# The Structure of the Lisp Compiler

In the course of my work in algebraic symbol manipulation, I have had the pleasure to implement a Lisp system with compiler for several S/360 operating systems. My confederates in this venture were J. Griesmer, J. Harry, and M. Pivovonsky. In the course of that development I had the opportunity to study the history and structure of the Lisp compiler. It is my intention to pass along some of that information.

The first Lisp compiler was written by Robert Brayton with the assistance of David Park, in SAP for the 704. That compiler was started in 1957 and was working in 1960 by which time Brayton left MIT. During that interval of time a Lisp compiler written in Lisp was implemented by Klim Maling but that compiler was apparently dropped. The argument advanced was that Brayton's being written in assembly language, would obviously be faster. Difficulties in maintenance developed when Brayton left the project. After Brayton and Maling, Timothy Hart and Michael Levin wrote a compiler in Lisp which was distributed with the 704 Lisp 1.5 system. The compiler that I am most familiar with and will describe today is a descendant of that compiler.

The Lisp compiler has developed over the years due to

965 - IBM - 04

several factors:

1. The Lisp language has been refined and the compiler has had to keep up.

2. The Lisp system was implemented on new computers with different order codes and memory organizations, therefore the code generation had to change.

3. Bugs were identified and corrected.

4. Various optimizations of the compiler itself and the code it produces were deemed desirable.

To date none of these factors have completely receded into the woodwork and therefore development continues.

To cite just a few of the more sophisticated and active efforts which I consider lambda calculus motivated: GEDANKEN by J. Reynolds; PAL by A. Evans; CPL - Barron et. al.; ECL by B. Wegbreit.

To these should be added the many straight Lisp 1.5 type efforts, of which the Stanford Lisp systems, the BBN-Lisp on PDP-10 with the TENEX time sharing system, BALM on the CDC 6600 by Harrison, et al, are just a few examples. S/360 Lisp is of this latter type and while it is likely to persist for some years could hardly be touted as a penultimate system. I foresee a series of modest improvements. The overridding consideration being improvement but compatibility with the


965-IBM-04

large body of code that we currently have available.

It is with all this in mind that I launch into this presentation of our S/360 Lisp system and the structure of its compiler.

Firstly, the Lisp system should be understood as having:

1. A language
2. A data-model including storage organization and management
3. A supervisor
4. A large set of built in primitives
5. Two evaluation mechanisms, the interpreter and the compiler
6. A definition schemata

All these facets are of course interesting and vital. But today I am concerned only with the compiler and will to some extent, assume that you are familiar with the other facets of the Lisp system. Moreover that you are here mainly to learn about the organization of the compiler and are unconcerned about other topics which I would be otherwise happy to discuss.

The Lisp compiler is a run time function which creates named code bodies. The effect is that new primitive operators are introduced into the system. Such operators may have access to global variables and are free to have global

side-effects. The compiler is written in Lisp. It is rather well understood and indeed correctness proofs of several subset compilers have been published in 1971 by R. London.

As we are concerned with an extremely simple language with typeless data objects, the compiler is not overly complex. The compiler is primarily concerned with variable binding and variable evaluation, and to that end it uses more efficient environment models than the a-list of the interpreter. These environment models are in order of efficiency: stack variables, SPECIAL variables and COMMON variables.

It is a goal of the definition schema in this lisp system that functions may each be defined in any order. This is a real boon to the interactive definition process, but could lead to a class of errors or else result in inefficient code. The errors that I have in mind come when the number of arguments or their evaluation or transmission modes is changed in a redefinition.

In fact that is just what happens and furthermore is considered tolerable. For reasons of efficiency the compiler prefers to think that all bound variables are purely local (unless otherwise informed) for which it has an efficient stack model. As you can see this is an incorrect assumption which allows for more efficient code but makes the programmer identify those variables which may be used free by some called function.


965-IBM-04

Another incorrect assumption that this compiler thrives on is that a call to a currently undefined function is a call to a function that expects exactly the number of arguments given and that it expects them evaluated. This is not so bad in the case the function remains undefined as a run time error occurs. Also any attempt to define it as other than the assumed class can give rise to an error.

