackboard evaluator, although the technique presented later for the L-PAL gedanken

evaluator would clearly apply.

The conditional sugarings if and unless are not of concern in the blackboard evaluator, as each of them leads to a suitable β node in the control by the correspondences of (4.1-18). Although we could process while (and until) by regarding

```
while B do E                                    (4.2-2a)
```

as syntactic sugaring for

```
Loop# (λ(). B) (λ(). E)                         (4.2-2b)
```

where Loop# is the function shown in (4.1-20), and in fact do that in the gedanken evaluator, there is a more efficient method for blackboard evaluation. It hinges on the observation already made that (4.2-2a) can be regarded as sugaring for

```
L: if B do (E; goto L)                          (4.2-2c)
```

where L is some identifier not appearing elsewhere. Of course this is a J-PAL program, but it turns out we can get its effect by suitable desugaring. We standardize the tree shown in (a) of Figure 4.2-4 into that shown in (b). As in

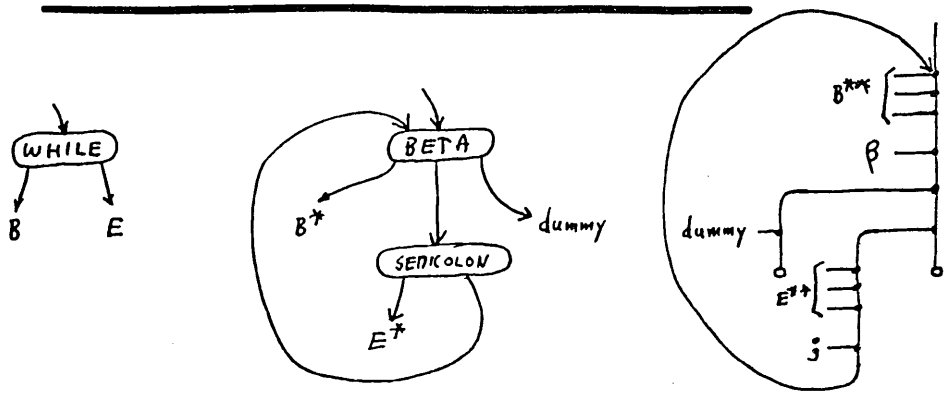


Figure 4.2-4: Standardization for while

Figure 3.5-7, B* stands for the standardization of B and E* for that of E. The control structure is shown in (c), in which B** and E** are the flattened B* and E*. For blackboard purposes, assume that the control of (c) is abbreviated by δ_k. Then the control δ_k is

$$\begin{aligned} \delta_k &= \delta_{k+1}\ \delta_{k+2}\ \beta\ B^{**} \\ \delta_{k+1} &= \delta_k\ ;\ E^{**} \\ \delta_{k+2} &= \text{dummy} \end{aligned} \qquad (4.2\text{-}3)$$

We are unable to build a control like this in the L-PAL gedanken evaluator, since it involves a structure with a loop and it is not possible to write in R-PAL a program which produces such a structure. We can (and do) use this technique in the J-PAL gedanken evaluator, since it is written in L-PAL. Its efficiency for blackboard evaluation is suggested by the example shown below. To see the gain, the reader should try desugaring the while as shown in (4.2-2b) and starting the blackboard evaluation. (The definition of (4.1-2a) is needed

as part of the program to be evaluated.) It takes many more steps.

It seems worthwhile to comment on introducing while (and until) into L-PAL, since they seem to be J-PAL constructs. Doing so gives us the advantage of being able to use while and until in the code for the J-PAL gedanken evaluator, with two gains:

- The programs are more perspicuous, since we may use a simple while construct instead of a recursive function to accomplish iteration.
- There is the pedagogic advantage of exhibiting contrasting techniques to accomplish the same task.

This latter point manifests itself in two ways: Firstly, we see that while can be regarded either recursively (with Loop#) or iteratively (via a loop in the control structure). Secondly, some tasks which are programmed recursively in the R-PAL and L-PAL gedanken evaluator are programmed iteratively in the J-PAL evaluater.

An Extended Example: We give now a fairly long example of an L-PAL blackboard evaluation, showing most of the techniques introduced in this section. Consider the program

```
let k, T = 1, nil
in
while k le 2 do (T := T aug k; k := k+1);                    (4.2-4)
T
```

This program clearly builds a 2-tuple and returns it as its value. Before examining the evaluation, the reader is advised to decide in his own mind the value of this program. It is not (1,2), nor is it (2,2).

The program (4.2-4) desugars as

```
[λ (k, t). ↓ ] (1, nil)
while k le 2 do (T := T aug k; k := k + 1; T                 (4.2-5)
```

in which the down arrow indicates that the AE on the next line is to stand at that point. We then get the control structure

$$
\begin{aligned}
&\delta\ \lambda_1^{k,T}\ \gamma_2\ 1\ \mathrm{nil}\\
&\delta_1 = \delta_6\ ;\ \delta_2\\
&\delta_2 = \delta_3\ \delta_5\ \beta\ \mathrm{le\ k\ 2}\\
&\delta_3 = \delta_4\ ;\ :=\ \mathrm{T\ aug\ T\ k}\\
&\delta_4 = \delta_2\ ;\ :=\ \mathrm{k\ +\ k\ 1}\\
&\delta_5 = \mathrm{dummy}\\
&\delta_6 = \mathrm{T}
\end{aligned}
\qquad (4.2\text{-}6)
$$

Here we have used consistently the convention that the right son of a semicolon node is to be abbreviated. Note the loop: δ_2 is used in δ_4.

Evaluation in an L-PAL blackboard evaluator is shown in Figure 4.2-5. We need a way to represent tuples. Since a tuple is an R-value, we use a vector of address with a wiggly underline. Thus the 1-tuple whose component is σ_1 is written $\underset{\sim}{\sigma_1}$, as for example in the memory on line 14. Note the use of the context rules, on lines 2, 7, 12 and elsewhere. As is our usual practice, we have elided many uninteresting lines.

Note that the answer is that 2-tuple whose components are each σ_1. Thus the value of (4.2-4) is (3,3), since cell 1 holds $\underset{\sim}{3}$ at the end. Because of PAL's context rules -- in particular the one that requires an L-value as the second stack item under aug -- all of the components of the tuple share with $\underline{k}$. (Changing the "2" in (4.2-4) to a "5" would produce the 5-tuple (6,6,6,6,6) as value.) The easiest way to change the original program to cause it to produce (1,2) is to change the assignment to $\underline{T}$ to read

```
T := T aug ($ k)
```

(The parentheses are not needed.) The "$" forces the R-value of $\underline{k}$ to be taken, and then a new cell holding that R-value will be "aug"ed onto the tuple.

The L-PAL Gedanken Evaluator

Having gained intuitive understanding of L-PAL through the blackboard evaluator, we proceed now with the formal definition of the language, using the L-PAL gedanken evaluator. The mechanism is very similar to that of R-PAL, the addition of a memory being the biggest change. The main program is

```
def Gedanken_evaluator Program =
    let Control_structure = Translate Program
    and M0 = Initial_memory nil                          (4.2-7)
    in
    Evaluate (Control_structure, Empty-stack, PE, M0)
```

As before, Evaluate is quite simple:

```
def rec Evaluate (C, S, E, M) =
    Null C -> (Rval(M, t S), M) |                        (4.2-8)
    Evaluate (Transform(C, S, E, M))
```

Our task now is to explain Translate and Transform, as well as the details of the final answer returned.

Translate: As in the R-machine, Translate involves the composition of standardization and flattening, so that we have

```
def Translate Program = FF (ST Program, nil)             (4.2-9)
```

Here Program is an abstract syntax tree representation of a PAL program, and ST and FF, which differ only slightly from their R-PAL counterparts, have yet to be discussed. All the cases for ST for R-PAL as shown on page 3.5-193 remain unchanged, but we replace the last line (the "Error" line) by the following code:

	Control	Stack	Environment	Memory
1	$E_0\ \gamma\ \lambda_1^{k,T}\ \tau_2\ 1\ nil$	E_0	0: PE	0
2	τ_2	$1\ nil\ E_0$		
3	$\gamma\ \lambda_1^{k,T}\ \tau_2$	$\sigma_1\ \sigma_2$		1
4	$E_0\ \gamma$	$^{\circ}\lambda_1^{k,T}\ \sigma_1\ \sigma_2\ E_0$		2
5	$E_1\ \delta_6\ ;\ \delta_2$	E_1	$1:\{k \sim \sigma_1, T \sim \sigma_2\}\ (0)$	
6	$;\ \delta_3\ \delta_5\ \beta\ le\ k\ 2$	E_1		
7	le	$\sigma_1\ 2\ E_1$		
8	$\beta\ le$	$1\ 2$		
9	$;\ \delta_3\ \delta_5\ \beta$	$true\ E_1$		
10	$;\ \delta_4\ ;\ :=\ T\ aug\ T\ k$	E_1		
11	aug	$\sigma_2\ \sigma_1\ E_1$		
12	$:=\ T\ aug$	$nil\ \sigma_1$		
13	$;\ :=$	$\sigma_2\ \sigma_1$		
14	$\delta_4\ ;$	$dummy\ E_1$		2 σ_1
15	$\delta_2\ ;\ :=\ k\ +\ k\ 1$	E_1		
16	$:=\ k\ +$	$\sigma_1\ 1\ E_1$		
17	$;\ :=$	$\sigma_1\ 2$		
18	$;\ \delta_2\ ;$	$dummy\ E_1$		1 2
19	$;\ \delta_3\ \delta_5\ \beta\ le\ k\ 2$	E_1		
20	le	$\sigma_1\ 2\ E_1$		
21	$;\ \delta_3\ \delta_5\ \beta\ le$	$2\ 2\ E_1$		
22	$;\ \delta_4\ ;\ :=\ T\ aug\ T\ k$	E_1		
23	aug	$\sigma_2\ \sigma_1\ E_1$		
24	$:=\ T\ aug$	$\sigma_1\ \sigma_1$		
25	$;\ \delta_4\ ;\ :=$	$\sigma_2\ \sigma_1\ \sigma_1\ E_1$		2 $\sigma_1\ \sigma_1$
26	$;\ \delta_2\ ;\ :=\ k\ +\ k\ 1$	E_1		
27	$+$	$\sigma_1\ 1\ E_1$		
28	$:=\ k\ +$	$2\ 1$		
29	$;\ \delta_2\ ;\ :=$	$\sigma_1\ 3\ E_1$		
30	$;\ \delta_3\ \delta_5\ \beta\ le\ k\ 2$	E_1		1 3
31	$\beta\ le$	$\sigma_1\ 2\ E_1$		
32	$;\ \delta_3\ \delta_5\ \beta\ le$	$3\ 2\ E_1$		
33	$E_1\ \delta_6\ ;\ dummy$	E_1		
34	$E_1\ T$	E_1		
35	$\vdash$	σ_2		
36	$\vdash$	$\sigma_1\ \sigma_1$		

Figure 4.2-5 : Blackboard Evaluation of (4.2-4)

```
| Type IF
    ->   BETA_ (ST (x 1)) (ST (x 2)) DUMMY
| Type WHILE
    -> ( let u = LAMBDA nil (ST (x 1))
         and v = LAMBDA_ nil (ST (x 2))
         in
         GAMMA_ (GAMMA_ Loop_VAR u) v
       )
| Type ASSIGN
    -> ( let u = ST (x 1)                                    (4.2-10)
         and v = ST (x 2)
         in
         Is_tag (x 1) COMMA
           -> GAMMA_ (GAMMA_ Assign_VAR u) v
           |  ASSIGN_ u v
       )
| Type DOLLAR -> DOLLAR_ (ST (x 1))
| Type ALPHA  -> ALPHA_ (ST (x 1)) (ST (x 2))
| error
```

Here Loop_VAR and Assign_VAR denote representations of the variables we have been calling Loop# and Assign#, respectively; ALPHA is the tag used for semicolon nodes; IF, WHILE, ASSIGN and DOLLAR are tags; and DOLLAR_, ALPHA_ and ASSIGN_ are the obvious tagging functions. Code for unless and until is not shown here; it is of course similar to that for if and while. As in Chapter 3, we defer till later any discussion of these representation issues. Complete programs are given at the end of the chapter. A graphical representation of the transformation of ST is shown in Figure 4.2-6.

The code for FF differs from that shown for R-PAL on page 3.5-194 in that the last line there (the "Error" line) is to be replaced by the code

```
| Type ALPHA  -> FF ( x 1, FF (x 2, c) )                   (4.2-11)
| Sons x eq 2 -> FF (x 2, FF (x 1, (Get-tag x, c) ) )
| Sons x eq 1 -> FF (x 1, (Get_tag x, c) )
| error
```

Note that the code for ALPHA insures that the control structure for the left son is evaluated before that for the right son, corresponding to the fact that we execute first that which appears before a semicolon and then that which appears after it. The transformation of FF is shown graphically in Figure 4.2-7.

Transform: The function Transform, which transforms one (C, S, E, M) state into another, is all that is needed to complete the specification of the L-PAL gedanken evaluator. It is similar to R-PAL's Transform:

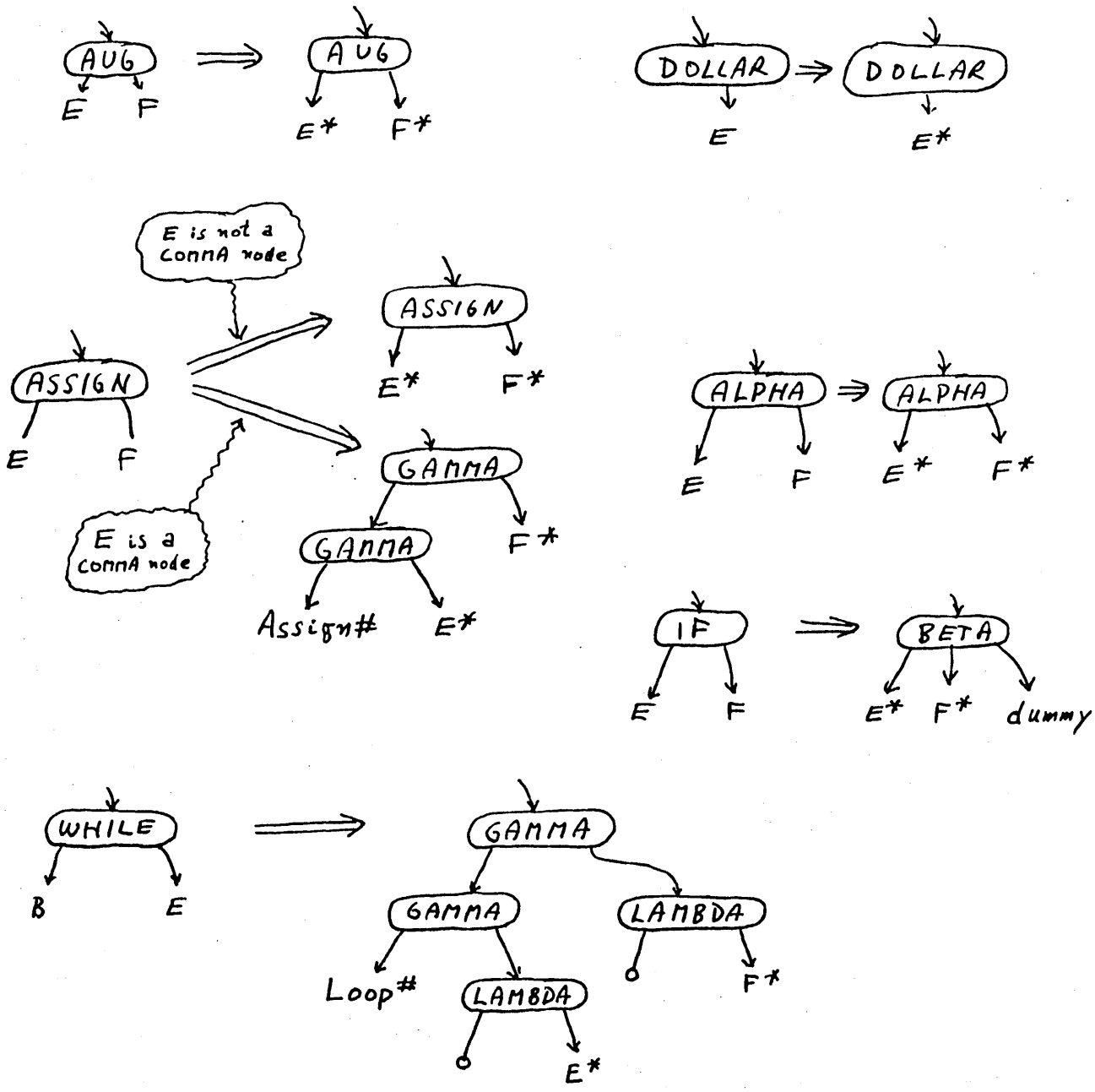


Figure 4.2-6: Effect of ST in L-PAL

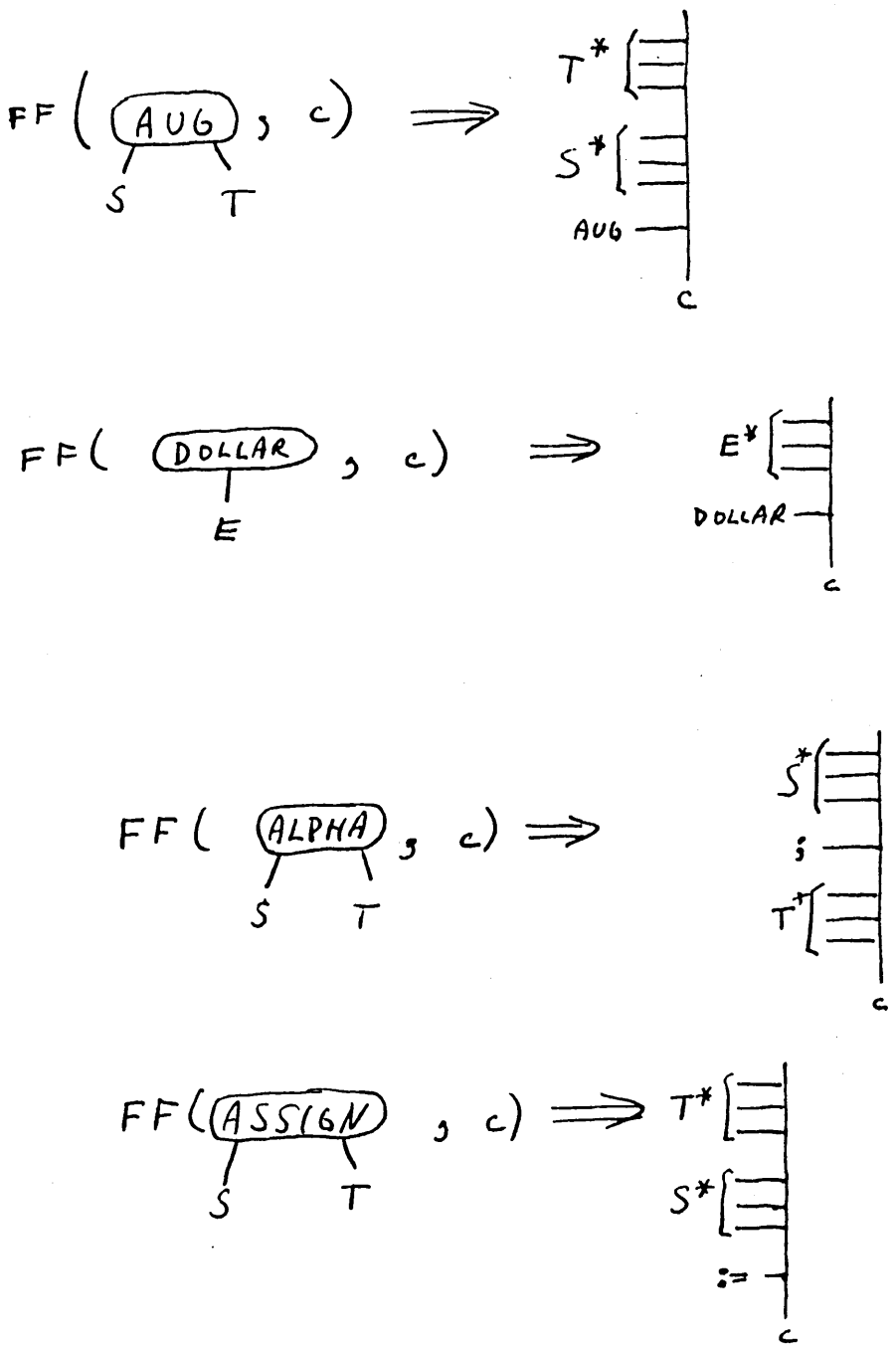


Figure 4.2-7: Effect of FF in L-PAL

```
def Transform (C, S, E, M) =
     let A = C, S, E, M
     and x = t C // Top of control.
     in
       Is_constant x     -> Eval_constant A
     | Is_variable x     -> Eval_variable A
     | Is_lambda_exp x   -> Eval_lambda_exp A
     | x eq ALPHA        -> Do_alpha A // semicolon
     | x eq ASSIGN       -> Do_assign A
     | x eq DOLLAR       -> Do_dollar A
     | Is_address(t S)   -> LtoR A
     | x eq BETA         -> Do_conditional A
     | x eq AUG          -> Do_aug A                          (4.2-12)
     | x eq RETURN       -> Do_return A
     | x eq GAMMA        ->
         ( let r = t S
           in
             Is_closure r    -> Apply_closure A
           | Is_constant r   -> Apply_constant A
           | Is_tuple r      -> Apply_tuple A
           | Is_Y r          -> Apply_Y A
           | Is_eta r        -> Apply_eta A
           | error
         )
     | error
```

For the most part, the routines called to do the work are similar to those with the same name in the R-PAL evaluator. Thus evaluation of constants, variables and λ-expressions, as well as handling of RETURN and conditionals, is the same in L-PAL. Of course we must add the fourth component, M, to the machine state, but in each case M is unaffected by the transformation. The relevant code is shown at the end of the chapter.