The compiler thus assumes that the user declares his free variables correctly and compiles his unusual functions before any calls to it are compiled. The three modes for variables that the compiler uses are:

1. stack variables.
2. SPECIAL variables.
3. COMMON variables.

The implementations of each of these are:

1. Stack variable -- A stack cell is reserved for each bound variable occuring in the lambda-expression, these cells are filled in at entry to the code body with the arguments of the function. A stack cell is allocated for each program variable found as bound variables in embedded PROGs, these are initialized to NIL at function entry. Temporary variables are created by the compiler itself. All of these cells are accessable only to the current activation of the code body in question.

965 - IBM - 04

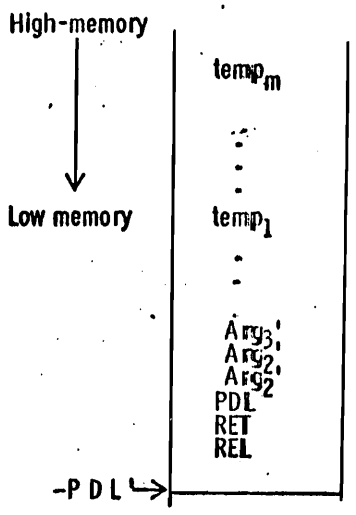



Fig. 1

2. SPECIAL variables -- This mode is used for those variables declared SPECIAL at compile time. A value cell is associated with the identifier which denotes a variable. At binding time the contents of the stack cell allocated to that bound variable is exchanged with the contents of the value cell. Subsequent references and assignments are to the value cell. At function return time these two cells are exchanged again, thus restoring the former values.

The implementation of SPECIAL variables is quite efficient but inadequate in the following cases.

1. The downward FUNARG:

   A functional argument is passed which uses a free variable which has been bound again in the environment the functional value is applied.

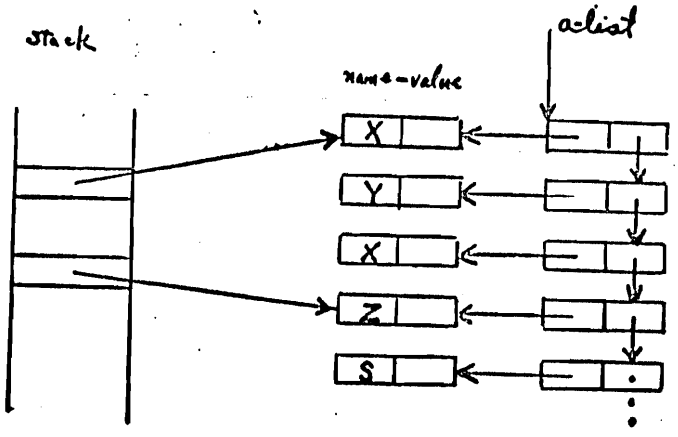
2. The upward FUNARG:

965 - IBM - 04

A functional value is returned which has a free variable which has a different binding when that value is applied.

For these cases the full power of the COMMON variable mode is necessary. The Lisp interpreter has access to SPECIAL values only if no variable by that name occurs on the a-list.

3. COMMON variables -- This is the most correct variable model used because it uses the a-list in a manner isomorphic to the use by EVAL. The embodiment is to have a stack cell point to the name-value pair on the a-list. Evaluation of such a variable is a LOAD and a CDR. Assignment is a LOAD and a RPLACD. When COMMON variables are bound (activated), the a-list is augmented by appending that name-value relationship cell. At deactivation time the a-list is restored to its former value. In those functions which use a COMMON variable free, or for that matter any undeclared operand free variable, the a-list is searched at binding (activation) time and a stack cell is made to point to the appropiate name-value cell.




Compiler treatment of common variables Z and X.


965 - IBM - 04

Fig.

The COMMON variable mechanisms of the compiled code are more efficient than those of the interpreter except that free variables which are mentioned but not used do get extra attention. The COMMON mechanism is less efficient than the stack or SPECIAL variable mechanisms. It is more correct however.