When the top of the control is ALPHA, corresponding to a semicolon in the original program, we wish merely to discard the result stored in the top of the stack. This is accomplished as shown:

```
def Do_alpha (C, S, E, M) =
     r C, r S, E, M                                           (4.2-13)
```

When ASSIGN is the top control item, we wish to update the memory. We have

```
def Do_assign (C, S, E, M) =
     let Val =  // Right side of :=
          Is_address (2d S)
          -> Contents (M, 2d S)
          |  2d S
     in                                                       (4.2-14)
     let New_M =  //New memory
```

```
            Is_address (t S)
            -> Update (M, t S, Val)
            |  M
        in
        r C, Push (DUMMY, r2 S), E, New_M
```

Here Val is the R-value to the right of the ":=". If the top stack item is not an address, the assignment is nugatory. As we decreed earlier, we take the value of the assignment to be dummy. DUMMY has a suitable value.

The control item DOLLAR signifies the programmer's intent that an R-value be the top stack item. The code

```
def Do_dollar (C, S, E, M) =
    r C, Push ( Rval(M, t S), r S) ), E, M
```
(4.2-15)

has the desired effect.

Each of the remaining cases requires that the top of the stack be an R-value. In order to guarantee this, we want the function LtoR defined by

```
def LtoR (C, S, E, M) =
    let New_S = Contents (M, t S)
    in
    C, Push (New_S, r S), E, M
```
(4.2-16)

to be called when needed. It follows from the placement of the test

```
Isaddress (t S) -> LtoR A
```

in (4.2-12) that this transformation is invoked whenever both of the following hold:

(a) The top control item is "β", "aug" or "γ"; and
(b) the top stack item is an L-value.

On the next iteration of Transform, the control item of (a) governs and evaluation proceeds as usual.

The control item "aug" signifies that a vector of addresses (at the top of the stack) is to be extended by the addition of another component:

```
def Do_aug (C, S, E, M) =
    let New_M, x =
        Is_address (2d S)
        -> (M, 2d S)
        |  Extend (M, 2d S)
    in
    let V = Augment_tuple (t S) x
    in
    r C, Push (V, r2 S), E, New_M
```
(4.2-17)

Note the definition of "New_M" and of x. PAL requires that the right side of this simultaneous definition evaluate to be a 2-tuple, and it should be clear that this will be the case whether or not the second stack item is an address. (The function Extend returns a 2-tuple.) The function Augment_tuple used knows how items in the stack are represented.

Application of a closure is as follows:

```
def Apply_closure (C, S, E, M) =
    let New_M, Rand =
        Is_address (2d S)
        -> (M, 2d S)
        | Extend (M, 2d S)
    and R = t S // The rator
    in                                                        (4.2-18)
    let New_C = Prefix (Body R, Push (RETURN, r C))
    and New_S = Push (E, r2 S)
    and New_E = Decompose (bV R, Rand, Env R, New_M)
    in
    New_C, New_S, New_E, New_M
```

The differences from R-PAL all arise from presence of the memory. Note particularly that Decompose has a fourth parameter: the memory in effect when it in called. It needs this to be able to access tuple components in the case when the λ-closure being applied has a structured bV-part.

All other cases are enough similar to R-PAL that no further discussion is needed.

Order of Evaluation: In the last section of Chapter 3, starting on page 3.5-180, we observed that the gedanken evaluator for R-PAL is over-specified in that a right-to-left order of evaluation of operands is forced, even though doing so is irrevelant to the semantic intent. A method of remedying this defect was proposed, having to do with a random selection in Translate between GAMMA and AMMAG for the control structure, the former leading to right to left evaluation and the latter to left to right. At the time, we mentioned that the whole discussion was rather irrevelant to R-PAL, since it is not possible to write in R-PAL a program whose value depends on order of execution. Now that we have L-PAL, it is easy to write such programs. A simple one, similar in spirit to (3.5-28), is

```
let x = 1
in
2 * (x := x+1; x)  +  3 * (x := x+1; x)
```

whose value is either 12 or 13. Another example

```
let x = 1
in
[x > 0 -> (λt.t) | (λt.-t)] [x := -x; x]
```

clearly depends on whether the rator or the rand is evaluated first. The design decision taken in PAL is predicated on the belief that a program is unnecessarily obscure if its successful execution depends critically on such implicit implementational details as order of evaluation. The decision is that such programs should be undefined.

As suggested in section 3.5, we could modify the gedanken evaluator by calling the random function Choice at suitable places to achieve the required undefinedness. We do not do so in our definition of the L-machine (or the J-machine) simply to avoid obfuscating other important concepts with unnecessary detail. In a formal definition of PAL, however, Choice would be used as suggested above. Our present objectives are pedagogic rather than formal.

Library Functions

There is some unfinished business still before us: We have yet to specify the function Assign# used in desugaring simultaneous assignments, and we must explain PAL's def construct which we have been using in our coding. The topics are related, since we can made Assign# available to the user by assuming a suitable def for it.

The Function Assign#: Recall that an ASSIGN node whose left son is a COMMA node is desugared as follows:

Figure 4.2-8: Standardization of Simultaneous Assignment

Thus for example, the statement

x, y := E (4.2-19)

is desugared as if the programmer had written

Assign# (x, y) E (4.2-20)

The assumption is that the identifier Assign# is in the environment when the program runs. We see later how it gets there, but are concerned now with defining the function which it denotes.

The function is a curried function which takes two arguments, the first of which must always be a tuple. (Calls for Assign# can only be produced by ST,

and then only when the left side of the assignment denotes a tuple.) The function requires that its second argument be a tuple of equal order. Since we require that all R-vaues on the right be determined before any assignment is done, the function makes a one-level copy (with "$") of each component of its second argument, and then the assignments are done. We have then the following code:

```
def Assign# x y =
     let n = Order x
     and w, k = nil, 1
     in
     unless n eq Order y do error;
     while k le n do                                    (4.2-21)
        ( w := w aug $ (y k);
          k := k + 1
        );
     k := 1;
     while k le n do (x k := w k; k := k+1)
```

Since this program is to be interpreted by the L-PAL gedanken evaluator, it may use all of the power of L-PAL. (But not simultaneous assignment. Why not?) Note how natural is the use of the one-armed conditional, and how convenient is the while construct for iterating through first the components of y and then those of x. The unsharing functor "$" in the assignment to w is critical, since without it the components of w would share with those of y, thus nullifying the effect of creating w.

This definition has an unfortunate property: It over-specifies PAL in terms of order of assignment to the components in a simultaneous assignment. Better in some sense would be a random choice of which way to go. The objective is that statements such as

```
x, x := 1, 2
```

be undefined.

The "def" Construct: The assumption underlying our exposition of simultaneous assignment is that the identifier Assign# is somehow in the environment of the user's program before it starts to run. Of course this is not a new concept, since all of the postulated functions such as Stern, Null, Isstring as well as those denoted by such functors as "+" and "not" are in the primitive environment PE. What is different here is that Assign# is written in PAL, and what is needed is an explanation of PAL's "def" construct.

Recall from the discussion of def at the beginning of section 3.5 (on page 3.5-152) that a program of the form

```
def ⟨definition⟩
```

has no defined semantics when taken in isolation but does when it appears in a suitable context. The PAL syntax, as given in Appendix 2.1 of the PAL Manual,

says

$$\langle \text{program} \rangle \ ::= \ \{ \text{ def } \langle \text{definition} \rangle \ \}_1^{\infty} \mid \langle \text{expression} \rangle \qquad (4.2\text{-}22)$$

This syntax defines that which may be submitted as input to a PAL compiler. It suggests the term "def-program" for a <program> consisting of one or more def's. What we are concerned with now is how one can combine several def-programs with an expression.

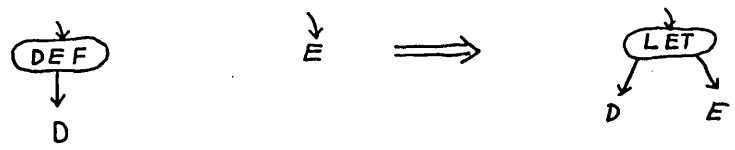
Let us think of the problem like this: Suppose that a programmer has a collection of one or more def-programs (such as the definition of Assign# in (4.2-21)) as well as an expression. The def-programs are to be taken in some order and followed by the expression, and our task is to ascribe semantics to the result. We do so now in two different ways, and select one of them for inclusion in the formal definition.

The first explanation is in terms of textual modification. Each def-program is to be altered as follows:

- The first "def" in it is replaced by "let".
- All other instances of "def" which occur in it are replaced by "in let".
- The word "in" is appended to the program.

We then form one long program by concatenating together all of the modified def-programs, in order, followed by the expression. The result will be a syntactically correct PAL program (if all the original programs were correct) and its semantics are deducible from the rules already given. Note that the order of appearance of the def-programs is important, since each may refer to identifiers defined in previous ones.

An alternate way to explain a collection of def-programs and an expression is in terms of input to the function Translate which is part of the gedanken evaluator. Assume that all of the programs exist as syntax trees suitable as input to Translate. For that purpose, we assume a new node type, def, whose single son is a definition. Now consider the <u>last</u> def node and the expression, and perform the transformation



which replaces them by a single let node. Since this latter is an expression, we can repeat the process on all of the def nodes in succession, until all that remains is a single let node which can be input to Translate. Figure 4.2-9 shows successive steps of this transformation on two def-programs and an

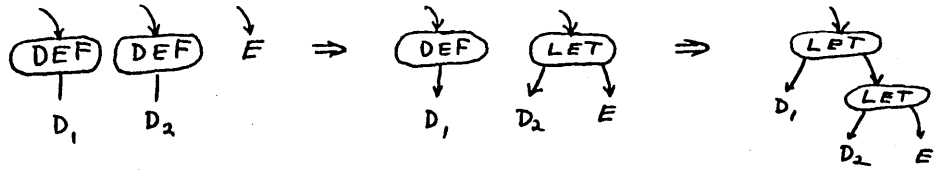


Figure 4.2-9: Processing "def" programs

expression.

A curried function Do_def to do this task may be defined by the PAL program

```
def rec Do_def x  =
     Is_tag x DEF
     -> [ fn y. LET_ (x 1) (Do_def y) ]
     |  x
```
(4.2-23)

Note that this function is to be applied to the def's in the order in which they are to be processed, and "eats up" arguments until it encounters one which is not a def.

All of the preceeding discussion applies equally to R-PAL and to L-PAL, and we see in the next chapter that it applies also to J-PAL.

The L-PAL Library: We have seen that standardization of L-PAL programs may produce calls for Assign# or Loop#. Thus it is assumed that definitions of these variables must precede the user's program. Since the code for Assign# contains while, which desugars into a call for Loop#, it is necessary that Loop# appear before Assign#.

We can fix the formalization easily by modifying the definition of the function Gedanken_evaluator as given in (4.2-7) to read

```
def Gedanken_evaluator Program =
     let Tree = Do_def Loop_T Assign_T Program
     in
     let Control_structure = Translate Tree
     and M0 = Initial_memory nil
     in
     Evaluate (Control_structure, nil, PE, M0)
```
(4.2-24)

Here Do_def is the function defined in (4.2-23), and Loop_T and Assign_T are syntax trees which are def-nodes corresponding to (4.1-20) and (4.2-21), respectively. The variable Tree is a syntax tree for the two library functions and the user's program, and the rest of the evaluation proceeds as before.

An alternate approach is to leave the function Gedanken_evaluator unaltered, and to make it the user's responsibility to include either or both

of Loop_T or Assign_T as needed. Suppose that User_T denotes the tree to be evaluated. Then instead of applying the function of (4.2-24) to User_T, one might instead apply the function of (4.2-7) to the tree denoted by

Do_def Loop_T Assign_T User_T

The result is clearly the same. The advantage to the latter approach has to do with efficiency: The user need not include Loop_T or Assign_T unless they are needed. Although the issue of efficiency is not really relevant in connection with a formal definition, it nonetheless seems worth mentioning in these notes. This alternate approach is assumed in the formalization at the end of the chapter.

4.3 Listings of the L-PAL Evaluator

The following pages contain a complete listing of the gedanken evaluator for L-PAL, as it has actually run in a PAL implementation (on Multics). All necessary representational issues are faced up to. As in the R-PAL listings in Chapter 3, the only variable appearing here that is not defined (other than those in PAL's primitive environment) is Error.

Any discrepencies found between the programs shown here and those shown earlier in the chapter should be resolved in favor of those shown here.

```
//                    PRELIMINARY DEFINITIONS

//  Preliminary definitions for the evaluator.



//  Selectors and constructors for the stack and control.

def t(x, y) = x  // Top of stack or control.

and r(x, y) = y  // Rest of stack or control.

and Push(x, s) = x, s  // Put new item on stack or control.

def rec Prefix(x, y) =  // Put control x at top of control y.
        Null x -> y
      | Push(t x, Prefix(r x, y) )

def r2 x = r(r x)  // Rest of (rest of (stack or control)).

and r3 x = r(r(r x))  // Rest of (rest of rest).

and 2d x = t(r x)  // Second element of stack or control.

and 3d x = t(r(r x))  // Third...

def Empty_stack = nil  // The empty stack.

//  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *

//  Tagger and tag-checkers for structures.

def Tag n s  =  s aug n  //  Tag structure s with tag n.

and Is_tag s n  =  //  Does structure s have tag n?
      Istuple s  ->  n eq s(Order s)  |  false

and Get_tag s  =  s(Order s)  //  Return the tag of s.

and Sons s  =  Order s - 1  //  Return number of sons of s.
```

```
//  Selectors, predicates and constructors for lambda-expressions
//  and lambda-closures.

def LAMBDA = '_lambda'  // Tag for lambda-expressions and closures.

def bV x = x 2       //  Select bv-part of a lambda-exp or closure.
and Body x = x 3     //  Select body part...
and Env x = x 4      //  Select environment part...

def       Test(x, n) =
               |stuple x
               -> Order x eq n
                  -> |sstring(x 1)
                     -> x 1 eq LAMBDA
                     |  false
                  |  false
               |  false
          within

     |s_lambda_exp x = Test(x, 3)
and  |s_closure x = Test(x, 4)

def Cons_lambda_exp(bV, Body)  =  //  Construct a lambda-expression.
     LAMBDA, bV, Body

and Cons_closure(L_exp, Env)   =  //  Construct a lambda-closure.
     LAMBDA, bV L_exp, Body L_exp, Env
```

```
//  Definitions and predicates for the left-hand evaluator.



//  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *

//  Items and predicates for control structure and stack.

def GAMMA       = '_gamma'
and BETA        = '_beta'
and CONSTANT    = '_constant'
and VARIABLE    = '_variable'
and ADDRESS     = '_address'    // Used only in stack.
and ASSIGN      = '_assign'     // :=
and DOLLAR      = '_dollar'
and AUG         = '_aug'
and TUPLE       = '_tuple'      //  Used only in the stack.
and ALPHA       = '_alpha'
and ETA         = '_eta'        // Used in stack for recursion.
and RETURN      = '_return'

def       Test(x, y) =
             Istuple x
             -> Order x eq 2
                -> Isstring(x 1)
                   -> x 1 eq y
                   |  false
                |  false
             |  false
     within
     Is_constant x   = Test(x, CONSTANT)
 and Is_variable x   = Test(x, VARIABLE)
 and Is_address x    = Test(x, ADDRESS)
 and Is_eta x        = Test(x, ETA)

 and Is_tuple x =
        Test(x, TUPLE)   ->  true  // Is it a constructed tuple?
     |  Test(x, CONSTANT)  ->  Null(x 2)  // Is it nil?
     |  false  // Neither.

 and Is_identifier x =  // Is x a constant or a variable?
     Test(x, CONSTANT) or Test(x, VARIABLE)

def Same_var(x, y)  =  // Are x and y the same variable?
     x 2 eq y 2
```

```
//  Call for Y_VAR is produced in Translate for rec-defs.

def Y_NAME   = 'Y#'   // The name of "Y".

and Loop_NAME = 'Loop#'    // Used in evaluation of while expressions.

and Assign_NAME = 'Assign#'    // Used in simultaneous assignment.

def Y_VAR =
     VARIABLE, Y_NAME

and Loop_VAR =
     VARIABLE, Loop_NAME

and Assign_VAR =
     VARIABLE, Assign_NAME

def Is_Y x =
     Isstring x -> x eq Y_NAME | false

and NIL =
     CONSTANT, nil

and DUMMY =
     CONSTANT, '_dummy'
```

```
//  Tags for abstract syntax tree.

def TEST      = '_test'       // test ... ifso ... ifnot ...
and ARROW     = '_arrow'      // ... -> ... | ...
and IF        = '_if'         // if ... do ...
and WHILE     = '_while'      // while ... do ...
and AP        = '_ap'         // functional application
and FN        = '_fn'         // lambda
and EQUAL     = '_equal'      // definition
and WITHIN    = '_within'
and REC       = '_rec'
and FF        = '_ff'         // function form definition
and AND       = '_and'        // 'and' definition
and COMMA     = '_comma'      // tuple maker
and LET       = '_let'
and WHERE     = '_where'
and BINOP     = '_binop'
and UNOP      = '_unop'
and PERCENT   = '_percent'


//  Taggers for tags in abstract syntax tree.

def TEST_ x y z      = Tag TEST (x, y, z)
and ARROW_ x y z     = Tag ARROW (x, y, z)
and AP_ x y          = Tag AP (x, y)
and IF_ x y          = Tag IF (x, y)
and WHILE_ x y       = Tag WHILE (x, y)
and FN_ x y          = Tag FN (x, y)
and LET_ x y         = Tag LET (x, y)
and WHERE_ x y       = Tag WHERE (x, y)
and EQUAL_ x y       = Tag EQUAL (x, y)
and WITHIN_ x y      = Tag WITHIN (x, y)
and REC_ x           = Tag REC (nil aug x)
and FF_ x y          = Tag FF (x, y)
and AUG_ x y         = Tag AUG (x, y)
and ASSIGN_ x y      = Tag ASSIGN (x, y)
and ALPHA_ x y       = Tag ALPHA (x, y)
and DOLLAR_ x        = Tag DOLLAR (nil aug x)
and BINOP_ x y z     = Tag BINOP (x, y, z)
and UNOP_ x y        = Tag UNOP (x, y)
and PERCENT_ x y z   = Tag PERCENT (x, y, z)

and AND_ x           = Tag AND x
and COMMA_ x         = Tag COMMA x

//  Note that the last two taggers are not curried, as are
//  all the others.


//  Taggers for standardized syntax tree.

def GAMMA_ x y       = Tag GAMMA (x, y)
and BETA_ x y z      = Tag BETA (x, y, z)
and LAMBDA_ x y      = Tag LAMBDA (x, y)
```

```
//  Some useful functions for transform.

def Value_of x  =  // Evaluate a control element, to put it on stack.
     x

and Val_of x  =  // De-tag a stack element, to get its value.
     x 2

def Apply x y  =
     let t = (Val_of x) (Val_of y)
     in
     Is_address t -> t | (CONSTANT, t)

and Augment_tuple x y  =
     Is_tuple x  ->  (TUPLE, Val_of x aug y)
     |  Error 'first argument of aug not a tuple'


//  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *

//                      M E M O R Y

def Initial_memory () =
     0, nil

def Contents(Memory, Address) =
     let A = Address 2
     in
     Look(Memory 2)
     where rec Look m =
           Null m -> Error 'address not in memory'
        |  A eq m 1  ->  m 2  // Found.
        |  Look(m 3)  // Keep looking.

and Update(Memory, Address, Value) =
     Memory 1, (Address 2, Value, Memory 2)

and Extend(Memory, Value) =
     let Next_C = 1 + Memory 1
     in
     let Next_M = Next_C, Value, Memory 2
     in
     (Next_C, Next_M), (ADDRESS, Next_C)

def Rval(Memory, X) =
     Is_address (X)  ->  Contents(Memory, X)  |  X
```

```
//                    E N V I R O N M E N T

//  An environment is either empty (nil), or a 3-tuple:
//        Name, Value, Environment

//  The primitive environment:

//  Define primitive environment, and provide function to look
//  up variables in the environment.

def PE =  // The primitive environment.
     Y_VAR, Y_NAME,  // for recursion
     nil

and Lookup(Var, Env) =  // Look up a variable in the environment.
     L Env  // Start looking in Env.
     where rec L e =
             Null e  ->  Error 'variable not found in environment'
           | Same_var(Var, e 1)  ->  e 2  // Found.
           | L (e 3)  // Keep looking.


//  The following function is used in applying a lambda-closure.
//  The names on the (possibly structured) bv-part 'Names' are
//  added to the environment 'Env', associated with the corres-
//  ponding part of 'Values'.  The new environment is returned as
//  the value of the function.

def rec Decompose(Names, Values, Env, Memory)  =
     test  Is_variable Names  // Is it a single variable?
     ifso  (Names, Values, Env)  // Yes, so add it to environment.
     ifnot
       (  let V = Contents(Memory, Values)
          in
          test  Is_tuple V
          ifnot Error 'conformality failure'  // Tuple applied to scalar.
          ifso
          test  Order Names eq Order(Val_of V)
          ifnot Error 'conformality failure.'  // Differing tuple lengths.
          ifso  // Process a multiple-bv part.
            (  Q 1 Env
               where rec Q n e =
                     n > Order Names  ->  e  |
                     Q (n+1) (Decompose(Names n, Val_of V n, e, Memory))
            )
       )
```

```
def rec D x =  // Standardize a definition.
     let Type = Is_tag x
     in
        Type EQUAL  ->  x  // Already OK.
     |  Type WITHIN
          -> ( let u, v = D(x 1), D(x 2)
               in
               EQUAL_ (v 1) ( GAMMA_ (LAMBDA_ (u 1) (v 2)) (u 2) )
             )
     |  Type REC
          -> ( let w = D(x 1)
               in
               EQUAL_ (w 1) ( GAMMA_ Y_VAR (LAMBDA_ (w 1) (w 2)) )
             )
     |  Type FF
          -> ( EQUAL_ (x 1 1) (Q (Order(x 1)) (x 2))
               where rec Q k t =
                     k < 2  ->  t
                     |  Q (k-1) (LAMBDA_ (x 1 k) t)
             )
     |  Type AND
          -> ( EQUAL_ L (Tag COMMA R)
               where rec L, R = Q 1 nil nil
               where rec Q k s t =
                     k > Sons x  ->  (s, t)
                     |  ( let w = D(x k)
                          in
                          Q (k+1) (s aug w 1) (t aug w 2)
                        )
             )
     |  Error 'improper node found in D'
```

```
def rec ST x  =  // Standardize abstract syntax tree.

     let Type = Is_tag x
     in
        Is_identifier x  ->  x
     |  Type BETA or Type TEST or Type ARROW
          ->  BETA_ (ST(x 1)) (ST(x 2)) (ST(x 3))
     |  Type IF
          ->  BETA_ (ST(x 1)) (ST(x 2)) DUMMY
     |  Type WHILE
         ->  ( let u = LAMBDA_ nil (ST(x 1))
               and v = LAMBDA_ nil (ST(x 2))
               in
               GAMMA_ (GAMMA_ Loop_VAR u) v
             )
     |  Type ASSIGN
         ->  ( let u = ST(x 1)
               and v = ST(x 2)
               in
               Is_tag (x 1) COMMA
                    -> GAMMA_ (GAMMA_ Assign_VAR u) v
                     | ASSIGN_ (ST(x 1))(ST(x 2))
             )
     |  Type LAMBDA or Type FN
          ->  LAMBDA_ (x 1) (ST(x 2))
     |  Type COMMA
          -> ( Q 1 NIL
               where rec Q k t =
                     k > Sons x  ->  t
                   |  Q (k+1) ( AUG_ t (ST(x k)) )
             )
     |  Type PERCENT
          -> GAMMA_ (x 2) ( AUG_ (AUG_ NIL (ST(x 1))) (ST(x 3)) )
     |  Type LET
          -> ( let w = D(x 1)  // Standardize the definition.
               in
               GAMMA_ ( LAMBDA_ (w 1) (ST(x 2)) ) (ST (w 2))
             )
     |  Type WHERE  ->  ST(LET_ (x 2) (x 1))
     |  Type AP  ->  GAMMA_ (ST(x 1)) (ST(x 2))
     |  Type BINOP
          ->  GAMMA_ ( GAMMA_ (CONSTANT, x 3) (ST(x 1)) ) (ST(x 2))
     |  Type UNOP
          ->  GAMMA_ (CONSTANT, x 2) (ST(x 1))
     |  Sons x eq 1 -> Tag (Get_tag x) ( nil aug ST(x 1) )
     |  Sons x eq 2 -> Tag (Get_tag x) ( ST(x 1), ST(x 2) )
     |  Error 'improper node found in ST'
```

```
//  The function FF flattens a standardized tree into a
//  control structure.

def rec FF(x, c)  =  // Flatten standardized tree x onto control c.


     let Type = Is_tag x
     in
        Is_identifier x -> (x, c)
     |  Type LAMBDA
          -> ( let Body = FF( x 2, nil )
               in
               Cons_lambda_exp(x 1, Body), c
             )
     |  Type BETA
          -> ( let TA = FF(x 2, nil)  // True arm.
               and FA = FF(x 3, nil)  // False arm.
               in
               FF( x 1, (BETA, (FA, (TA, c))) )
             )
     |  Type ALPHA
          -> ( let Rest = FF(x 2, c)
               in
               FF(x 1, (ALPHA, Rest))
             )
     |  Sons x eq 2 -> FF( x 2, FF( x 1, (Get_tag x, c) ) )
     |  Sons x eq 1 -> FF( x 1, (Get_tag x, c) )
     |  Error 'improper node found in FF'


//  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *

def Translate Program  =  // The routine that does all the work.
     FF( ST Program, nil )
```

```
//  State transformations for the LPAL Evaluator.


def Eval_constant(C, S, E, M) =
     r C, Push(Value_of(t C), S), E, M

and Eval_variable(C, S, E, M) =
     let New_S = Lookup(t C, E)  // Look up top of C in E.
     in
     r C, Push(New_S, S), E, M

and Eval_lambda_exp(C, S, E, M) =
     let New_S = Cons_closure(t C, E)
     in
     r C, Push(New_S, S), E, M

and Do_alpha(C, S, E, M) =
     r C, r S, E, M

and Do_assign(C, S, E, M) =
     let New_M = Is_address(t S) -> Update(M, t S, Rval(M, 2d S)) | M
     in
     r C, Push(DUMMY, r2 S), E, New_M

and Do_dollar(C, S, E, M) =
     r C, Push( Rval(M, t S), r S ), E, M

and LtoR(C, S, E, M) =  // Replace L-value at stack top by R-value.
     let New_S = Contents(M, t S)  // Look up stack top in memory.
     in
     C, Push(New_S, r S), E, M

and Do_conditional(C, S, E, M) =
     let Selected_arm = (Val_of(t S) -> 3d | 2d) C
     in
     Prefix(Selected_arm, r3 C), r S, E, M

and Do_aug(C, S, E, M) =  // aug
     let New_M, x = Is_address(2d S) -> (M, 2d S) | Extend(M, 2d S)
     in
     let V = Augment_tuple (t S) x
     in
     r C, Push(V, r2 S), E, New_M

and Do_return(C, S, E, M) =
     r C, Push(t S, r2 S), 2d S, M
```

```
and Apply_closure (C, S, E, M) =
      let New_M, Rand = Is_address(2d S) -> (M, 2d S) | Extend(M, 2d S)
      and Rator = t S
      in
      let New_C = Prefix(Body Rator, Push(RETURN, r C))
      and New_S = Push(E, r2 S)
      and New_E = Decompose(bV Rator, Rand, Env Rator, New_M)
      in
      New_C, New_S, New_E, New_M

and Apply_constant(C, S, E, M) =
      let V = Apply (t S) (Rval(M, 2d S))
      in
      r C, Push(V, r2 S), E, M

and Apply_tuple(C, S, E, M) =
      let V = Apply (t S) (Rval(M, 2d S))
      in
      r C, Push(V, r2 S), E, M

and Apply_Y(C, S, E, M) =
      let V = ETA, 2d S
      in
      let New_S = Push(2d S, Push(V, r2 S) )
      in
      C, New_S, E, M

and Apply_eta(C, S, E, M) =
      Push(GAMMA, C), Push(t S 2, S), E, M
```

```
//  Main program for the LPAL Gedanken Evaluator.


def Transform(C, S, E, M)  =  // Do one step of an evaluation.
     let A = C, S, E, M
     and x = t C  // Top of control.
     in
        Is_constant x     -> Eval_constant A
     |  Is_variable x     -> Eval_variable A
     |  Is_lambda_exp x   -> Eval_lambda_exp A
     |  x eq ALPHA        -> Do_alpha A  // semicolon
     |  x eq ASSIGN       -> Do_assign A
     |  x eq DOLLAR       -> Do_dollar A
     |  Is_address(t S)   -> LtoR A  // R-value to top of stack.
     |  x eq BETA         -> Do_conditional A
     |  x eq AUG          -> Do_aug A
     |  x eq RETURN       -> Do_return A
     |  x eq GAMMA        ->
          ( let r = t S
            in
               Is_closure r       -> Apply_closure A
            |  Is_constant r      -> Apply_constant A
            |  Is_tuple r         -> Apply_tuple A
            |  Is_Y r             -> Apply_Y A
            |  Is_eta r           -> Apply_eta A
            |  Error 'improper rator'
          )
     |  Error 'bad control'


def rec Evaluate(C, S, E, M) =
     Null C -> (Rval(M, t S), M) |
     Evaluate(Transform(C, S, E, M))

def Gedanken_evaluator Program =
     let Control_structure = Translate Program
     and M0 = Initial_memory nil
     in
     Evaluate(Control_structure, Empty_stack, PE, M0)
```

Chapter 5

JUMPS AND LABELS

The one remaining feature to be added to L-PAL to make it J-PAL (and therefore, complete PAL) is the "goto" statement. Note the PAL program with a label at the beginning of Chapter 4 in (4.1-1c). The idea is that, in a program such as

```
... ; L: S1; ... ; goto L; ...                    (5.0-1)
```

obeying the "goto L" statement causes the successor of that statement to be the statement S1. We say that the appearance of "L:" before S1 is a label, which labels the statement. Thus the goto statement merely alters the "flow of control" through the program.

But the problem is harder than this. Clearly in (4.1-1c) or in (5.0-1), obeying the goto changes only the control, since the stack is empty both before and after the goto and the environment at both places is the same. Consider instead the program

```
S1;
M: S2;
(    let x = ...
     in                                            (5.0-2)
     S3; if ... do goto M; S4
);
S5;
```

here obeying the goto must change more than just C: It must also change E, since x is in the environment at the place where the goto appears and is not in the environment at the place in the program where the label M appears. Clearly this situation is more complex than that of the preceeding paragraph. To distinguish them, we call the goto of (5.0-2) a skip, and that of (5.0-1) a hop. The distinction is that execution of a skip causes layers of the environment to be discarded, while in a hop the environment is left unchanged.

There is an even more complex type of goto: one which we call a jump. Consider the following:

```
1     let x = nil
2     in
3     S1;
4     (    let y = 0
5          in                                       (5.0-3)
6          M: S2; x := M; S3
7     );
8     S4;
9     if ... do goto x;
10    ...
```

(The numbers on the left are not part of the program but are for reference below.) The scope of x is clearly the entire program after line 2. Thus obeying the assignment statement on line 6 causes x to denote a value which is a label. (We have yet to say what sort of R-value a label is. This and the next few examples should give us more ideas of what is needed.) Thus on line 9 the goto leads to the label M. The environment at M has more information in it than does the environment that exists on line 9. Normally, the environment layer for y would have disappeared (in some sense) as we passed the right parenthesis on line 7. Now somehow we must remember it, so that it can be reinstated as part of the goto. This requirement suggests that part of the value of a label must be an environment.

We distinguish between skips and jumps to make one point: There exist many languages that implement skips, but very few which implement jumps. Skips, called "non-local gotos", are available in Algol, PL/I and related languages, while the only languages that implement jumps seem to be those which, like PAL, have been strongly influenced by the works of Strachey and Landin.

J-PAL introduces one other linguistic facility -- the punctuations valof and res. We see later that they can be treated quite economically as sugarings of jumps and labels. Briefly, the value of the expression

valof E (5.0-4a)

(which may be read as "value of E") is that of E, with the proviso that an occurrence in E of the expression

res F (5.0-4b)

(which may be read as "result is F") causes the value of F to be taken immediately as the value of the entire expression (5.0-4a), regardless of what other text may appear in E. The effect of a res is, in most cases, a hop, but it is quite possible to write programs in which res produces a skip or even a jump.

Example: Note the distinction between a skip and a jump: In the former, environment layers are discarded, whereas in the latter environment layers that normally would have been discarded earlier in the evaluation are reinstalled. Let us look at another example of a jump, a rather trivial one which nonetheless suggests some possibilities:

```
1    let F () =
2        true
3        -> (Print 'A'; L)
4        | (L: Print 'B')
5    in                                          (5.0-5)
6    let x = F nil
7    in
8    if Islabel x do goto x;
9    Print 'C'
```

This is a complete J-PAL program which runs on the computer, producing printed output. Before proceeding, the reader might want to attempt to deduce what that output will be.

Lines 1 to 5 define a function F, and on the sixth line x is defined to be the value produced by applying F to nil. What will that value be? Actually, that is not quite the proper question, since applying F to nil may have a side effect as well as returning a value. The body of F (lines 2 through 4) is a conditional expression with "true" as the boolean, so the "true" arm on line 3 is obeyed. Doing so causes 'A' to be printed (a side effect) and L to be returned as the value of the application, the value to be associated with x. On line 8 we ask if x denotes a label. (The predicate "Islabel" is in PAL's primitive environment with the obvious meaning. See Appendix 4.1 of the PAL Manual.) Since x does in fact denote a label, the goto is executed.

Now the explanation gets sticky. The label L appears on line 4, and going to it certainly causes 'B' to be printed. But what happens next? That is, what does the evaluator do after printing 'B'? It is evidently done with the body of F and about to do a return, but to what point of the program does it return? The answer is this: Going to L reinstalls the complete machine state (CSE) that existed when L was "declared". That declaration took place after F was applied to nil in line 6, and the state includes (in C and S) information that leads to return to that place on completion of evaluation of the body of F. Thus we are back in the invocation of F that appears on line 6. Time seems to have been backed up. Are we in a loop? No, since now F returns a different value: The value of the combination whose rator is Print and whose rand is 'B'. This value, dummy, is now associated with x on line 6. Since x does not denote a label, we do not do the goto on line 8 but instead print 'C' on line 9 and the program terminates, having printed 'ABC'. What could be simpler?

This example raises quite a few interesting questions, such as the following:

- Just what sort of R-value is label? How can we get enough information into x so that going to it puts us back to the application on line 6?
- What is the scope of a label? Note that the L returned as a value on line 3 is the "same one" that appears before a colon on line 4. As it happens, L's scope is all of the text on lines 2 through 4, but we need rules.
- "L" has the syntax of a variable. Can it be updated?
- At what points in a program may we place a label?

Answering these questions takes most of the rest of the chapter.

Organization of the Chapter: Our presentation of J-PAL differs from that of L-PAL in Chapter 4. We start off with an overview of the J-PAL gedanken evaluator, and then treat various points in terms of how they are handled by it

-- usually by showing differences from the L-PAL case. Section 5.1 shows how R-PAL and L-PAL constructs are processed by the J-PAL evaluator, and section 5.2 how jumps are done. The chapter ends with a complete listing of the J-PAL gedanken evaluator. Blackboard evaluation, which was so very important to our presentation of R-PAL and was of interest in L-PAL, is less important for J-PAL. The reason is that it is difficult to find meaningful programs that both exploit the power of jumps (as opposed to hops) and are sufficiently short that blackboard evaluation is feasible. The program in (5.0-5) is such a program, and a blackboard evaluation of it appears in section 5.2.

5.1 Changes to the Gedanken Evaluator

Our presentation of J-PAL's semantics requires that we specify the J-PAL gedanken evaluator. Because the changes from the L-PAL evaluator are extensive, we discuss in this section how L-PAL constructs are handled by the J-PAL evaluator, and in section 5.2 how to do jumps.

Overview of the J-PAL Gedanken Evaluator

Our logical bootstrapping procedure permits us to specify J-PAL in terms of an L-PAL program, and we take extensive advantage of this ability. The overall structure of the evaluator is as follows:

```
// definitions for representational issues
def C, S, E, M = nil, nil, nil, nil
// definitions for memory
def Print_ x = // user-callable Print routine
// definitions for environment                                    (5.1-1)
def Translate P = // and also D, ST, LL, FF
def  )
...  }   programs for Transform
def  )
def Transform () =
def Gedanken_evaluator Program = ...
```

Note that C, S, E and M are variables global to essentially all of this. For example, whereas in L-PAL we had

```
def Eval_constant (C, S, E, M) =
     r C, Push (Value_of (t C), S), E, M
```

the corresponding J-PAL program is

```
def Eval_constant () =                                            (5.1-2)
     C, S = r C, Push (Value_of (t C), $ S)
```

Since C and S are global to this definition (i.e., already in the environment), and since we are writing in L-PAL, we merely update them to hold the desired values. E and M, not being changed by this function, are not mentioned. The unsharing operator $ is needed to prevent a disasterous sharing: The second

component of the 2-tuple returned by Push (see (3.5-3) on page 3.5-154) shares with Push's second argument. Since S is updated in the normal operation of Transform, it is essential that no part of the stack (or of anything else) share with it.

The main program for the evaluator is

```
def Gedanken_Evaluator Program =
     Initialize_memory nil;
     C := Translate Program;
     S := Empty_stack;
     Initialize_env nil;
     until Null C do Transform nil
```
(5.1-3)

This program looks quite different from that of (4.2-7) for L-PAL. For one thing, the function Evaluate is gone, all of its work being done by the until construct in the last line. Further, the entire evaluation is for effect, so that a Print program must evidently be provided. Note that memory is initialized before the call to Translate: We see later that parts of the control structure are stored in cells in the memory.

Memory: We start our discussion with the M component of the evaluator. Extending memory in the L-machine was made somewhat awkward by the need to carry M along as an explicit argument in every machine state transformation. Having M as a global variable facilitates a simpler treatment.

As before, we represent a memory by a 2-tuple, whose first component is that integer which is the last address used (initially zero), and whose second component is a Mem:

```
A Mem is either empty (nil)
or it is a 3-tuple, whose components are
     an address,
     a contents, and
     a Mem.
```

However, there exists only one memory: the one stored in the global variable M. The first function we consider serves to initialize that memory:

```
def Initialize_memory () =
     M := 0, nil
```
(5.1-4)

Note that this function has no useful value but is executed solely for its effect. Now we need the three function Extend, Update and Contents, but we must first select a representation for addresses. The address for cell k (where k denotes some integer) is represented by the 2-tuple

(ADDRESS, k)

We have then

```
def Extend Value  =  // Find a new cell to hold value.
     let k = 1 + M 1 // Address of next free cell.
     in                                                    (5.1-5a)
     M := k, (k, Value, M 2); // Create new memory.
     (ADDRESS, k) // Return the new address.

def Update (Cell, Value) =
     M := M 1, (Cell 2, Value, M 2)                        (5.1-5b)

def Contents Cell =
     let c = Cell 2
     in
     Look (M 2)                                            (5.1-5c)
     where rec Look m =
                 Null m -> error
           | m 1 eq c -> m 2
           | Look (m 3)
```

Compare these definitions with those in (4.1-13). Finally, the following two definitions are frequently convenient:

```
def Rval x =
     Is_address x -> Contents x | x
and Lval x =                                               (5.1-6)
     Is_address x -> x | Extend x
```

A function such as Lval could not be written for the L-Machine, since it would have to return the new memory and would thus be no more than Extend.

Transform: The major change to Transform has to do with the fact that we need no longer pass around the (C, S, E, M) state. We have

```
def Transform () =
    let x = t C
    in
       Is_constant x     -> Eval_constant nil
     | Is_variable x     -> Eval_variable nil
     | Is_lambda_exp x   -> Eval_lambda_exp nil
     | Is_address x      -> Hop nil
     | Is_delta x        -> Make_labels nil
     | x eq ALPHA        -> Do_alpha nil
     | x eq ASSIGN       -> Do_assign nil
     | x eq RETURN       -> Do_return nil
     | Is_address (t S)  -> LtoR nil                          (5.1-7)
     | x eq BETA         -> Do_conditional nil
     | x eq GO_TO        -> Jump nil
     | x eq DOLLAR       -> Do_dollar nil
     | x eq AUG          -> Do_aug nil
     | x eq GAMMA
         -> ( let r = t S // The rator.
              in
                Is_closure r  -> Apply_closure nil
              | Is_constant r -> Apply_constant nil
              | Is_tuple r    -> Apply_tuple nil
              | error
            )
     | error
```

Hop, Make_labels and Jump are the only new functions. The former is needed because of the fact, alluded to earlier, that parts of the control are stored by Translate into the memory, so that an address may appear as a control item. The last two functions are used to declare labels and to implement goto, respectively, and are discussed in section 5.2. The code for Hop is quite simple:

```
def Hop () =                                                  (5.1-8)
    C := Prefix (Contents (t c), r C)
```

Eval_constant has already been given in (5.1-2), and Eval_variable and Eval_lambda_exp differ similarly from their L-PAL counterparts. For assignment we have

```
def Do_assign () =
    if Is_address (t S) do
        Update (t S, Rval (2d S));                            (5.1-9)
    C, S := r C, Push (DUMMY, r2 S)
```

Note how much simpler this code is than that in the L-machine, because it is not here necessary to carry the memory around all of the time.

Do_return and LtoR are similar to their L-PAL counterparts. The handling of conditionals differs slightly:

```
def Do_conditional () =
    let V = Contents ((Val_of (t S) -> 3d | 2d) C)
    in                                                    (5.1-10)
    C, S := Prefix (V, r3 C), r S
```

The control structure produced by the J-PAL Translate for a conditional always has the code for each arm stored in memory cells, corresponding to our R-PAL and L-PAL blackboard decision to abbreviate each arm with a δ.

For Do_aug we have changes similar to those for Do_assign:

```
def Do_aug () =
    let V = Aug (t S) (Lval (2d S))
    in                                                    (5.1-11)
    C, S := r C, Push (V, r2 S)
```

The three Apply functions are little changed, Apply_closure being slightly simplified in use of memory. The code is

```
def Apply_closure () =
    let R = t S  // the rator
    and Rand = Lval (2d S)
    in                                                    (5.1-12)
    let New_C = Prefix(Contents(Body R), Push(RETURN, r C))
    and New_S = Push ($ E, r2 S)
    and New_E = Decompose (bV R, Rand, Env R)
    in
    C, S, E := New_C, New_S, New_E
```

Note that Apply_Y and Apply_eta are missing: Recursion is achieved through use of a library routine.

Translate: In going from the R-machine to the L-machine, most of the changes made were to Transform, the changes to Translate being confined for the most part to providing straightforward processing for the new node types. Now we must make extensive changes to Translate, the new main program being

```
def Translate Program =                                   (5.1-13)
    FF (LL (ST Program), nil)
```

ST differs from the L-PAL version only in that the present one processes four new node types: COLON, GO_TO, VALOF and RES. The kind of processing done is unchanged. The function LL solves the problem of scope of labels, and we defer discussion of it to section 5.2, in which we consider labels.

One aspect of the J-machine's Translate is appropriately discussed here, and that is its use of memory. In blackboard evaluation in R-PAL and L-PAL, we have consistently abbreviated certain control structures: λ-bodies, arms of conditionals, and (in L-PAL) the right sons of semicolon nodes. Although this use of abbreviation was only for convenience and did not affect the formalization, the use of abbreviation in blackboard handling of the while construct is critical. (See (4.2-3) on page 4.2-223.) Since we want to make

loops in the control to handle while, we now formalize the idea of abbreviations.

The idea of an abbreviation is that a single mark, such as δ_3, stands for some complex item. The effect we want is that, on encountering δ_3 in a suitable context, we look it up in a table and replace it by what we find there. (That is precisely what we have been doing in blackboard evaluation, all along.) Although we could easily implement such a table of abbreviations, it turns out that the memory has all of the needed properties. Thus we choose to put pieces of control structure into memory and to permit an address as a control item. For example, the control structure for

while B do E

would be just like that shown in (4.2-3), except that each δ would be replaced by a σ.