The compiler treats ordinary operator names like free variables and compiles the code to:

1. Evaluating the arguments.
2. Transmit the arguments.
3. Do a subroutine jump indirectly through the SPECIAL value cell of that identifier.

It is the users responsibility to see that any operator that deviates from the ordinary class receive properties which the compiler may know before any function which calls it is compiled.

965 - IBM - 04

Many of these defaults and responsibilities were put in as aids to the interactive program development, but are in fact the sources of some errors.

It is interesting to note that in Brayton's early compiler, the "compile before or along with" conventions were rigidly enforced. A function not satisfying those requirements

didn't compile.

Despite my earlier promise that I would concentrate strictly on the structure of the compiler I should like to describe the Lisp language. My argument is that even those of you who are expert at Lisp need to know precisely what Lisp I am concerned with.

To that end I present a syntax description of Lisp as I see it.

Syntax of LISP

e is: {c | id} a constant or variable name

({LAMBDA / FLAMBDA} {(id*) / id} body) where body is an e
(LABEL id e)
(SETQ id e)
(COND (p q)*) where p, q are e
(PROG (id*) s*)

where s is { id / (GO id) / (RETURN e) / (COND (p s)*) / e }

(FUNCTION e)
(QUOTE s-exp)
(rator rand*) where rator, rand are e.


965 - IBM - 04

A Lisp expression e is:

c a constant.

id a variable name or identifier.

(lambda-op bv body) a lambda-expression or procedure
where lambda-op is LAMBDA or FLAMBDA, and
bv the bound variables is either id or (id*), and
body is an e.

(LABEL id e) a label-expression.

(SETQ id e) assignment.

(COND (p g)*) a conditional-expression
where the predicate p is an e, and
the consequent g is an e.

(PROG pv s*) a program-expression
where pv the program-variable part is (id*), and
each statements s is a:
label which is an id, or
(GO id) a go-statement, or
(RETURN e) a return-statement, or
an e, or
(COND (p s)*) a conditional-statement.

(FUNCTION e) .

(QUOTE s-expression) a quoted s-expression.

(rator rand*) a combination
where the operator rator is an e, and
each operand rand is an e.

It should be noted that except for constants and variables


965 - IBM - 04

every Lisp expression is a combination. Some of these combinations are distinguished for semantic reasons.

Semantics in Lisp is usually given by describing an interpreter (EVAL e a) where e has been described and a is an environment which gives meaning to free variables. The environment a is an ordered set of name-value relationships which is most generally represented as an a-list.

As an aside I should like to mention my own bias that an interpreter described as a S,E,C,D transform is even more illuminating. Such transforms were described by P. Landin in 1964. (see appendix A for such a semantics presentation)

It is well known that Lisp requires a minimal set of data primitives for its own self-description. To define EVAL in Lisp we require only the following additional primitives:

(EQ x y) identity.

(ATOM x) the atom predicate.

(IDENTP x) the identifier predicate.

(CAR x) and (CDR x) the selectors.

(RPLACD x y) and (RPLACA x y) the storing functions.

and an allocation function like CONS.

The fact that very little is required to understand Lisp and the fact that it is self described and self implemented has allowed the users to play with its semantics. Lisp has not been very self protective. Some examples of such introduced semantics are macro-expressions and code-expressions.



965 - IBM - 04

A macro-expression m-e is simply a combination whose rator is understood to be the name of a macro definition function mdef. The value of a macro-expression is given by: (EVAL (mdef m-e ) a) .