Hops, Skips and Jumps

In the introduction to the chapter, the trichotomy of hops, skips and jumps was introduced, in increasing order of complexity. Recall that execution of a hop involves changing only C and that execution of a skip or jump involves also changes in E. (We see later that S is also involved.) We wish now to study the distinction in more detail. We start with consideration of iteration, leading to a discussion of hops.

Iteration: The concept of iteration is a fundamental one in programming, and syntactic sugaring for it appears in one guise or another in almost every programming language. In PAL, iteration appears explicitly only as the while and until constructs, the syntax of which

while E1 do E2
until E1 do E2

we have already encountered in L-PAL. Still other variants on the conditional and iteration occur in other languages, and it seems clear that program readability is greatly enhanced when a rich catalogue of possible syntactic forms is available.

Despite its importance, however, the concept of iteration seems difficult to define abstractly. The difficulty centers on restricting the concept in a meaningful way. For example, the following PAL program (which we have encountered before as (4.1-1c) at the beginning of Chapter 4) involves looping to the label L, and may be thought of as an "iterative" definition of the factorial function.

```
def f n =
    let r, k = 1, 0
    in
    L:  if k eq n do goto M;
        k := k+1;                                        (5.1-14a)
        r := r*k;
        goto L;
    M:  r
```

In its overall semantics, (5.1-14a) equivalent to

```
def f n =
    let  r, k = 1, 0
    in                                                   (5.1-14b)
    until k eq n do
      (  k := k + 1;
         r := r * k
      );
    r
```

In point of fact we have seen that labels in PAL (and in many other languages--for example, ALGOL) are more powerful than is necessary to accommodate iteration; in other words, (5.1-14a) exploits only a subset of the properties of labels, precisely that subset which we have referred to as hops.

In the same vein, recursion includes iteration as a special case; indeed, our initial explication of while and until in L-PAL was in terms of the recursive function Loop# of (4.1-20). Operational insight into the distinction between full recursion and iteration is provided by the two following programs, each of which again defines the factorial function.

```
def rec f n = n eq 0 -> 1 | n*f(n-1)                     (5.1-15a)
```

```
def f n =
    g (1, 0)
    where rec g (r, k) =                                 (5.1-15b)
        k eq n -> r | g ( r*(k+1), k+1 )
```

Here (5.1-15a) depends upon the full power of rec, in the sense that every intermediate result (one for each call to f) must be stored until the escape condition "n eq 0" is met, at which point evaluation unwinds. By contrast, on calls to g in (5.1-15b) the results of previous calls are irrelevant: All necessary information is carried from one call to the next by means of the bound variables r and k. Thus (5.1-15b) -- which resembles closely (5.1-14b) in effect and thus can be considered to be iterative -- represents a degenerate case of recursion in which the full potential of rec is not required. Evidently the degeneracy hinges upon the fact that the recursive variable f in (5.1-15a) is encountered as part of a rand (in this case of the operator "*"), whereas in (5.1-15b) the recursive variable g occurs only in the role of a rator in a combination which is itself the entire value of a recursive call. We conclude that any abstract distinction between recursion and iteration depends critically

on the constraints in terms of which the discussion is circumscribed, and pass on instead to study of how the distinction manifests itself in PAL.

Conceptually at least, we can view the J-PAL implementation of while as being achieved by a loop in the control structure. Thus

while B do E

leads to the control structure

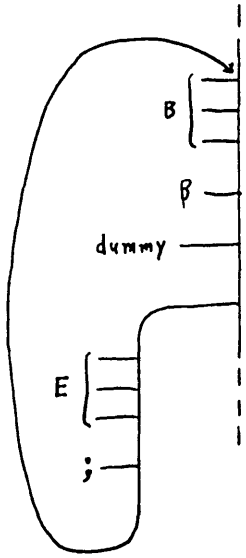


Figure 5.1-1: Reentrant control for the construct
while B do E

It is more convenient to use abbreviations than to attempt actually to construct such a loop, so we produce instead something like

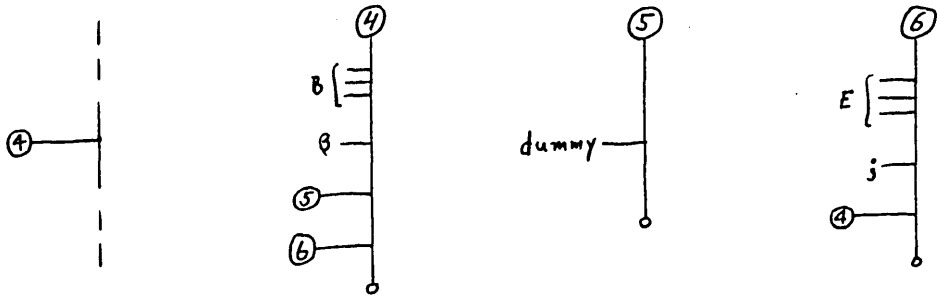


Figure 5.1-2: Control for while, with abbreviations.

Review the code for Hop in (5.1-8) and for Do_conditional in (5.1-10) to see that the effect is as desired. On encountering the control item σ_4, indicated in the figure by a circled 4, the J-machine prefixes onto the control that control structure stored in cell 4, and then obeys it. Thus the code for B is obeyed. If the result is true, the code in σ_5 is loaded, leading to execution of E followed again by σ_4. This continues until evaluation of B leads to false.

Hops: We have used the term hop to refer to that linguistic facility which requres no more than a reentrant loop in the control structure, in contradistinction to the more general linguistic facilities which we call "skips" or "jumps". The node of the control structure at which reentry occurs is called the entry point.

Although while and until are the only examples of hops in PAL, it should not be concluded that hops are an unimportant construct. Indeed, in most languages hops are the only explicit facility whereby the programmer can specify deviations from the normal flow of control. For example, by our definition labels in FORTRAN are hop entry points. It would have been possible -- even preferable, from the point of view of efficiency -- to have chosen entry points as the value of labels in the design of PAL. Since our objectives are pedagogic rather than practical, however, the decision was made to opt for the full generality of jumps.

From the point of view of use (as opposed to implementation), the principle distinction between a hop and a jump lies in the scope within which it is effective. In the normal course of executing a PAL program, the J-machine creates a new environment each time a λ-closure is applied. Conversely, on exit from a λ-body the newly created environment is lost and the old control resumed. A little thought should make it clear that in these circumstances -- indeed whenever one is dealing with a language that entails variables whose scope is limited -- it is not meaningful to hop to an entry point from outside the λ-body that contains it. For example, consider the (defective PAL) program

```
let  x = 5
in
[let  y = 7 in L: x := x+y];                                    (5.1-16)
if x le 15 do goto L; // Error
x+1
```

and assume that the value of the label L were an entry point. The execution of the "goto L" command would involve hopping into the scope of y after having discarded its definition, so that the evaluation of y in the assignment to x would abort.

A second difficulty with (5.1-16) is that in PAL the definition of the label L is itself unknown at the point of call, so that the program is defective on two counts rather than one. That is, the scope of L does not include the goto statement. But the first difficulty can occur without the second when assignment of labels is allowed. As an example, consider the valid program

```
let x, M = 5, nil
in
[let y = 7 in M := L; L: x := x+y];                             (5.1-17)
if x le 15 do goto M;
x+1
```

Here M is known at the point where the goto appears, and its value is updated earlier by the assignment "M := L". If the value of the label L consisted

merely of an entry point (which in PAL it does NOT), then executing the goto would again entail hopping into the scope of y after its definition had been lost, with ensuing chaos.

From the foregoing examples it might seem at first that hops are meaningful provided only that they occur within a single λ-body. But trouble can arise even when this constraint is met, as witnessed by the (again, defective PAL) program

```
let  y = 4
in
test  (L: y) < 3                                        (5.1-18)
ifso  y
ifnot (y := y-2 ; goto L) // Error
```

The reader should see that the stack is empty just before the interpretor begins execution of "goto L". If the effect of this command were merely to reenter the control structure, evaluation would abort because there would be no second argument for the operator "<" on the stack. We conclude that the value of a label must comprise more than just an entry point if the programmer is to have freedom to label arbitrary subexpressions, even if all flow of control is confined within a single λ-body.

Like (5.1-16), the example just given is defective because of PAL's scope rules for labels (which are discussed later). It suffices for the moment to state that the program

```
let y, M = 4, nil
in
test   (M := L; L: y) < 3                               (5.1-19)
ifso   y
ifnot  (y := y-2; goto M)
```

is correct PAL and would also abort if the value of a label in PAL were an entry point.

The limitations of hops which we have just explored are not encountered in languages such as FORTRAN by virtue of several constraints imposed by the language designers. Typical constraints are:

(a) A label cannot be assigned, be passed as an argument, or be the value of a function.

(b) The scope of all variables is the entire program.

(c) Labels may only be placed in restricted contexts.

Much of the time such constraints are innocuous, but they are occasionally troublesome. The alternative, which is to generalize the concept of a hop to the concept of a jump, is explored in the next section. It appears that both hops and jumps deserve a place in our catalogue of linguistic facilities, but that the latter should be used sparingly.

5.2 Jumps

It should be acknowledged at the outset that very little guidance, either in mathematical theory or in computational practice, is at hand to guide a language designer in establishing what the semantic intent of a generalized jumping construct should be. Without apology -- but also without much confidence that a more restrictive formulation may not ultimately prove more acceptable -- we therefore proceed on the basis of remedying the limitation of hops.

Before starting, we should make one point. Programs involving a maze of jumps tend to be hard to read and understand, as anyone who has encountered FORTRAN-2 will surely attest. (The language has no sugaring for conditionals.) Indeed, it has been suggested by E. W. Dijkstra (1968) that labels should be abolished entirely from programming languages. Nonetheless, there are instances in which the full power of jumps as they are available in PAL seems to be called for, although simple examples illustrating such a need are hard to come by. The modern trend in programming language design is toward providing enough syntactic sugaring for hops so that labels are rarely needed, and this trend has been followed in PAL. In addition to the various sugarings discussed so far, the valof-res construct discussion later in this section is available.

Having made these points, let us adopt a set of desidarata regarding the use of labels in PAL:

(a) A label identifier should be a variable.

(b) A label should be allowable on any subexpression of a program. That is, programs such as (5.1-19) should be allowed.

(c) The labelling of any subexpression should not affect the meaning of a program if the label is not jumped to during execution.

(d) If the label on a subexpression is L, then execution of the jump command "goto L" with the memory in some state should be equivalent to entering the subexpression normally, but with memory as it is at the occurrence of the goto.

(e) An R-value of type label may be used just as may any other value. Specifically, it may be assigned as the value of a variable, may be a component of a tuple, may be passed as an argument to a function, may be the value of a function, etc.

By "entering the subexpression normally" in (d) we mean entering it NOT via a jump, but rather via the control structure which would exist if all labels (i.e. writings of the form "⟨NAME⟩:") were excised from the program. This point is discussed further later.

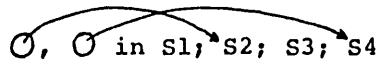
These desiderata suffice to specify the semantics of jumps, but do not specify the scope of labels. We defer detailed treatment of label scope until after we have extended the state-transition function Transform to accommodate evaluation of, and jumping to, a label.

The Value of a Label

The examples we have seen make it clear that the value of a label is a complete machine state with three components: C, S and E. Further, point (a) above makes it clear that the syntactic device "L:" declares L to be a variable whose initial value is of type label. Thus, for example, the segment

```
let ... in S1; L: S2; S3; M: S4                    (5.2-1a)
```

is treated somewhat as if the programmer had written



(5.2-1b)

Here let# is written to suggest that this is a rather special sort of let, one which serves to define labels. The arrows emanating from the circles suggest that L labels S2 and that M labels S4. Of course the programmer cannot write let# -- (5.2-1b) is meant only to be suggestive.

The issue of scope of labels, which we defer until later, has to do with just where the let# is placed. The reader should rest confident that the examples we give are correct.

Examples: We need now some examples of the use of labels, to see some of the implications of desiderata at the beginning of this section. The following program segment makes clear that labels really are variables, as suggested by (a):

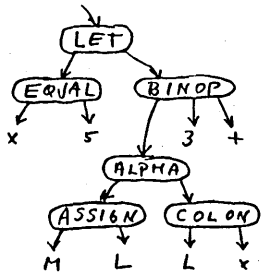
```
A:  Print 'A';
    goto  B;
B:  Print 'B';                                     (5.2-2)
    B :=  C;
    goto  A;
C:  Print 'C'
```

In the absence of the assignment statement the program would loop, printing 'ABABAB...'. However, the semantics of "goto B" involves transfer to whatever value the variable B has at the time the goto is obeyed. The second time around, that value is C, so the program prints 'ABAC' and terminates.

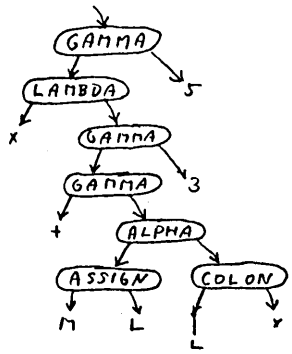
To get further insight, consider the fragment

```
let x = 5
in                                                 (5.2-3)
(M := L; L: x) + 3
```

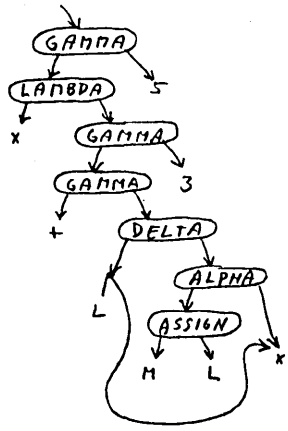
and assume that this lies within the scope of the variable M. The syntax tree for this fragment is shown in (a) of Figure 5.2-1. The new node here is COLON, whose left son is a label and whose right son is the tree that is labelled. The standardized tree is shown in (b) of the figure. Note that the left son of the COLON node is a one-tuple whose component is L. Otherwise there is nothing new here.



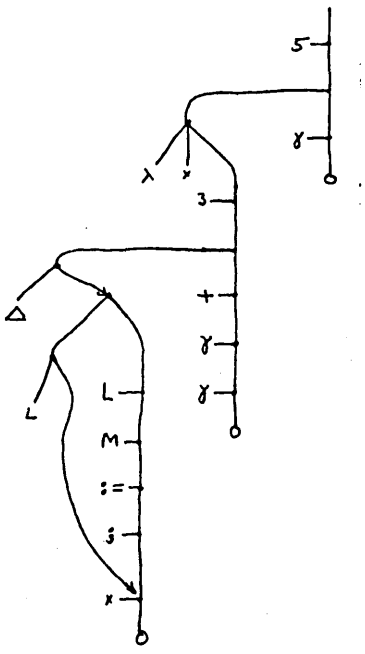
(a) Syntax tree



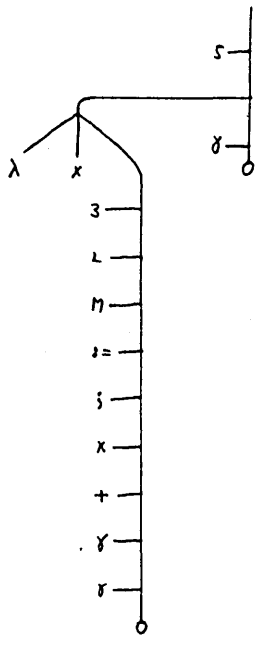
(b) Standardized tree



(c) Output of LL



(d) Control structure



(e) Control without label

Figure 5.2-1: Trees for (5.2-3)

What we want now is a control structure for this. An intermediate result on the way to creation of control is shown in (c). (This is the output of the function LL mentioned in (5.1-13) on page 5.1-258.) This picture is somewhat simplified, but it is substantially correct for our present purposes. The Δ node is placed at the point where the label is to be declared. Its left son is (in this case) a 2-tuple whose components are the label and the place labelled. The right son of the delta is that piece of tree which is the scope of the label.

In dealing with λ-expressions we have found it convenient to set off the body as a separate control item. A λ-body is then evaluated in the new environment created at the time the λ-closure is applied, thereby clearly delineating the scope of its bound variable. Since the effect of obeying the control item DELTA includes creation of a new environment, it is consistent to set off the right son in a similar way. Thus, we are led to flatten the program representation as shown in Figure 5.1-1(d). By contrast, if the writing "L:" were deleted from (5.2-3) the flattened control structure would have been as shown in Figure 5.1-1(e). The distinction between (d) and (e) lies in insertion of the control item Δ, whose left son is the label name and entry point, and right son is the expression over which L is to be known. This latter we refer to hereafter as a label body.

Details of how the control structure of (d) is generated from the abstract syntax tree of (a) are covered later during discussion of Translate. For the moment, it suffices to observe that this control structure (however it may have been produced) exhibits certain important information in a convenient form:

(1) The control item Δ provides forewarning that a label is about to be declared.

(2) The label name and entry point are readily accessible.

(3) The label body is manifest.

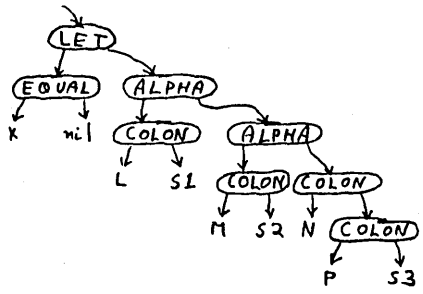
In this program there is only one label declared at the delta node, but it may be the case that several are declared. For example, conventional programming practice permits the possibility of forward jumps, as in

S1; if ... do goto L; S2; L: S3

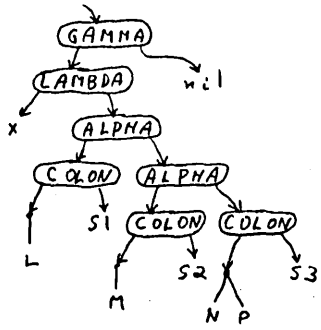
The scope of L is the entire writing. (4.1-1c) contains a forward jump. Let us consider the following example

```
let x = nil
in                                          (5.2-4)
L: S1; M: S2; N: P: S3
```

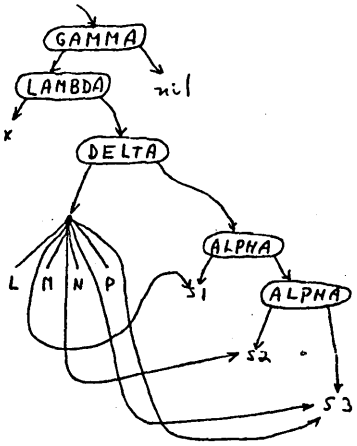
Here S1, S2 and S3 may be any statements, including possibly the case that S1 is "goto N". Figure 5.2-2 shows successively the syntax tree, the standardized tree, the output of LL and the control structure for this program. (We continue to simplify slightly.) Note in (b) that the left son of the third COLON node is a 2-tuple, whose components are the two labels of S3. In (c) the Δ node is



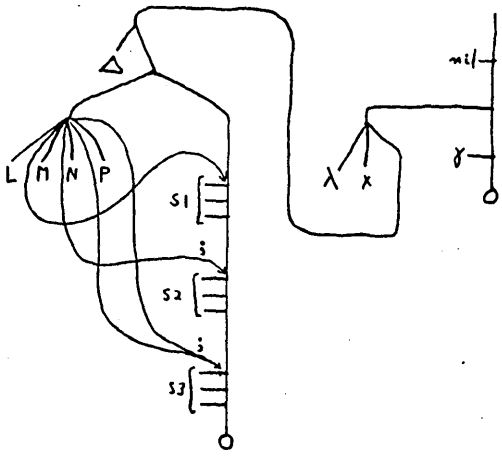
(a) Syntax tree



(b) Standardized tree



(c) Output of LL



(d) Control structure

Figure 5.2-2: Trees for (5.2-4)

placed so as to declare all of the labels on entry to the λ-body, and (d) shows the control. Four labels being declared, the DELTA node has an 8-tuple as the left son. The first, third, fifth and seventh elements are the variables to be declared, and the even-numbered elements point to the relevant part of the control.

Blackboard Evaluation: We have seen some examples of control structures for labels, and we now concern ourselves with evaluation. We consider again the program of (5.0-5), which we repeat here for convenience:

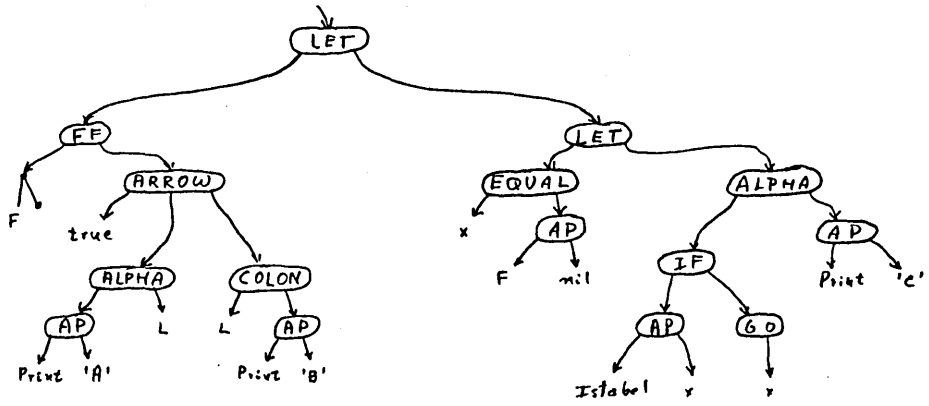
```
let F () =
    true
    -> (Print 'A'; L)
    |  (L: Print 'B')
in                                                    (5.2-5)
let x = F nil
in
if Islabel x do goto x;
Print 'C'
```

The abstract syntax tree, standardized tree and output of LL are shown as (a), (b) and (c) of Figure 5.2-3. (Much of (c) is not drawn, since it is identical to the corresponding part of (b). To the extent that COLON nodes are absent, the output of LL is identical to its input.) The intent of the circled 1 in (c) is that the node below it is stored in a memory cell -- σ_1 in this case.

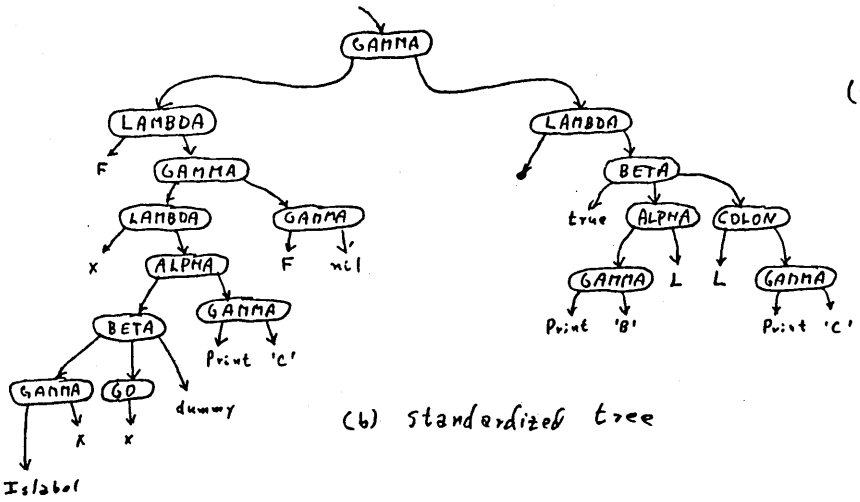
Figure 5.2-4 shows two drawings of the control structures produced by FF. Again, circled numbers denote addresses. The upper drawing shows the control structure as a whole, and the lower drawing shows separately the contents of each cell. The reader should satisfy himself that the drawings are equivalent. Note the correspondence between the occurrences of σ_1 in this figure and in Figure 5.2-3(c): This correspondence will be seen to be critical to the successful operation of Translate.

Figure 5.2-5 shows the AE form of (5.2-5) and the control sequence for blackboard evaluation. So as to emphasize the relation between blackboard evaluation and the gedanken evaluator, each δ refers to the same piece of control as the corresponding σ. (Of course, the numbering of the δ's is arbitrary -- any eleven numbers in any order could be used. The numbering shown here happens to be that actually produced by FF.)

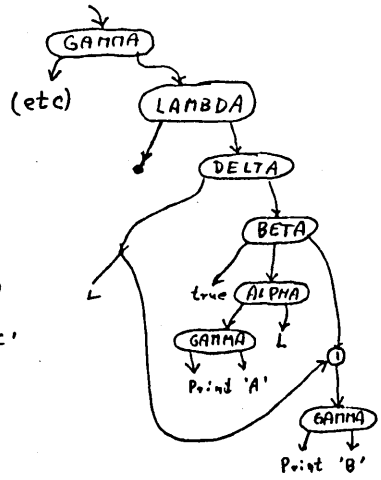
Figure 5.2-6 shows blackboard evaluation in a J-PAL evaluator. The column at the right labelled P shows printed output, an entry being shown there each time the identifier Print is applied. The column headed J contains line numbers actually referred to as part of the blackboard evaluation, as opposed to the column to its left which appears only to facilitate reference to the evaluation in this text. Now note the appearance of $'\Delta^{L}_{10}$ on line 6. The subscript indicates that the piece of control structure abbreviated as δ_{10} is to be executed next. The two superscripts indicate that identifier L is to be declared as a label, referring to the entry point δ_1. The line immediately



(a) Syntax tree



(b) Standardized tree



(c) Output of LL (partial)

Figure 5.2-3: Trees for (5.2-5)

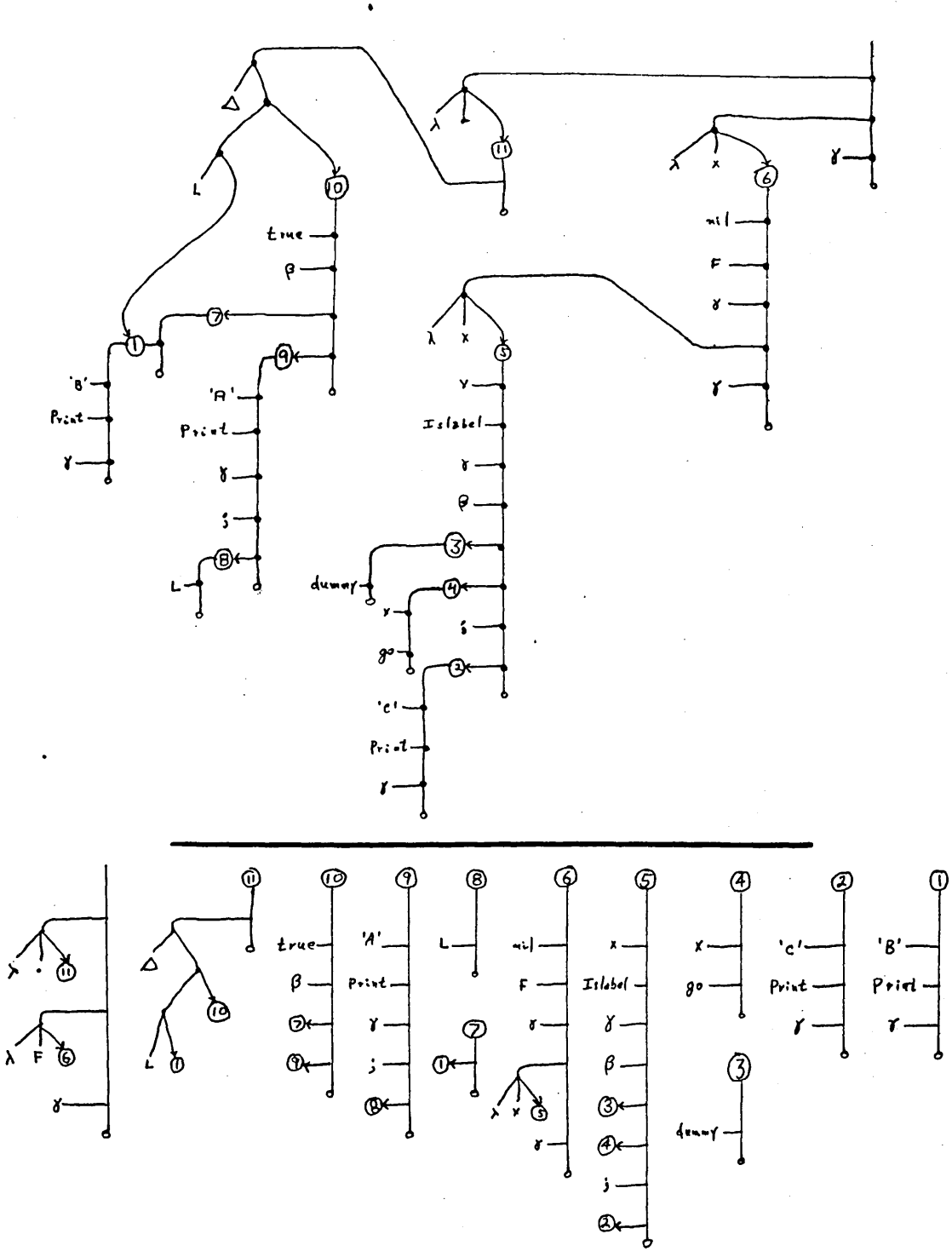


Figure 5.2-4: Control Structure for (5.2-5)

```
let F() =
    true → (Print 'A'; L) | (L: Print 'B')
in
let x = F nil
in
if Islabelx do goto x;
Print 'C'
```

$(\lambda F.\ \downarrow\ )[\lambda().\ true \rightarrow (Print\ 'A'; L) \mid (L: Print\ 'B')]$

$[\lambda x.\ (Islabel\ x \rightarrow (goto\ x) \mid dummy);\ Print\ 'C'][F\ nil]$

$\gamma\ \lambda_6^F\ \lambda_{11}^{nil}$

$\delta_{11} = \Delta_{10}^{'L}$

$\delta_{10} = \delta_9\ \delta_7\ \beta\ true$

$\delta_9 = \delta_8\ ;\ \gamma\ Print\ 'A'$

$\delta_8 = L$

$\delta_7 = \delta_1$

$\delta_1 = \gamma\ \lambda_5^x\ \gamma\ F\ nil$

$\delta_5 = \delta_2\ ;\ \delta_4\ \delta_3\ \beta\ \gamma\ Islabel\ x$

$\delta_4 = goto\ x$

$\delta_3 = dummy$

$\delta_2 = \gamma\ Print\ 'C'$

$\delta_1 = \gamma\ Print\ 'B'$

Figure 5.2-5: Preparation of (5.2-5) for Blackboard Evaluation

	J	Control	Stack	Environment	Memory		P
1		$E_0\ \gamma\ \lambda_6^F\ \lambda_n^{nil}$	E_0	0: PE	0		
2		γ	$^0\lambda_6^F\ ^0\lambda_{11}^{nil}$				
3		$E_1\ \gamma\ \lambda_5^x\ \gamma$ F nil	E_1	1: F σ_1 0	1	1: $^0\lambda_{11}^{nil}$	
4		γ	σ_1 nil				
5		γ	$^0\lambda_{11}^{nil}\ \sigma_2$		2	2: nil	
6		$E_2\ '\Delta_{10}^L$	E_2	2: — 0			
7	7	$E_1\ \gamma\ \lambda_5^x\ E_3\ \delta_{10}$	$E_3\ E_1$	3: L σ_3 2	3	3: δ_1^7	
8		$E_3\ \delta_9\ \delta_7\ \beta$ true	E_3				
9		$\underline{\delta_9}\ \delta_7\ \beta$	true				
10		$E_3\ \delta\ ;\ \gamma$ Print 'A'	E_3				
11		$\delta_8\ ;$	dummy E_3				A
12		$\gamma\ \lambda_5^x E_3\ L$	E_3				
13		$E_1\ \gamma$	$'\lambda_5^x\ \sigma_3\ E_1$				
14		$E_4\ \delta_2\ ;\ \delta_4\ \delta_3\ \beta\ \gamma$ Isl x	E_4	4: x σ_3 1			
15		γ	Isl σ_3				
16		$\beta\ \gamma$	Isl δ_1'				
17		$\underline{\delta_4}\ \delta_3\ \beta$	true				
18		$go\ x$	E_4				
19		go	σ_3				
20		$E_4\ \delta_2\ ;\ go$	$\delta_1^7\ E_4$				
21		$E_1\ \gamma\ \lambda_5^x\ E_3\ \delta_1$	$E_3\ E_1$				
22		$E_3\ \gamma$ Print 'B'	E_3				
23		$\gamma\ \lambda_5^x E_3$	dummy E_3				B
24		$E_1\ \gamma$	$'\lambda_5^x$ dummy E_1				
25		$E_5\ \delta_2\ ;\ \delta_4\ \delta_3\ \beta\ \gamma$ Isl x	E_5	5: x σ_4 1	4	4: dummy	
		γ	Isl σ_4				
		$\beta\ \gamma$	Isl dummy				
		$;\ \delta_4\ \underline{\delta_3}\ \beta$	false E_5				
		$E_5\ \delta_2\ ;$ dummy	E_5				
		$E_5\ \gamma$ Print 'C'	E_5				
		—	dummy				C

Figure 5.2-6: Blackboard Evaluation of (5.2-5)

following is labelled 7 in the "J" column, and this number is placed as a superscript on the value associated with L in memory cell σ_3. A new environment layer, E3, is created for L and connected to the environment (2) which was current when the Δ was encountered. The R-value δ_1^7 associated with L contains two items: The subscript 1 indicates that δ_1 is to be obeyed if L is gone to, and the superscript 7 indicates that information on line 7 of the evaluator is to be used. Specifically, the intent of any future goto with δ_1^7 as a target is to be as follows:

(a) The control and stack are to be put back as they are on that line labelled 7, up to and including the innermost environment markers (E3 in this case), but not beyond them.

(b) The control structure abbreviated by δ_1 is then to be loaded into the control.

This is just what happens on going from line 20 to line 21. Note that no special provision is needed for the environment in the blackboard evaluator, as opposed to the J-machine, because in the former putting environment markers back into the control (and stack) has the desired effect.

Were several labels to be declared at one place, as would be the case for (5.2-4), a control item such as $^{2,3}\Delta_4^{L,M}$ might be used. Obeying this would create two new environment layers, one associating L with a cell containing δ_2^7 (assuming the J-entry to be 7) and the other associating M with a cell containing δ_3^7. Line 7 must be a line in which the environment is such that <u>both</u> L and M are known, so that after a jump to one of these labels execution will be in an environment where both variables are known.

<u>Conditionals</u>: None of the control structures we have seen so far has included a conditional with a label placed on one arm and in which we are concerned with the successor of the conditional. An example of this idea is

S1; test B ifso S2 ifnot (L: S3); S4 (5.2-6)

If S3 is entered normally, by virtue of B having been evaluated as <u>false</u>, its successor is clearly S4. Because of decision (d) on page 5.2-264, it follows that execution of S3 by virtue of a transfer to L must also be followed by S4. The rules we have been using up to now suggest the control structure shown in Figure 5.2-7(a). The effect of a transfer to L will not be such that the successor of S3 is S4. (This may not be obvious at this point, but it is true. There is no way to go "backwards" on the arrows in the figure.) Needed is a control structure which includes with each arm of each conditional the successor to that arm. For (5.2-6) proper control is shown in Figure 5.2-7(b). Clearly a transfer to L would lead to execution of S3 and then of S4.

Let us do a blackboard evaluation that illustrates this idea. Consider

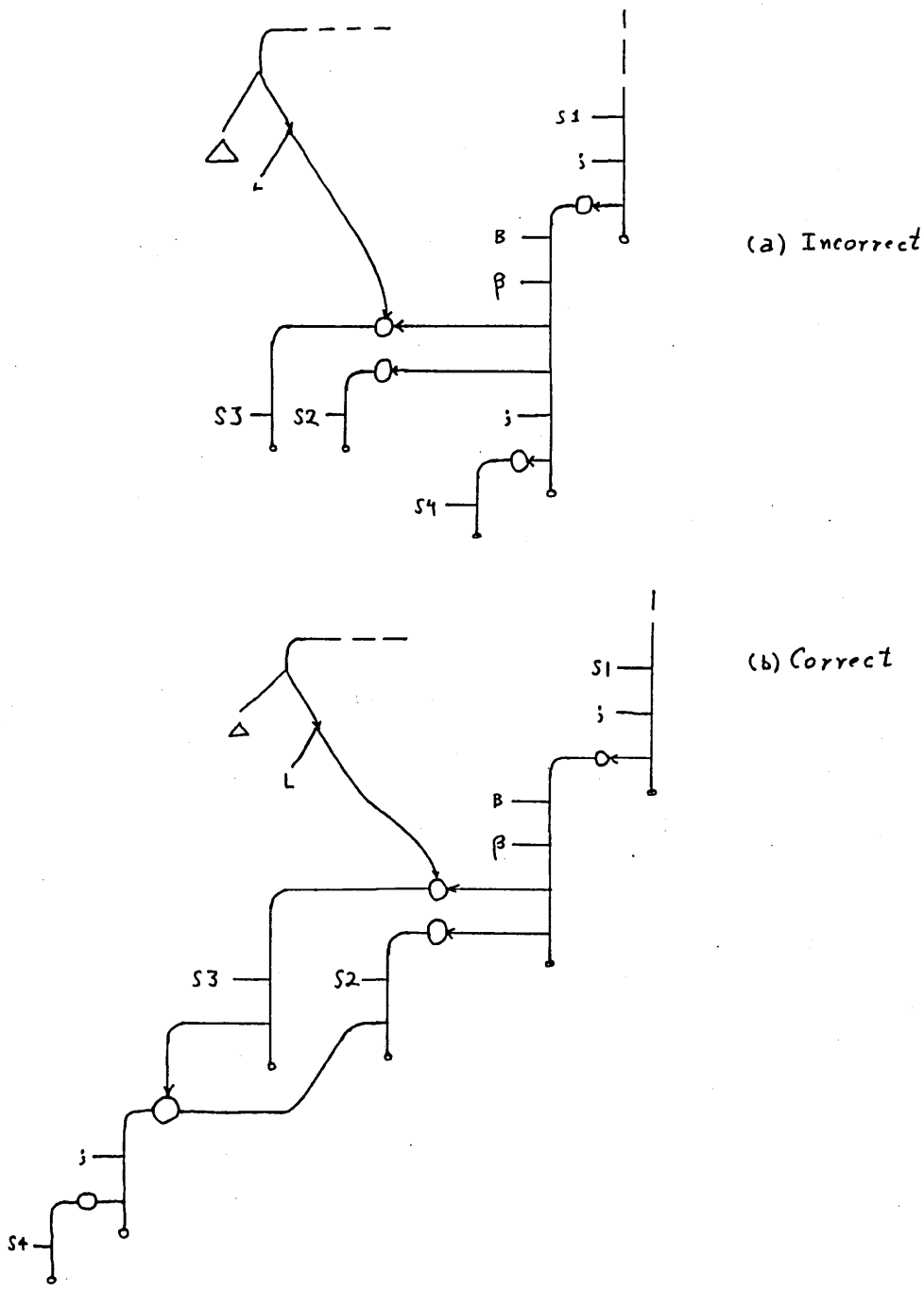


Figure 5.2-7: Control Structure for (5.2-6)

```
let x, y, M = 5, 4, nil
in
x := 1 + (    M := L;
           L: y := y eq 4 -> 6 | 2;                    (5.2-7)
              y + 3
         );
if x eq 10 do goto M;
x
```

In some sense this program is vacuous, since we have claimed that in J-PAL evaluation is for effect only and the program contains no Print statement. Nonetheless blackboard evaluation is illuminating. Note first what is going on: The first time x is updated the value of y is 4, so that we assign 6 to y and then 1 + [y + 3] or 10 to x. In the process the label M has been set equal to the value of the label L, so that obeying later "goto M" leads to evaluation at the entry point corresponding to L, in an environment in which x is 10 and y is 6, and with a state wherein execution of "x := 1 + [...]" has been set up but not yet completed. Evaluation of the bracketed expression this time assigns 2 to y, and returns (y + 3) or 5. Resuming therefore updates x again, this time to 6, and the evaluation ends.

Figure 5.2-8 shows an AE form and blackboard control sequence for (5.2-7), and Figure 5.2-9 shows the evaluation. Figure 5.2-10 shows, for the sake of comparison, gedanken evaluator control structure.

Transform

As in chapters 3 and 4, we have gained intuitive understanding through simulation of the gedanken evaluator using the conventions of the blackboard machine. Now we must complete the formal definitions. In this section we assume that control structure exists, discussing here only Transform. In the next section we finally address the issue of scope of labels in our specification of Translate.

Jump: There are only two new transformation functions in the J-machine for labels: Make_labels, which is called for a Δ node, and Jump, which is called for a goto. We look first at Jump, and then at Make_labels. Hop is also new, but it has already been discussed.

Jump is called when the top control item is GO_TO, and expects the top stack item to be a label. What Jump is to do is to effect a transfer to that label. (This is the action done in the blackboard machine when go is at the top of the control.) What we must now specify is just how the R-value of a label is to be represented.