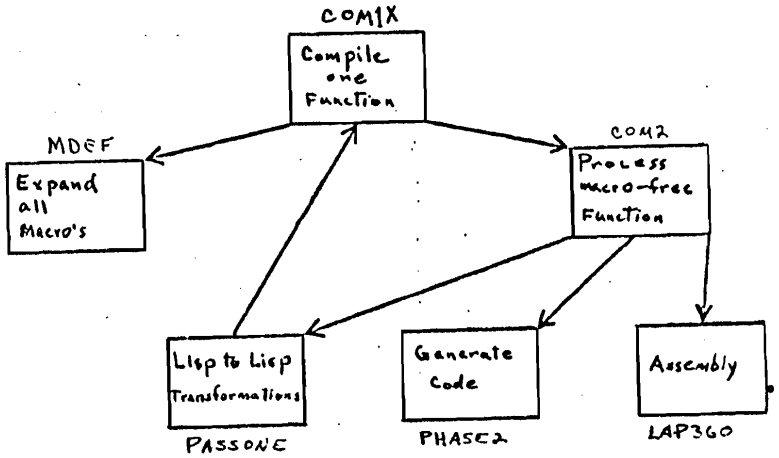
A code-expression is a combination of the form:

(CODE instr*) where the instr are LAP instructions.

Both macro and code expressions are intended as compilation tricks but EVAL could handle them. In practice EVAL handles macro's but not code-expressions.

Skipping any further discussion of semantics I believe we should be in a position to follow a structural description of the compiler.

Each named procedure is compiled by a Lisp function COM1X which has the following structure:

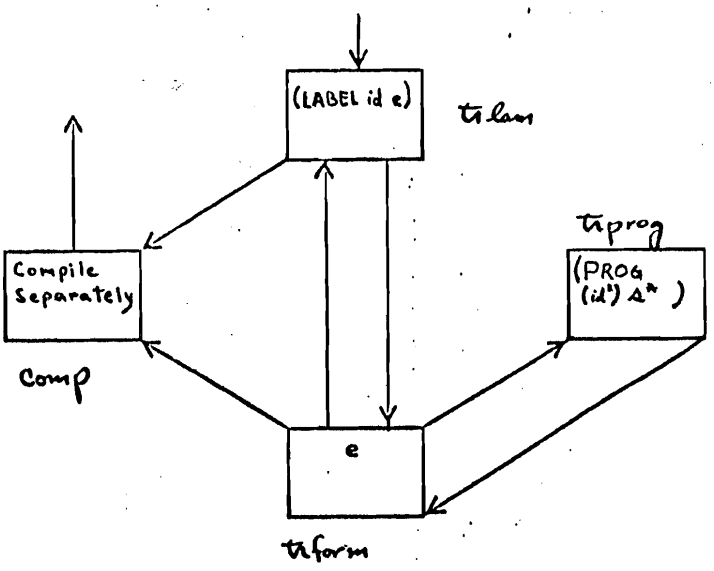


COM1X calls MDEF which expands all the Lisp macros. COM1X is recursively called for certain procedure-valued expressions. In which case a separate function is created and then referenced by the "containing" function.

COM2 controls the compilation of a properly macro-expanded named lambda-expression. There are three parts to this compilation namely, PASSONE, PHASE2, and LAP360.

PASSONE performs such Lisp to Lisp transformations as:

1. Recursion removal
2. Free variable identifictaion
3. Introduction of binding mechanisms
4. Elimination of indefinite number of arguments
5. Elimination of FLAMBDA operators
6. Remove operand procedure-valued expressions
7. Introduce interpreter calls for delayed evaluation conditions