Look closely at the blackboard evaluations we have done. As mentioned, a label value such as δ_1^6 indicates that δ_1 is to be obeyed in the situation that prevailed on line 6 of the evaluation. The effect of a goto, then, is to install a complete new machine state: all three of C, S and E. Note that the

```
let x, y, M = 5, 4, nil
in
x := 1 + [M := L; L: y := y eq 4 → 6 | 2; y+3];
if x eq 10 do goto M;
x
```

$$[\,\lambda_1(x, y, M).\ \downarrow\,]\ (5, 4, nil)$$

$$x := 1 + \downarrow;_2\ (x\ eq\ 10 \rightarrow_3 (goto\ M)\ |_4\ dummy);_5\ x$$

$$_6M := L;_7\ L:\ y := y\ eq\ 4 \rightarrow_8 6\ |_9\ 2;_{11}\ y+3$$

$$\gamma\ \lambda_1^{xyM}\ \tau_3\ 5\ 4\ nil\ \ \gamma\Delta_6^L$$

$$\delta_1 = \delta_2\ ;:= x + 1$$
$$\delta_2 = \delta_3\ \delta_4\ \beta\ eq\ x\ 10$$
$$\delta_3 = \delta_5\ go\ M$$
$$\delta_4 = \delta_5\ dummy$$
$$\delta_5 = x\ ;$$
$$\delta_6 = \delta_7\ ;:= M\ L$$
$$\delta_7 = \delta_8\ \delta_9\ \beta\ eq\ y\ 4$$
$$\delta_8 = \delta_{10}\ 6$$
$$\delta_9 = \delta_{10}\ 2$$
$$\delta_{10} = \delta_{11}\ ;:= y$$
$$\delta_{11} = +\ y\ 3$$

Figure 5.2-8: Control for Blackboard Evaluation of (5.2-7)

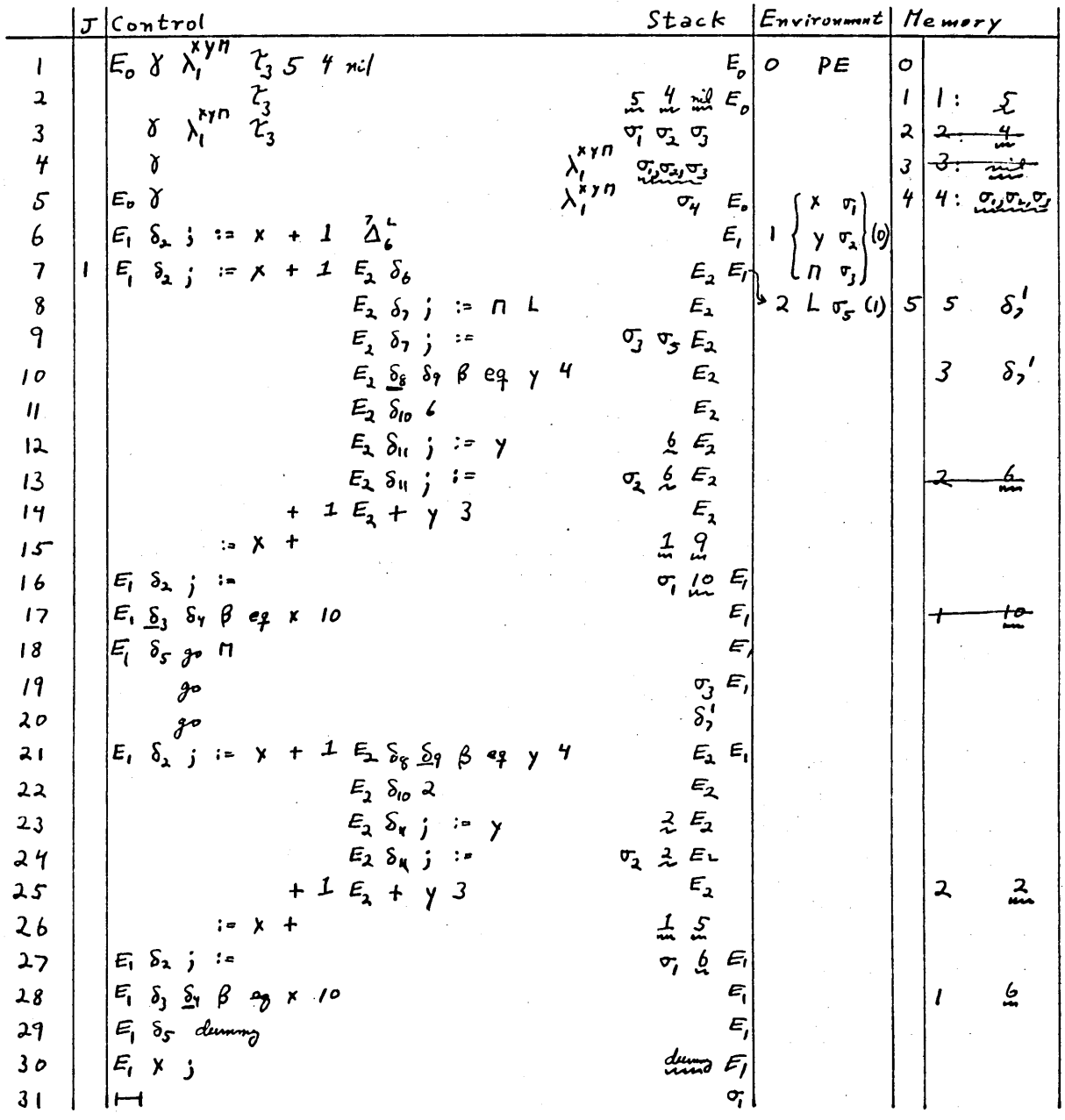


Figure 5.2-9: Blackboard Evaluation of (5.2-7)

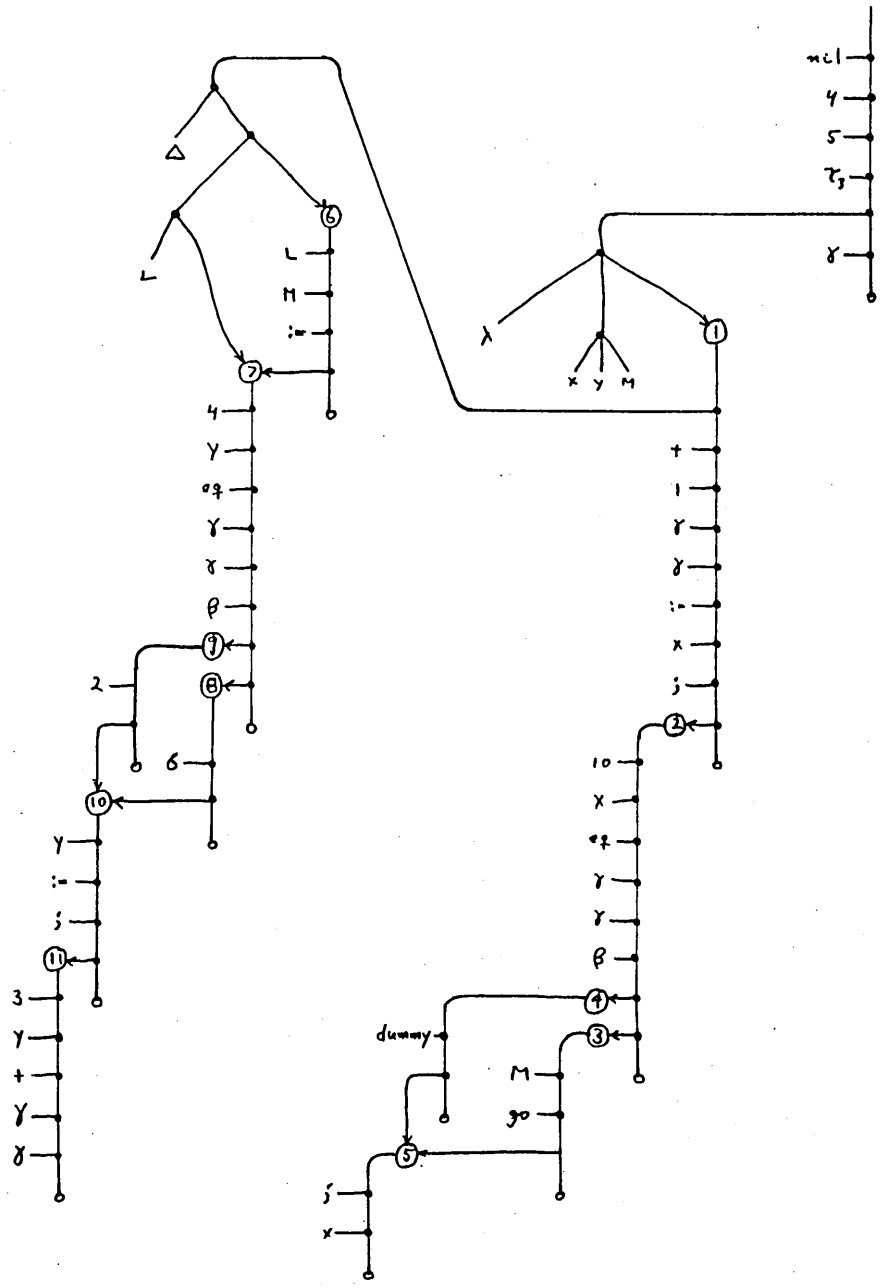


Figure 5.2-10: Control Structure for (5.2-7)

values of C, S and E at the time of the goto do not take part in the next line.

For gedanken evaluation, a different packaging than that suggested by the notation δ_1^6 is more appropriate. We thus decree that a label has three components, corresponding to the C, S and E to be installed as part of the goto. Of course, it also must have some sort of tag indicating that it is a label, so we decree that a label is to look like this:

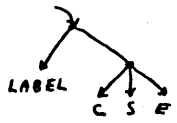


Figure 5.2-11: R-value of a label

Jump then is quite simple:

```
def Jump () =
    unless Is_label (t S) do error;                    (5.2-8)
    C, S, E := t S 2
```

The first line is a necessary check, and the next line is a simultaneous assignment whose right side clearly denotes a 3-tuple. Our task now is to see how such an object is made.

Makelabels: We have already seen that Make_labels is the function called in Transform when Δ is the top control item. We know from our discussion of Jump what it must do, and are now ready to look at the code:

```
def Make_labels () =
    let V = t C 2
    in
    let L, k = V 1, Order(V 1) // Labels
    and New_C = Push(RETURN, r C)
    and New_S = Push($ E, $ S)
    and New_E = $ E
    in                                                 (5.2-9)
    while k > 1 do
      ( let Lab = LABEL, (Push(L k, New C), New_S, New_E)
        in
        New_E := L(k-1), Lval Lab, $ New_E;
        k := k - 2
      );
    C, S, E := Prefix(Contents (V 2), New C), New_S, New_E
```

This program requires some comment. A delta node is a rather complex bundle of information. (Study Figures 5.2-2 or 5.2-4 or 5.2-10.) It is a 2-tuple, whose first component is DELTA (so that one can tell what it is) and whose second component (V in the program) is the useful data. V has two components: A tuple

of labels to be declared, and the address of the control to obey upon completion of the act of declaration. (This piece of control is precisely the scope of the label variables.) Note in the program that L is the tuple of labels (the first component of V) and k is its order. The first, third, ... components of L are each variables, and the second, fourth, ... are each the (address of the) associated control.

Next New_C is calculated, to be used both as part of each label and as part of the control to be done next. We want to obey the control of (V 2) in an environment in which the labels are known, then to remove the labels from the environment, and then to continue with what was going on when the Δ was encountered. The expression

```
Prefix (Contents (V 2), Push (RETURN, r C))          (5.2-10a)
```

has as value the desired control. (Compare with the code in Apply_closure in (5.1-12) on page 5.1-258.) The definition of New_C along with the assignment at the end of the code produce this value.

As in Apply_closure we must save the current environment on the stack, so:

```
New_S = Push ($E, $S)                                (5.2-10b)
```

The new environment is to contain all of the current environment, along with the labels being declared. At first glance it appears that New_E is just an unshared copy of E. However, note the assignment to New_E within the while loop:

```
New_E := L(k-1), Lval Lab, $ New_E                   (5.2-10c)
```

Clearly, executing this statement creates a new environment layer on top of the existing one, a layer in which the label L(k-1) is associated with a memory cell (the one returned by Lval) whose contents is Lab, to be discussed next. The "$" is necessary to prevent a disasterous sharing.

Note now the definition of Lab:

```
Lab = LABEL, (Push (L k, New_C), New_S, New_E)       (5.2-10d)
```

This is precisely the right kind of value for a label: a 2-tuple whose first component is LABEL and whose second component is a (C, S, E) 3-tuple. A subsequent transfer with this label as target will cause the control stored in the address (L k) to be obeyed, followed by resumption of the item after the Δ.

We must still discuss the environment, New_E. It is clear that, after processing the Δ node, we are in an environment in which all of the labels declared are known. It follows from (d) on page 5.2-264 that any transfer to such a label must also result in execution in such an environment. Thus the environment associated with <u>each</u> label must include <u>all</u> of them. Fortunately this is easy: Each label is created by execution of (5.2-10d). (Recall that this definition appears within the iteration on k.) For each such label, the E component <u>shares with</u> New_E. Thus all of the labels share an E component, and it is this component that is updated in (5.2-10c).

The reader is advised to study carefully this description. Although sharing is a very important concept in PAL, this is one of the few instances in the formalization where it is used in such a critical way. The key idea is that we form a (C, S, E) 3-tuple with an E component which we later update to hold the desired information.

Scope of Labels

A problem we have been deferring right along is this: A writing such as "L:" serves to define the variable L as a label. What is the scope of L? That is, in just what part of the program is L to be known? It is this question that is answered so poorly in section 4.3/S of the PAL Manual, and it is time now to answer it. In doing so, we also specify Translate for the J-machine.

Informal Scope Rules: The definition of the value of a label in PAL is predicated on the stipulation that it be meaningful to jump to a label even in circumstances where a hop would be meaningless. Two instances in which hopping was inadequate were cited in (5.1-17) and (5.1-19); in the first case we could not hop into the scope of a local definition, and in the second we could not hop into the middle of a combination. To remedy these deficiencies we have defined the value of a label to comprise all information necessary to jump to it from any place in a program where its value is known. Specifically, we have included in a label value not only an entry point into the label-body, but also the environment and stack that are to be installed when a jump to the label is executed.

The fundamental question to be answered when establishing the scope of a label concerns the point in normal program execution at which the information requisite for constructing the label's value is at hand. We cannot declare a label outside of the smallest enclosing λ-body without prejudicing its environment component, nor can we declare it outside the smallest enclosing functional application without prejudicing its stack component. Thus labels such as L in either of

```
let x = 5 in (L: x := x+1)          (5.2-11a)

x := 5 + (L: 7);                    (5.2-11b)
```

cannot be declared until just before evaluation of the (parenthesized) subexpression in which "L:" appears.

The situation is distinctly different for sequences and for conditionals. Assume the following appears as the entirety of a λ-body:

```
A: S1;
B: test ...
   ifso  (C: S2; D: S3)             (5.2-12)
   ifnot (E: S4; F: S5);
G: S6
```

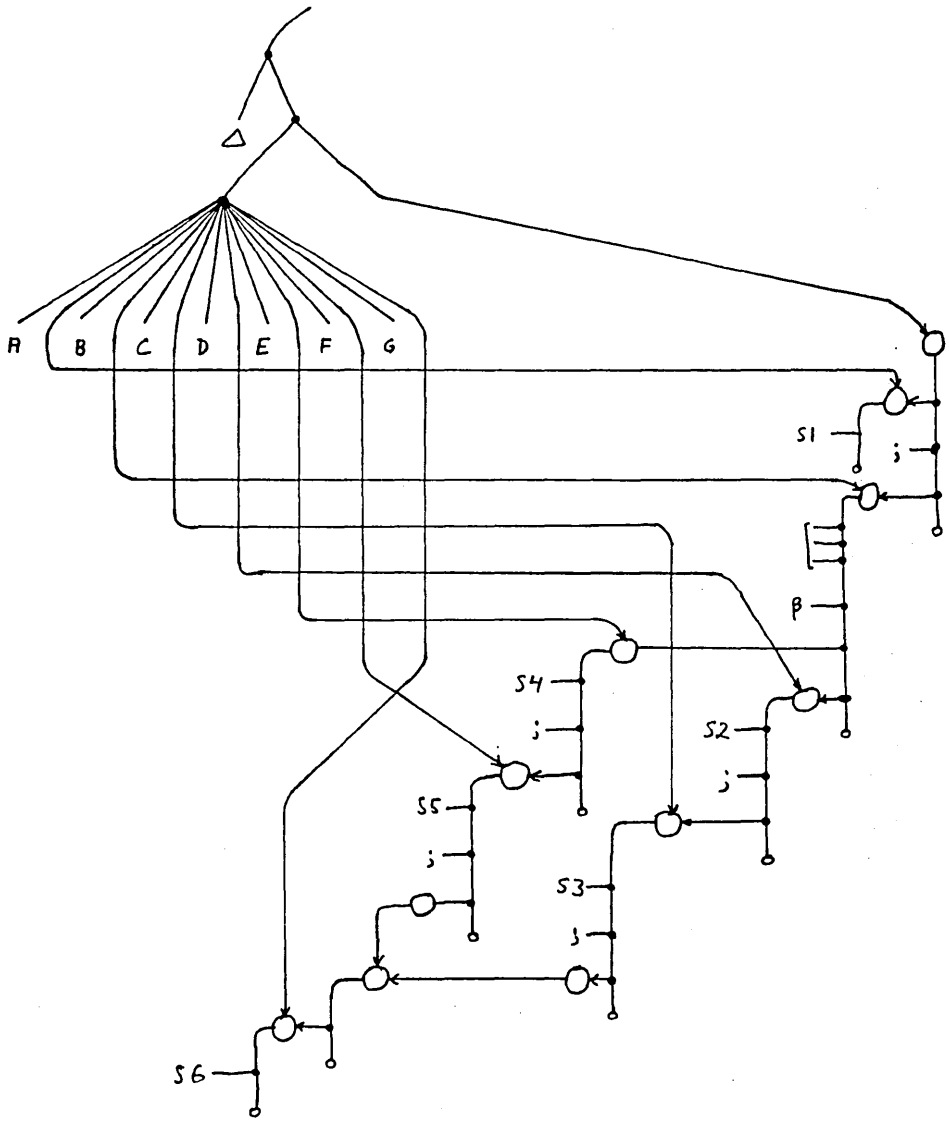


Figure 5.2-12: Control Structure for (5.2-12)

Each of the seven labels shown has as scope the entire body, and each may meaningfuly be transferred to from anywhere in it. Thus it is necessary that all seven of the labels be declared on entry to the λ-body. Figure 5.2-12 shows the control structure for (5.2-12).

The situation when a label is attached to part of a premise of a conditional is different. Consider

test (S1; L: E) ifso S2 ifnot S3 (5.2-13a)

Should L be known outside of the parentheses? We elect to say "no", but the rationale behind the decision is less clear cut. On the other hand, there is no loss of generality due to this decision, since the semantics of (5.2-13a) and of

S1; L: test E ifso S2 ifnot E3 (5.2-13b)

are identical except for the scope of L. This should be obvious, and it can be proved by appeal to Translate: Each of these produces equivalent control structures.

Tree Form of the Scope Rules: What we have been discussing is how far from its position in the text a label is known, a discussion which is awkward because of our concern with the textual representation of a program. This discussion becomes much simpler if formulated in terms of the tree representation. For example, consider (a) of Figure 5.2-13. Our discussion has already indicated that any label whose scope includes all of either A or B is to be known in both of them, a fact that we can express by saying that the label's scope propagates

Figure 5.2-13: Label Propagation

through the semicolon node. We thus draw the arrows shown in (b), indicating that labels known in either son are known outside. We can specify all of PAL's label scope rules this way, and do so in Figure 5.2-14. This picture shows, for example, that labels known in either arm of a conditional propagate through the BETA node, but that labels in the premise do not. We apply the rules to the standardized tree which is the output of ST, thus restricting the number of node types needed. Labels propagate upward only through COLON, ALPHA, BETA and WHILE nodes, and then only as shown. There is no propagation through GAMMA, LAMBDA, AUG, ASSIGN, GO_TO or DOLLAR.

As an example of the use of these scope rules, let us consider the program

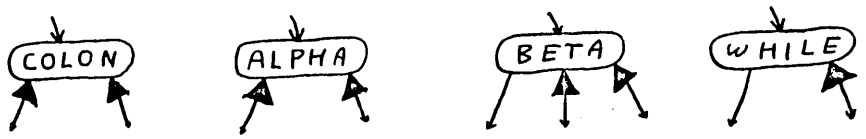


Figure 5.2-14: Propagation Rules

```
let  x = 5
in
7 + [  E1;
       while E2 do L: E3;
       M: E4;                                          (5.2-14)
       test (N: E5; E6)
       ifso E7
       ifnot (E8; P: Q: E9);
       x
    ]
```

The syntax tree standardized tree, output of LL and control structure for this are shown in Figure 5.2-15. The dashed arrows in (b) indicate the subtrees in which each of the labels is known.

One final point needs to be made before we pass on to detailed treatment of Translate. In L-PAL we treated

```
while B do E                                           (5.2-15a)
```

as sugaring for

```
Loop# (λ(). B) (λ(). E)                                (5.2-15b)
```

whereas we have already indicated our intent to do differently in J-PAL. Although the efficiency issues already discussed adequately justify the change, there is one other rather compelling reason for it. We want labels in E in (5.2-15a) to propagate, an effect we achieve by the rule indicated in Figure 5.2-14. But labels in E in (5.2-15b) do <u>not</u> propagate beyond the λ-body of which they are part.

<u>Translate</u>: The Translate functions for R-PAL and L-PAL are identical:

```
def Translate P = FF (ST P, nil)
```

The two step process consists of standardizing the input and then flattening it. For J-PAL we require (or, at least, find useful) an intermediate step which propagates labels in the standardized tree, so that we have

```
def Translate P = FF (LL (ST P), nil)                  (5.2-16)
```

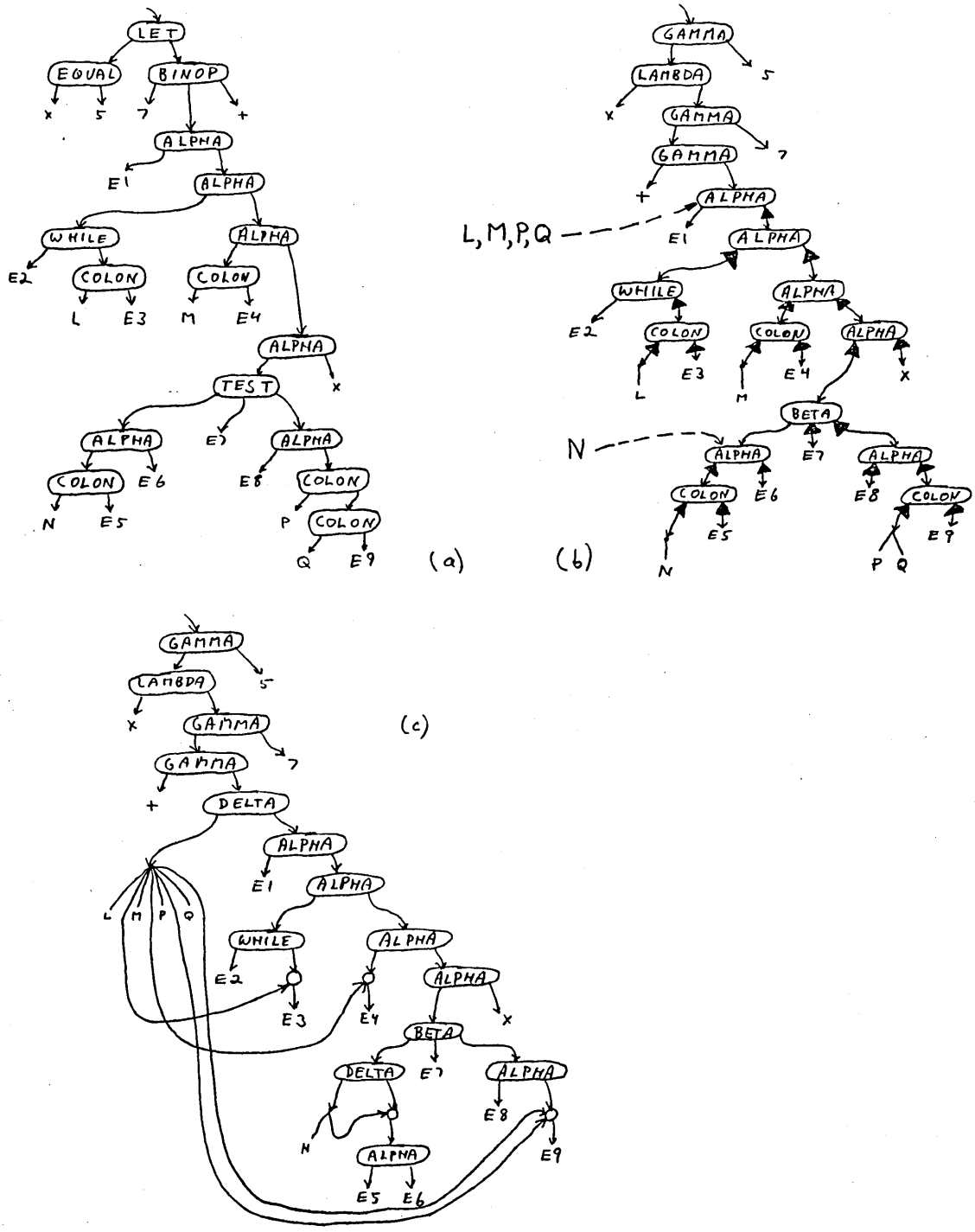


Figure 5.2-15: Trees for (5.2-14)