PASSONE

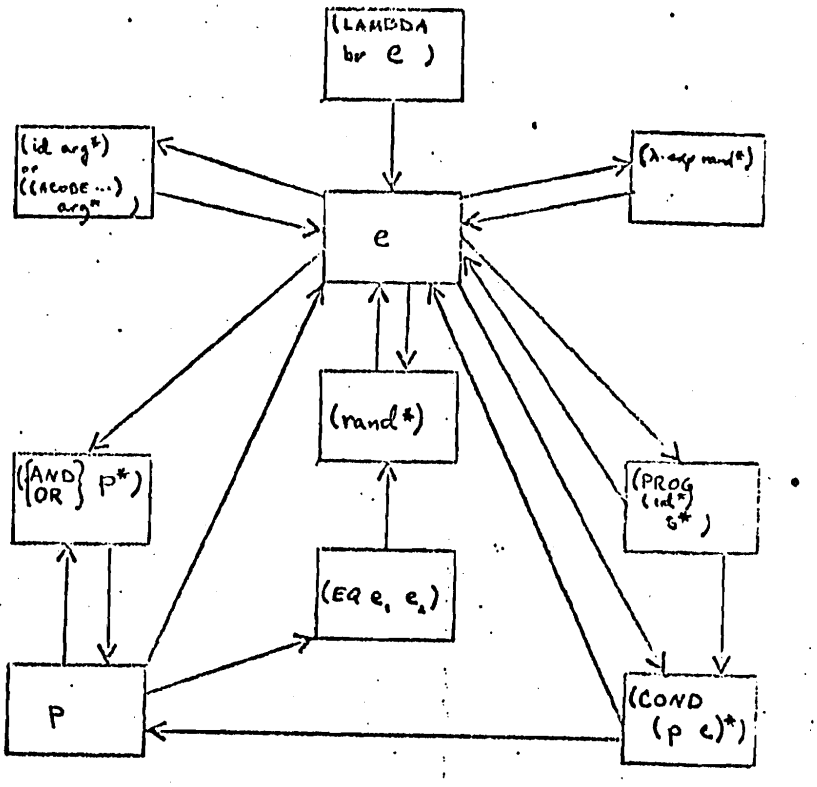
Fig (see appendix B for greater detail on PASSONE.)

PHASE2 recieves a much simplified lambda-expression from PASSONE. It produces a list of LAP (List Assembly Program) instructions which is meant to be the execution equivalent of the lambda-expression. PHASE2 creates the code. It Knows what to do for:

| | |
|---|---|
| (QUOTE s-expression) | constants |
| (SPECIAL id) | |
| (SETQ id e) | assignment |
| (COND(p q)*) | conditionals |
| (PROG pv s*) | program blocks |
| (RETURN e) | program returns |
| (GO id) | go-tos |
| (AND e*) | |
| (OR e*) | |
| (*CODE instr*) | code-expressions |
| ((LAMBDA(id*)e)e) | lambda-exp operator |
| (id e*) | calls |

PHASE2 also generates the code for variable binding prologues and variable restores on value return epilogues.


965 - IBM - 04




(see appendix C for greater detail on PHASE2)

LAP360 is a very simple minded 2-pass assembler. The only interesting fact worth mentioning is that it finally resolves special variables, quotes and function calls.


965-IBM-04

References

Reynolds, J. C., GEDANKEN, A Simple Typeless Language Based on the Principle of Completeness and the Reference Concept. CACM 13, 5 (1970) 308-319

Evans, A., PAL - a language designed for teaching programming linguistics, ACM 23rd National Conference, August 1968.

Barron, D., et al., The main features of CPL, Computer Journal 6, 2, July 1963.

Landin, P., The Mechanical Evaluation of Expressions, Computer Journal 6, 4, 1964.

London, R. L., Correctness of two compilers for a Lisp subset, Artificial Intelligence Memo AIM-151, Stanford University, October 1971.

Wegbreit, B., The ECL programming system, Aiken Computation Laboratory (Harvard University, 1971 (in preparation)).

965 - IBM - 04

# APPENDIX A. Semantics ---



965 - IBM - 04

965 - IBM - 04

# APPENDIX B. PASSONE transformations ---



965 - IBM - 04

Iterative form recursion removal

trlam[ (LABEL x (LAMBDA ($id_1 \ldots id_n$)
(COND {(p q) | ($p_i$ ($x\ a_{i1} \ldots a_{in}$))}* ))); b]

- trlam[ (LABEL x (LAMBDA ($id_1 \ldots id_n$)
(P**G ($g_1 \ldots g_n$)
$g_s$ (COND {(p (RETURN q)) | ($p_i$ (GO $g_i$))}* )
{$g_i$ (SETQ $g_1\ a_{i1}$) ... (SETQ $g_n\ a_{in}$) (GO $g_t$)}
$g_t$ (SETQ $id_1\ g_1$) ... (SETQ $id_n\ g_{in}$) (GO $g_s$) ))); b]

trlam[ (LABEL x (LAMBDA bv (PROG ...))); b]