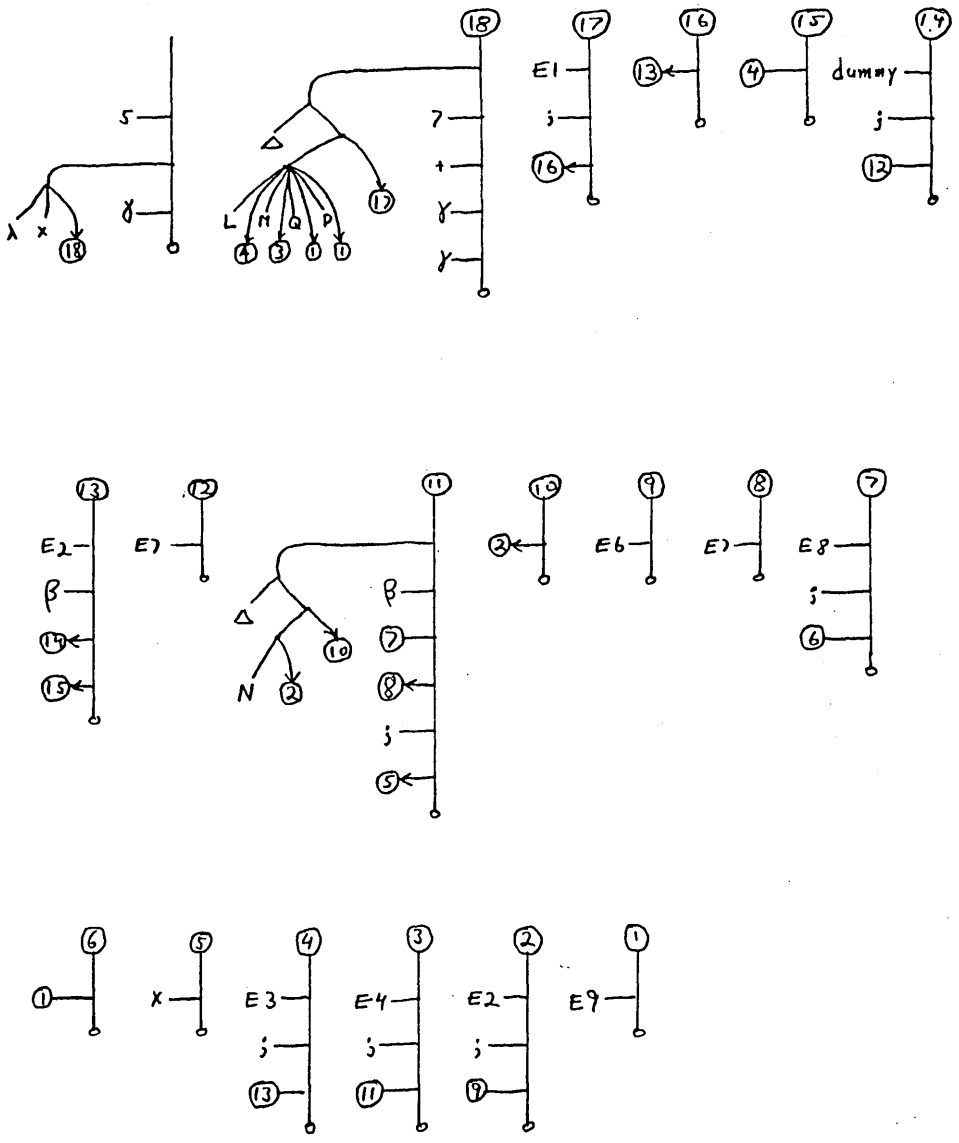
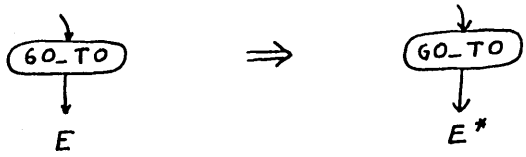


Figure 5.2-15(d): Control Structure for (5.2-14), as Produced by FF.

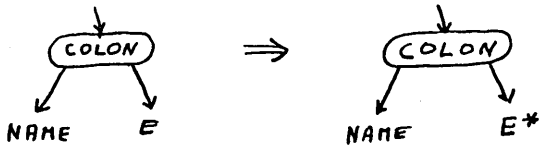
We have already seen what each of these three functions is to do, in several places including Figure 5.2-15, and we must now see how they do it.

The Function ST: For the most part, the J-PAL standardizing function ST operates as does its R-PAL and L-PAL counterparts. The new node types in its input are COLON, GO_TO, VALOF and RES, and the two node types in the output are COLON and GO_TO. VALOF and RES will be seen to be merely sugarings for other constructs.

For GO_TO the processing is simple:



COLON is slightly trickier. For the most part, what we want is something like

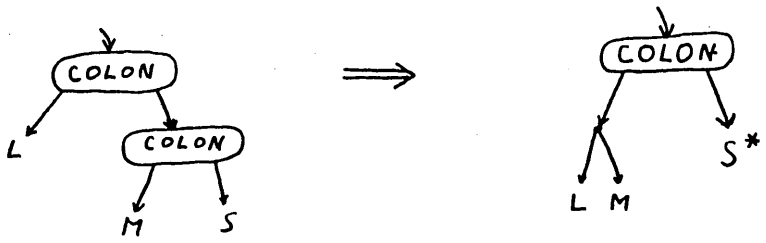


(5.2-17a)

However, the job of LL is eased if multiple labels on a single place are checked for by ST. Recall that PAL's syntax permits multiple labels, as in

...; L: M: S; ... (5.2-17b)

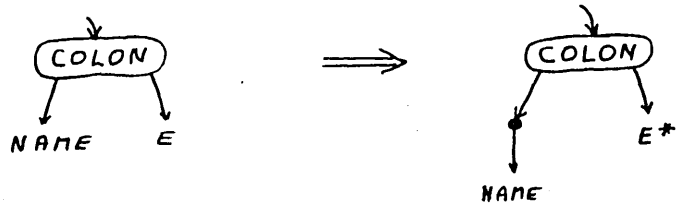
(We saw in (5.2-4) another example of this.) The tree form of (5.2-17b) and the standardized tree are



(5.2-17c)

It eases subsequent processing if in the output of ST the left son of COLON nodes is always a tuple. Thus in the case in which there is only one label, we decree that the output COLON node is to have a 1-tuple as its left son, so

(5.2-17a) should be replaced by



(5.2-17d)

The code in ST that does the work is

```
let w = ST (x 2)
in
Is_tag w COLON                                    (5.2-18)
-> COLON_ (w 1 aug x 1) (w 2)
|  COLON_ (nil aug x 1) w
```

The effect is that the right son of the COLON is standardized and then checked to see if it too is a COLON node. If not, the output COLON node has as its left son a 1-tuple which is the label. Otherwise the two nodes are coalesced into one.

It is necessary that this code work for more than two labels at the same place. It is worth looking closely at it to see why it does work, since the technique used here in a rather simple way is used in a critical way in LL. Note that to standardize a COLON node we first standardize its right son and then ask if that is also a COLON. If it is, we agglomerate the labels together. The point that needs careful study is just how the recursive calls to ST are organized.

The Function LL: LL is applied to a standardized tree possibly containing COLON nodes and returns a tree containing appropriate DELTA nodes. Each DELTA node has two sons: a (2 k)-tuple and a tree, where the tuple contains alternately labels and pointers into the tree. We call such a tuple a label-list. Consider again (5.2-14) and its standardized tree and output of LL as shown in (b) and (c) of Figure 5.2-15. Note that each label has been propagated as far up the tree as possible, as indicated by the upward arrows in (b). (These arrows are drawn as dictated by the rules of Figure 5.2-14.) Note further that each labeled tree is stored in a memory cell, as indicated by the small circles in (c). The reason is this: The DELTA node provides an association between a label variable and a piece of the tree, and it is necessary that this association be preserved in FF. To achieve this, we elect to associate a label with an address, and store in that address the tree that is labeled. FF will then flatten that piece of tree and store the flattened tree back into the same cell. This point was aluded to earlier on page 5.2-269 in the discussion of Figure 5.2-3.

The code for LL is shown in Figure 5.2-16, and it is not very transparent code. The validity of the label propagating scheme turns on the fact that for

```
//  The function LL processes labels, bringing each label as far
//  up the tree as possible.  The effect is that each label is
//  declared by a DELTA node as soon as its scope is entered.

def       Combine(x, y) = Q 1 x
               where rec Q k s =
                         k > Order y -> s | Q (k+1) (s aug y k)
          within
          Proc_labels x =
               Is_tag x DELTA
               ->  (x 1, x 2)
               |   (nil, x)
          within
          Combine_labels(u, v)  =
               let U = Proc_labels u
               and V = Proc_labels v
               in
               Combine(U 1, V 1), (U 2, V 2)
          within

rec LL x =


     let Type = Is_tag x
     in
        Is_identifier x  ->  x
     |  Type ALPHA
          -> ( let s, w = Combine_labels( LL(x 1), LL(x 2) )
               in
               DELTA_ s ( ALPHA_ (w 1) (w 2) )
             )
     |  Type BETA
          -> ( let s, w = Combine_labels( LL(x 2), LL(x 3) )
               in
               DELTA_ s ( BETA_ (LL(x 1)) (w 1) (w 2) )
             )
     |  Type WHILE
          -> ( let s, w = Proc_labels( LL(x 2) )
               in
               DELTA_ s (WHILE_ (LL(x 1)) w)
             )
     |  Type COLON
          -> ( let L, z = Proc_labels(LL(x 2))
               in
               let w = Lval z
               in
               DELTA_ (Q 1 L) w
               where rec Q k t =
                         k > Order(x 1) -> t | Q (k+1) (t aug x 1 k aug w)
             )
     |  Type LAMBDA  ->  LAMBDA_ (x 1) (LL(x 2))
     |  Sons x eq 1  ->  Tag (Get_tag x) ( nil aug LL(x 1) )
     |  Sons x eq 2  ->  Tag (Get_tag x) ( LL(x 1), LL(x 2) )
     |  error
```

Figure 5.2-16: The Function "LL" that Propagates Labels

any standardized subtree x, LL(x) will be a subtree in which all labels that are candidates for further propagation are agglomerated into a single Δ node at the root. Thus LL(x) will have the form illustrated in (a) of Figure 5.2-17 if candidate labels exist, and the form illustrated in (b) otherwise. It follows then that we can determine whether or not to propagate labels through any node by testing its type and the type of the result of applying LL to its sons.

The effect of label agglomeration is illustrated in (c) of Figure 5.2-17 in which we have a node x of type Δ. Applying LL to its second son will by hypothesis return a Δ-node, called v in the picture, with candidate labels already agglomerated as shown. To continue the agglomeration we wish to produce a single Δ-node whose left son contains both sets of label-entry point pairs. This is accomplished with the help of the three functions Combine, Proc_labels and Combine_labels, as shown in Figure 5.2-16. (PAL's syntax of within definitions is such that each of these is known within the following ones, and all three of them are known within LL.) One other rather interesting programming artifice is used to simplify the code:

```
def DELTA_ x y =                                        (5.2-19)
        Null x -> y | Tag DELTA (x, y)
```

This variation on the usual tagging function makes the code for LL simpler, since it permits applying DELTA_ to a label-list and a tree without first checking that the list is non-empty. (If there are no labels, the label-list is nil.)

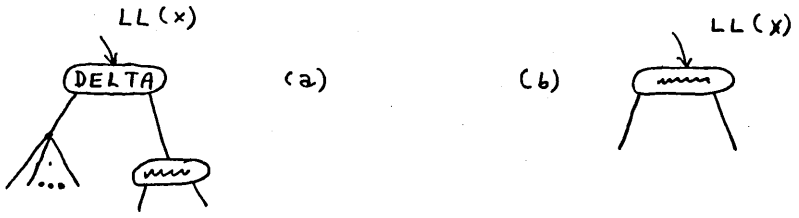
Let us look at the three auxiliary functions. The value of

```
Combine (x, y)
```

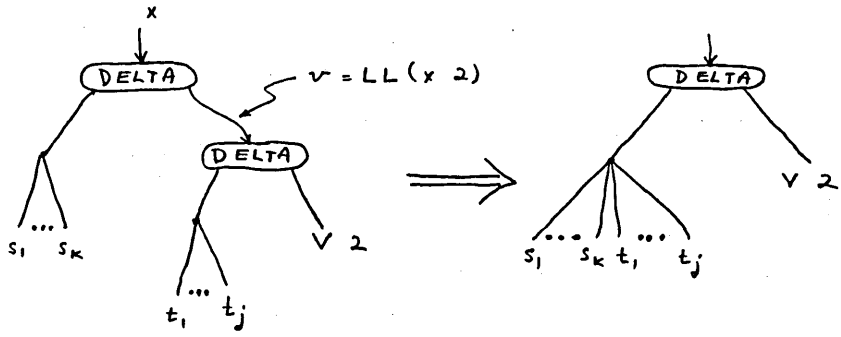
where x and y are each tuples is a single tuple containing the components of x followed by those of y. It is used to combine two label-lists, and works properly if either or both is empty. Proc_labels is applied to a node and returns a 2-tuple whose first component is the label-list for the labels declared at that node and whose second component is the tree at that node. Combine_labels is applied to two nodes and returns a 2-tuple whose first component is the label-list and whose second component is a 2-tuple whose components are the two trees.

Now examine again the code for LL. For each of ALPHA, BETA and WHILE, we calculate in s the label-list and in w the tree(s), and then build the appropriate DELTA node. The trick in the definition of DELTA_ in (5.2-19) saves the necessity of checking for an empty label-list in each of these. Note that in the BETA and WHILE cases LL is applied to the booleans, but any labels found are not propagated.

Now note the code for COLON. Proc_labels is called to store into L the label-list for the right son of the colon node and into z the tree for it. We then call Lval (see (5.1-6)) to get a new memory cell, a cell that contains z. We then call the recursive function Q to build a new label-list by "aug"ing onto L (which will be nil if no labels propagated to the top of the right son of the



Form of subtrees produced by LL.
Here ⌒ is any node type other than DELTA.



(c) Agglomeration of labels by LL.

Figure 5.2-17: Operation of the Function "LL"

original COLON node x) the label(s) being declared at this point. Note that each label thus declared is associated in the label-list with the same cell: w.

The Function FF: The J-PAL version of FF differs from the R-PAL or L-PAL versions in one important way: It stores pieces of control structure into memory in a way similar to the abbreviation scheme we have been using in blackboard evaluation. For example, we have always used a δ to abbreviate λ-bodies, so the corresponding code is

```
Type LAMBDA
     -> ( let Body = Lval (FF (x 2, nil))
          in
          Cons_lambda_exp (x 1, Body), c
        )
```

It differs from the L-PAL version only through the presence of the call to Lval.

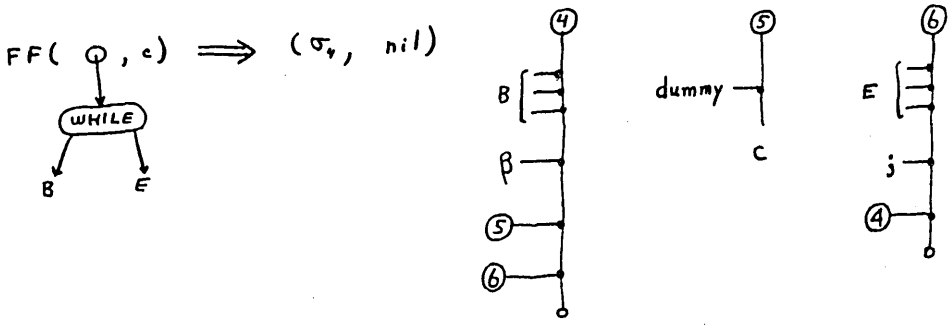
For the case where LL produces a piece of tree in a memory cell, FF will encounter the address of that cell in its input. The relevant code is

```
Is_address x  ->
     [ Update(x, FF(Contents x, c));  (x, nil) ]
```

FF does have one new problem: constructing the loop needed in the control for while. We have

```
Type WHILE ->
  (  let w = Lval NIL
     in
     let TA = Lval(FF(x 2, (ALPHA, (w, nil))))
     and FA = Lval(DUMMY, c)
     in
     Update (w, FF(x 1, (BETA, (FA, (TA, nil)))));
     (w, nil)
  )
```

The key fact here is that the address w is used in constructing TA, and then the contents of cell w is updated to hold a reference to TA. Suppose that the three calls to Lval return σ_4, σ_5 and σ_6. Then the effect of this processing is:



Note the similarity between this drawing and that in Figure 5.1-2.

The Constructs "valof" and "res"

Although jumps afford facilities for resuming any CSE state, they do not of themselves provide for carrying back a value calculated at the place from which the jump is made. This latter capability is provided in PAL by the valof and res constructs.

The easiest way to specify the semantics of valof and res is to view them as syntactic sugar for other expressions whose effects have already been defined. For preliminary purposes, we may say that for any expression E1

```
valof E1                                    (5.2-20a)
```

is equivalent to

```
let π = nil
in
π := E1;                                    (5.2-20b)
ρ: π
```

and that for any expression E2

```
res E2                                      (5.2-21a)
```

is equivalent to

```
π := E2; goto ρ                             (5.2-21b)
```

The intent is that (res E2) appear as a subexpression of E1, so that the name π and the label ρ of (5.2-21b) refer to the entities defined in (5.2-20b). The same names π and ρ are used in all instances of valof and res that are encountered. PAL's normal scope rules serve to associate each res with the proper valof.

Analysis of the effect of writing

```
valof [~~(res 5)~~]
```

is straightforward. If res is not encountered during evaluation of the bracketed expression, the effect of the valof is nugatory: We merely waste effort by setting up the dummy variable π, assigning the value of the bracketed expression to it, and then returning this via the evaluation of π. On the other hand, if res is encountered during the evaluation, then we assign 5 to π. All further evaluation is terminated by jumping immediately to the label ρ, thereby returning the value of π which in this case is 5. It should be clear from (5.2-20) and (5.2-21) that in the case of nesting, res will cause a return to the first valof above it in the syntax tree representation of a program.

These constructs can be useful in providing an error exit from deep within the evaluation of a recursive function. Consider for example the program

```
let f n = valof
  (  g n
     where rec g k =
             k eq 0.0  ->  1.0
           | k * (k < 0.0 -> (res k) | g(k-1.0))
  )
in
f(3), f(2.6), f(-2.3)
```
(5.2-22)

Here f is analogous to the factorial function, except defined on real numbers rather than on integers. If n is a whole non-negative number, g returns n factorial; if n is negative, g returns n; and if n is a fractional positive number, g returns the non-integral part of n, minus 1.0. Thus the value of (5.2-22) is the 3-tuple (6.0, -0.4, -2.3).

There may be several instances of res within a single valof. Thus

valof (~~ res 5 ~~ res 6 ~~ res 7 ~~)

returns 5, 6 or 7, depending on which res is first encountered during execution.

Since it is impossible to prevent the writing of obscure programs in any non-trivial programming language, the proper objective in language design would seem to be provision of facilities that are at least adequate for writing perspicuous ones. In this regard valof and res appear meritorious -- certainly it is desirable to avoid the use of explicit jumps and labels whenever possible. With the inclusion of every new linguistic facility, however, come new opportunities to lapse into obscurity. Consider

```
let f = valof (fn t. res t)
in
Isinteger f  ->  f - 3  |  f 4
```
(5.2-23)

Verification that the value of (5.2-23) is 1 is left as an exercise for the reader. It is an easy blackboard evaluation and an instructive one.

Our sole remaining task is to remedy a small defect in our preliminary definitions of valof and res. The assignment command "π := E2" in (5.2-20b) evaluates E2 in R-mode, and stores the resulting value in memory address π. A more general facility can be provided by redefining valof and res in such a way that the result of "valof E1" is the L-value of E2. Accordingly, we adopt the equivalences

```
valof E1  <=>  let π = nil
               in
               π := nil aug (E1);
               ρ: π 1
```

and

```
res E2  <=>  π := nil aug E2; goto ρ
```

as the actual definitions of valof and res in PAL. There is a rather pleasant return from the effort taken for this more complicated definition. It seems reasonable that, for any expression E not containing res, the expressions "valof(res E)" and "E" be identical in their semantics. But "valof(res x)" does not share with x if the first definition is used, while it does if we use the second.

All of the formal definition of valof and res appears in the function ST, where they are desugared as indicated. See the definition of ST at the end of this chapter.

5.3 Listings of the J-PAL Evaluator

The following pages contain a complete listing of the gedanken evaluator for J-PAL, as it has actually run in a PAL implementation (on Multics). All necessary representational issues are faced up to. As in the R-PAL listings in Chapter 3, the only variable appearing here that is not defined (other than those in PAL's primitive environment) is Error.

Any discrepencies found between the programs shown here and those shown earlier in the chapter should be resolved in favor of those shown here.

```
//                    PRELIMINARY DEFINITIONS
//  Preliminary definitions for the evaluator.

//  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *

//  Selectors and constructors for the stack and control.
def t(x, y) = x  // Top of stack or control.
and r(x, y) = y  // Rest of stack or control.
and Push(x, s) = x, s  // Put new item on stack or control.
def rec Prefix(x, y) =  // Put control x at top of control y.
        Null x -> y
      | Push(t x, Prefix(r x, y) )

def r2 x = r(r x)  // Rest of (rest of (stack or control)).
and r3 x = r(r(r x))  // Rest of (rest of rest).
and 2d x = t(r x)  // Second element of stack or control.
and 3d x = t(r(r x))  // Third...

def Empty_stack = nil  // The empty stack.

//  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *

//  Tagger and tag-checkers for structures.
def Tag n s  =  s aug n  //  Tag structure s with tag n.
and Is_tag s n  =  //  Does structure s have tag n?
     Istuple s  ->  n eq s(Order s)  |  false
and Get_tag s  =  s(Order s)  //  Return the tag of s.
and Sons s  =  Order s - 1  //  Return number of sons of s.
```

```
//  Selectors, predicates and constructors for lambda-expressions
//  and lambda-closures.

def LAMBDA = '_lambda'  // Tag for lambda-expressions and closures.

def bV x = x 2      //  Select bv-part of a lambda-exp or closure.
and Body x = x 3    //  Select body part...
and Env x = x 4     //  Select environment part...

def       Test(x, n) =
               |stuple x
               -> Order x eq n
                  -> |sstring(x 1)
                     -> x 1 eq LAMBDA
                     |  false
                  |  false
               |  false
          within

     |s_lambda_exp x = Test(x, 3)
and  |s_closure x = Test(x, 4)

def Cons_lambda_exp(bV, Body)  =  //  Construct a lambda-expression.
     LAMBDA, bV, Body

and Cons_closure(L_exp, Env)   =  //  Construct a lambda-closure.
     LAMBDA, bV L_exp, Body L_exp, Env
```

```
//  Definitions and predicates for the jumping evaluator.



//  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *

//  Items and predicates for control structure and stack.

def GAMMA       = '_gamma'
and BETA        = '_beta'
and DELTA       = '_delta'
and CONSTANT    = '_constant'
and VARIABLE    = '_variable'
and ADDRESS     = '_address'    // Used only in stack.
and ASSIGN      = '_assign'     // :=
and GO_TO       = '_goto'
and DOLLAR      = '_dollar'
and AUG         = '_aug'
and TUPLE       = '_tuple'      //  Used only in the stack.
and ALPHA       = '_alpha'
and LABEL       = '_label'      // Used only in stack.
and RETURN      = '_return'

and BASIC       = '_basic'      // Tag built-in functions, as Print_.

def       Test(x, y) =
             Istuple x
             -> Order x eq 2
                -> Isstring(x 1)
                   -> x 1 eq y
                   |  false
                |  false
             |  false
     within
     Is_constant x  = Test(x, CONSTANT) or Test(x, BASIC)
 and Is_variable x  = Test(x, VARIABLE)
 and Is_address x   = Test(x, ADDRESS)
 and Is_label x     = Test(x, LABEL)
 and Is_delta x     = Test(x, DELTA)

 and Is_basic x     = Test(x, BASIC)

 and Is_tuple x =
        Test(x, TUPLE)  ->  true  // Is it a constructed tuple?
     |  Test(x, CONSTANT)  ->  Null(x 2)  // Is it nil?
     |  false  // Neither.

 and Is_identifier x =  // Is x constant or variable ?
     Test(x, CONSTANT) or Test(x, VARIABLE) or Test(x, BASIC)

def Same_var (x, y) =  // Are x and y the same variable?
     (x 2) eq (y 2)
```

```
//                  Variables and Constants

//  Call for Y_VAR is produced in Translate for rec-defs.
//  PI, RHO, 1_ and NIL are used for "valof" and "res".

def Y_NAME  = 'yy' // The name of "Y".

def Y_VAR =
     VARIABLE, Y_NAME

and Assign_VAR =  // Routine for simultaneous assignment.
     VARIABLE, 'Assign#'

and PI   =  // Used in desugaring 'valof' and 'res'.
     VARIABLE, 'pi'

and RHO  =  // Used in desugaring 'valof' and 'res'.
     VARIABLE, 'rho'

and PRINT =  // Print routine for user.
     VARIABLE, 'Print_'

and 1_   =  // The constant '1'.
     CONSTANT, 1

and NIL  =
     CONSTANT, nil

and DUMMY  =
     CONSTANT, '_dummy'
```

```
//                      Tags and Taggers


//  Tags for abstract syntax tree.

def TEST      = '_test'       // test ... ifso ... ifnot ...
and ARROW     = '_arrow'      // ... -> ... | ...
and IF        = '_if'         // if ... do ...
and AP        = '_ap'         // functional application
and FN        = '_fn'         // lambda
and EQUAL     = '_equal'      // definition
and WITHIN    = '_within'
and REC       = '_rec'
and FF        = '_ff'         // function form definition
and AND       = '_and'        // 'and' definition
and COMMA     = '_comma'      // tuple maker
and LET       = '_let'
and WHERE     = '_where'
and COLON     = '_colon'
and VALOF     = '_valof'
and RES       = '_res'
and WHILE     = '_while'
and BINOP     = '_binop'
and UNOP      = '_unop'
and PERCENT   = '_percent'


//  Taggers for tags in abstract syntax tree.

def TEST_ x y z      = Tag TEST (x, y, z)
and ARROW_ x y z     = Tag ARROW (x, y, z)
and IF_ x y          = Tag IF (x, y)
and AP_ x y          = Tag AP (x, y)
and FN_ x y          = Tag FN (x, y)
and LET_ x y         = Tag LET (x, y)
and WHERE_ x y       = Tag WHERE (x, y)
and EQUAL_ x y       = Tag EQUAL (x, y)
and WITHIN_ x y      = Tag WITHIN (x, y)
and REC_ x           = Tag REC (nil aug x)
and FF_ x y          = Tag FF (x, y)
and AUG_ x y         = Tag AUG (x, y)
and ASSIGN_ x y      = Tag ASSIGN (x, y)
and ALPHA_ x y       = Tag ALPHA (x, y)
and DOLLAR_ x        = Tag DOLLAR (nil aug x)
and GOTO_ x          = Tag GO_TO (nil aug x)
and COLON_ x y       = Tag COLON (x, y)
and VALOF_ x         = Tag VALOF (nil aug x)
and RES_ x           = Tag RES (nil aug x)
and WHILE_ x y       = Tag WHILE (x, y)
and BINOP_ x y z     = Tag BINOP (x, y, z)
and UNOP_ x y        = Tag UNOP (x, y)
and PERCENT_ x y z   = Tag PERCENT (x, y, z)

//  AND_ and COMMA_ would have to be n-ary taggers, and hence
//  are not provided.
```

```
//  Taggers for standardized syntax tree.

def GAMMA_ x y       = Tag GAMMA (x, y)
and BETA_ x y z      = Tag BETA (x, y, z)
and LAMBDA_ x y      = Tag LAMBDA (x, y)

and DELTA_ x y       = Null x -> y | Tag DELTA (x, y)


//  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *

//  Some useful functions for transform.

def Value_of x  =  // Evaluate a control element, to put it on stack.
     x

and Val_of x  =  // De-tag a stack element, to get its value.
     x 2

def Apply x y  =
     let t = (Val_of x) (Is_basic x -> y | Val_of y)
     in
     Is_address t -> t | (CONSTANT, t)

and Aug x y  =  //  Augment x with y.
     Is_tuple x  ->  (TUPLE, Val_of x aug y)
     |  Error 'first argument of aug not a tuple'


//  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *

//  Define the five components of the evaluator.  E and M are used
//  as global variables in the following functions.

def C, S, E, M = nil, nil, nil, nil
```

```
//                          M E M O R Y

//  A memory is a 2-tuple, whose first component is that integer
//  which is the last address used (initially zero), and whose
//  second component is a Mem, like this:

//        A Mem is either empty (nil)
//        or it is a 3-tuple, whose components are
//           an address,
//           a contents,
//           a Mem.

def
     Extend Value  =  // Find a new cell to hold Value.
          let k = 1 + M 1  // Address of next free cell.
          in
          M := k, (k, Value, M 2);  // Create new memory.
          (ADDRESS, k)  // Return the new address.

 and
     Update(Cell, Value)  =
          M := M 1, (Cell 2, Value, M 2)

 and
     Contents Cell  =
          let c = Cell 2
          in
          Look (M 2)
          where rec Look m =
                    Null m  ->  Error 'address not in memory'
                |   m 1 eq c  ->  m 2
                |   Look (m 3)

 and
     Initialize_memory ()  =
          M := 0, nil


//  Two useful functions used by the evaluator.

//  Return argument if not an address, and contents otherwise.

def Rval x =
     Is_address x  ->  Contents x  |  x

//  Return argument if an address, and new cell containing it
//  otherwise.

and Lval x =
     Is_address x  ->  x  |  Extend x
```

```
//                  Print_ -- User-callable Print routine.

//  If the user includes in his program application of the variable
//  PRINT, this routine will be applied to its argument.  It will
//  print the argument on lines starting with '>> ', and will do
//  tuples by indenting.

def Max_D = 4  // Maximum depth of tuples to print.

def Print_ x =
     let rec F(T, s, d) = // Print T with indent s at depth d.
          let V = Val_of T  // The value.
          in
          test Is_tuple T  // Are we printing a tuple?
          ifnot Write(s, T, '*n')  // No, so print it.
          ifso  // We are printing a tuple.
          test Null V  // Is it the 0-tuple?
          ifso Write(s, T, '*n')  // Yes, so print it.
          ifnot  // It's a long tuple.
          test d ge Max_D  // Is depth too great?
          ifso Write(s, TUPLE, '*s', Order V, 'etc*n')
          ifnot  // Now we can print the long tuple.
            (  let k, N = 1, Order V
               and S = Conc(s, '*s*s*s')  // The new indent.
               and D = d + 1  // The new depth.
               in
               while k le N do  // Iterate through the thing.
                 (  F( Contents(V k), S, D ); k := k + 1  )
            )
     in

     Print '*n';
     F( x, '>> ', 0 );
     Print '*n'
```

```
//                    E N V I R O N M E N T

//  An environment is either empty (nil), or a 3-tuple:
//        Name, Value, Environment

//  The primitive environment:

def Initialize_env () =
     E := Y_VAR, Extend Y_VAR,
          (PRINT, Extend(BASIC, Print_),
           nil
          )


//  The function to look up a variable in the environment:

def Lookup Var =
     L E  // Start looking in the environment.
     where rec L e =
             Null e  ->  Error 'variable not found in environment'
           | Same_var(Var, e 1) -> e 2  // Found.
           | L(e 3)  // Keep looking.


//  The following function is used in applying a lambda0closure.
//  The names on the (possibly structured) bv-part 'Names' are
//  added to the environment 'Env', associated with the corres-
//  ponding part of 'Values'.  The new environment is returned as
//  the value of the function.

def rec Decompose(Names, Values, Env)  =
     test  Is_variable Names  // Is it a single variable?
     ifso  (Names, Values, Env)  // Yes, so add it to environment.
     ifnot
       (  let V = Contents(Values)
          in
          test  Is_tuple V
          ifnot Error 'conformality failure'  // Tuple applied to scalar.
          ifso
          test  Order Names eq Order (Val_of V)
          ifnot Error 'conformality failure.'  // Differing tuple lengths.
          ifso  // Process a multiple-bv part.
            (  Q 1 Env
               where rec Q n e =
                     n > Order Names  ->  e
                     |  Q (n+1) ( Decompose(Names n, Val_of V n, e) )
            )
       )
```

```
def rec D x =  // Standardize a definition.
     let Type = Is_tag x
     in
        Type EQUAL  ->  x  // Already OK.
     |  Type WITHIN
          -> ( let u, v = D(x 1), D(x 2)
                 in
                 EQUAL_ (v 1) ( AP_ (FN_ (u 1) (v 2)) (u 2) )
             )
     |  Type REC
          -> ( let w = D(x 1)
                 in
                 EQUAL_ (w 1) ( AP_ Y_VAR (FN_ (w 1) (w 2)) )
             )
     |  Type FF
          -> ( EQUAL_ (x 1 1) (Q (Order(x 1)) (x 2))
                 where rec Q k t =
                       k < 2  ->  t
                       |  Q (k-1) (FN_ (x 1 k) t)
             )
     |  Type AND
          -> ( EQUAL_ L (Tag COMMA R)
                 where rec L, R = Q 1 nil nil
                 where rec Q k s t =
                       k > Sons x  ->  (s, t)
                       |  ( let w = D(x k)
                              in
                              Q (k+1) (s aug w 1) (t aug w 2)
                          )
             )
     |  Error 'improper node found in D'

def rec ST x  =  // Standardize abstract syntax tree.
     let Type = Is_tag x
     in
        Is_identifier x  ->  x
     |  Type TEST or Type ARROW
          ->  BETA_ (ST(x 1)) (ST(x 2)) (ST(x 3))
     |  Type IF
          ->  BETA_ (ST(x 1)) (ST(x 2)) DUMMY
     |  Type FN
          ->  LAMBDA_ (x 1) (ST(x 2))
     |  Type COMMA
          -> ( Q 1 NIL
                 where rec Q k t =
                       k > Sons x  ->  t
                     |  Q (k+1) ( AUG_ t (ST(x k)) )
             )
     |  Type PERCENT
          -> GAMMA_ (x 2) ( AUG_ (AUG_ NIL (ST(x 1))) (ST(x 3)) )
     |  Type COLON
          -> ( let w = ST(x 2)
                 in
                    Is_tag w COLON   ->  COLON_ (w 1 aug x 1) (w 2)
                 |  COLON_ (nil aug x 1) w
```

```
        )
|  Type LET
     -> ( let w = D(x 1)  // Standardize the definition.
          in
          GAMMA_ ( LAMBDA_ (w 1) (ST(x 2)) ) (ST (w 2))
        )
|  Type WHERE  ->  ST(LET_ (x 2) (x 1))
|  Type VALOF
     -> ( let w = GAMMA_ PI 1_
          in
          let v = COLON_ (nil aug RHO) w  // RHO: w
          in
          let u = ASSIGN_ PI (AUG_ NIL (ST (x 1)))
          in
          GAMMA_ (LAMBDA_ PI (ALPHA_ u v)) NIL
        )
|  Type RES
     -> ( let w = ASSIGN_ PI (AUG_ NIL (ST (x 1)))
          in
          ALPHA_ w (GOTO_ RHO)
        )
|  Type AP  ->  GAMMA_ (ST(x 1)) (ST(x 2))
|  Type BINOP
     ->  GAMMA_ ( GAMMA_ (CONSTANT, x 3) (ST(x 1)) ) (ST(x 2))
|  Type UNOP
     ->  GAMMA_ (CONSTANT, x 2) (ST(x 1))
|  Type ASSIGN
     -> ( let u, v = ST(x 1), ST(x 2)
          in
          is_tag (x 1) COMMA
          ->  GAMMA_ (GAMMA_ Assign_VAR u) v
          |   ASSIGN_ u v
        )
|  Type GO_TO or Type DOLLAR
     ->  Tag (Get_tag x) (nil aug ST(x 1))
|  Type AUG or Type ALPHA or Type COLON or Type WHILE
     ->  Tag (Get_tag x) (ST(x 1), ST(x 2))
|  Error 'improper node found in ST'
```

```
//  The function LL processes labels, bringing each label as far
//  up the tree as possible.  The effect is that each label is
//  declared by a DELTA node as soon as its scope is entered.

def       Combine(x, y) = Q 1 x
               where rec Q k s =
                         k > Order y -> s | Q (k+1) (s aug y k)
          within
          Proc_labels x =
               Is_tag x DELTA
               ->  (x 1, x 2)
               |   (nil, x)
          within
          Combine_labels(u, v)  =
               let U = Proc_labels u
               and V = Proc_labels v
               in
               Combine(U 1, V 1), (U 2, V 2)
          within

rec LL x =


     let Type = Is_tag x
     in
        Is_identifier x  ->  x
     |  Type ALPHA
          -> ( let s, w = Combine_labels( LL(x 1), LL(x 2) )
               in
               DELTA_ s ( ALPHA_ (w 1) (w 2) )
             )
     |  Type BETA
          -> ( let s, w = Combine_labels( LL(x 2), LL(x 3) )
               in
               DELTA_ s ( BETA_ (LL(x 1)) (w 1) (w 2) )
             )
     |  Type WHILE
          -> ( let s, w = Proc_labels( LL(x 2) )
               in
               DELTA_ s (WHILE_ (LL(x 1)) w)
             )
     |  Type COLON
          -> ( let L, z = Proc_labels(LL(x 2))
               in
               let w = Lval z
               in
               DELTA_ (Q 1 L) w
               where rec Q k t =
                         k > Order(x 1) -> t | Q (k+1) (t aug x 1 k aug w)
             )
     |  Type LAMBDA   ->  LAMBDA_ (x 1) (LL(x 2))
     |  Sons x eq 1   ->  Tag (Get_tag x) ( nil aug LL(x 1) )
     |  Sons x eq 2   ->  Tag (Get_tag x) ( LL(x 1), LL(x 2) )
     |  Error 'improper node in LL'
```

```
//  The function FF flattens a standardized tree into a
//  control structure.

def rec FF(x, c)  =  // Flatten standardized tree x onto control c.

     let Type = Is_tag x
     in
        Is_identifier x -> (x, c)
     |  Is_address x -> ( Update(x, FF(Contents x, c)); (x, nil) )
     |  Type LAMBDA
          -> ( let Body = Lval(  FF( x 2, nil ) )
               in
               Cons_lambda_exp(x 1, Body), c
             )
     |  Type BETA
          -> ( let TA = Lval( FF(x 2, nil) )  // True arm.
               and FA = Lval( FF(x 3, nil) )  // False arm.
               in
               FF( x 1, (BETA, (FA, ( TA, c))) )
             )
     |  Type ALPHA
          -> ( let Rest = Lval( FF(x 2, c) ), nil
               in
               FF(x 1, (ALPHA, Rest))
             )
     |  Type DELTA
          ->   ( ( DELTA, (x 1, Lval(FF(x 2, nil))) ), c )
     |  Type WHILE
          -> ( let w = Lval NIL
               in
               let TA = Lval(FF(x 2, (ALPHA, (w, nil))))
               and FA = Lval(DUMMY, c)
               in
               Update ( w, FF(x 1, (BETA, (FA, (TA, nil)))) );
               (w, nil)
             )
     |  Sons x eq 2 -> FF( x 2, FF( x 1, (Get_tag x, c) ) )
     |  Sons x eq 1 -> FF( x 1, (Get_tag x, c) )
     |  Error 'improper node found in FF'


//  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *

def Translate P  =  // The routine that does all the work.
     FF ( LL( ST P ), nil )
```

```
def Eval_constant () =
     C, S  :=  r C, Push(Value_of(t C), $ S)

and Eval_variable () =
     C, S  :=  r C, Push ( Lookup(t C), $ S )

and Eval_lambda_exp () =
     C, S  :=  r C, Push(Cons_closure(t C, $ E), $ S)

and Hop () =
     C := Prefix ( Contents(t C), r C )

and Make_labels () =
     let V = t C 2  // Value part of the DELTA node.
     in
     let L, k = V 1, Order(V 1)  // Labels being declared.
     and New_C = Prefix(Contents(V 2), Push(RETURN, r C))
     and New_S = Push($ E, $ S)  // Stack to go on with.
     and New_E = $ E  // Environment that will have labels in it.
     in
     while k > 1 do  // Cycle through the labels.
       (  let Lab = LABEL, (Push(L k, r C), $ S, New_E)  // A label.
          in
          New_E := L(k-1), Lval Lab, $ New_E;  // Update New_E.
          k := k - 2
       );
     C, S, E := New_C, New_S, New_E

and Do_alpha() =  // Semicolon - discard top stack item.
     C, S := r C, r S

and Do_assign () =
     if Is_address(t S) do Update(t S, Rval(2d S));
     C, S  :=  r C, Push(DUMMY, r2 S)

and Do_return() =
     C, S, E := r C, Push(t S, r2 S), 2d S

and LtoR () =  // Replace L-value at stack top by R-value.
     S := Push(Contents(t S), r S)

and Do_conditional () =
     let Selected_arm = Contents( (Val_of(t S) -> 3d | 2d) C )
     in
     C, S := Prefix(Selected_arm, r3 C), r S

and Jump () =
     unless Is_label(t S) do Error 'goto to non-label';
     C, S, E := t S 2

and Do_dollar () =  // Do nothing -- Transform has done the work.
     C := r C
```

```
and Do_aug () =  // aug
     let New_S = Aug (t S) (Lval(2d S))
     in
     C, S  :=  r C, Push(New_S, r2 S)

and Apply_closure () =
     let Rator = t S
     and Rand  = Lval(2d S)
     in
     let New_C = Prefix(Contents(Body Rator), Push(RETURN, r C))
     and New_S = Push($ E, r2 S)
     and New_E = Decompose(bV Rator, Rand, Env Rator)
     in
     C, S, E := New_C, New_S, New_E

and Apply_constant () =
     let V = Apply (t S) (Rval(2d S))
     in
     C, S := r C, Push(V, r2 S)

and Apply_tuple () =
     let V = Apply (t S) (Rval(2d S))
     in
     C, S  :=  r C, Push(V, r2 S)
```

```
//  Main program for the jumping evaluator.


def Transform ()  =  // Do one step of an evaluation.
     let x = t C  // Top of control.
     in
        Is_constant x     -> Eval_constant nil
     |  Is_variable x     -> Eval_variable nil
     |  Is_lambda_exp x   -> Eval_lambda_exp nil
     |  Is_address x      -> Hop nil
     |  Is_delta x        -> Make_labels nil
     |  x eq ALPHA        -> Do_alpha nil
     |  x eq ASSIGN       -> Do_assign nil
     |  x eq RETURN       -> Do_return nil
     |  Is_address (t S)  -> LtoR nil  // R-value to top of stack.
     |  x eq BETA         -> Do_conditional nil
     |  x eq GO_TO        -> Jump nil
     |  x eq DOLLAR       -> Do_dollar nil
     |  x eq AUG          -> Do_aug nil
     |  x eq GAMMA
          -> ( let r = t S  // The rator.
               in
                  Is_closure r      ->  Apply_closure nil
               |  Is_constant r     ->  Apply_constant nil
               |  Is_tuple r        ->  Apply_tuple nil
               |  Error 'improper rator'
             )
     |  Error 'bad control'


def Gedanken_Evaluator Program =
     Initialize_memory nil;
     C := Translate Program;  // Set up the Control.
     S := Empty_stack;  // Initial stack.
     Initialize_env nil;
     until Null C do Transform nil;
     Rval(t S)
```