- trlam[ (LABEL x (LAMBDA bv (P**G ...))); b]

trlam[ (LABEL x (LAMBDA bv body)); b]

- (LABEL x (LAMBDA bv {trform[ body; bv: b]} /
(PROG pv
alist-save
common-var-bind
free-var-bind
special-var-bind
(SETQ $g_v$ trform[ body; bv: b] )
special-restore
alist-restore
(RETURN $g_v$) ) } ) )

where pv = ( {$g_a$ | empty} $g_v$ free* )

Constant forms

tform[ var ; b] where var is an id ∉ L and not declared special.
= (COMMON var) ~~and var is added to the last free.~~
and
free := (var • free)

if var were declared special then
= (SPECIAL var)

---

if var ∈ b but declared:
{(COMMON var)} if common
{(SPECIAL var)} if special

if var not declared but var ∈ b
then var

---

tform[T ; b] = (QUOTE *T*)
tform[F ; b] = (QUOTE NIL)
tform[x ; b] = (QUOTE x) if atom[x] ∧ ~idstp[
tform[(QUOTE x); b] = (QUOTE x)

space

space


965 - IBM - 04

tfform[(id rand*); b]
where id is undefined and undeclared and free,
or is defined as ⊏ SUBR or EXPR.
= (id tfform[rand; b]*)

tfform[(id rand*); b]
where id is defined as FSUBR or FEXPR
= (id (QUOTE rand)*)

tfform[(id rand*); b]
where id is defined as SUBR* or EXPR*
= (id (LIST ⊏ tfform[rand; b]*))

tfform[(id rand*); b]
where id is defined as FSUBR* or FEXPR*
= (id (LIST (QUOTE rand)*))

tfform[(id rand*); b]
where id is either ∈ b or special or common.
= (APPLX tfform[id; b] (LIST tfform[rand]*))

note: assumes arguments evaluated.

trform[(SETQ id e); b]
= {(RPLACD id e') / (SETQ (SPECIAL id) e') /
(SETQ id e')} where e' = trform[e; b]

trform[(COND (p q)*); b]
= (COND (trform[p;b] trform[q; b])*)

trform[(GO id); b] = (GO id)

trform[(AND p*); b] = (AND trform[p; b]*)
trform[(OR p*); b] = (OR trform[p; b]*)

trform[(FUNCTION e); b] = (FUNC (QUOTE comp[e;g]) $ALIST)

trform[(*CODE instr*); b] = (*CODE instr*)


965-IBM-04

$trform[(P^{**}G\ pv\ s^*);b] = trprog[(PROG\ pv\ s^*);b]$

$trform[(PROG\ (id_1 \ldots id_n)\ s^*);b] = trprog[(PROG\ (id_1 \ldots id_n)\ (SETQ\ id_1\ (QUOTE\ NIL)) \ldots (SETQ\ id_n\ (QUOTE\ NIL))\ s^*);b]$

$trform[\lambda\text{-}exp;b] = (FUNC\ (QUOTE\ comp[\lambda\text{-}exp;g])\ \$ALIST)$

$trform[((LABEL\ id\ e)\ rand^*);b] = trform[(trlam[(LABEL\ id\ e);b]\ rand^*);b]$

$trform[((FUNCTION\ e)\ rand^*);b] = trform[(e\ rand^*);b]$

$trform[((QUOTE\ s\text{-}exp)\ rand^*);b] = trform[(s\text{-}exp\ rand^*);b]$

$trform[((^*CODE\ instr^*)\ rand^*);b] = ((^*CODE\ instr^*)\ trform[rand;b]^*)$

$trform[((rator\ rand^*)\ rand'^*);b] = (APPLX\ trform[(rator\ rand^*);b]\ (LIST\ trform\ rand';b\ ^*))$

$trprog[(PROG\ pv\ s^*);b] =$

(PROG pv''
alist-save
common-var-bind
free-var-bind
special-var-bind
(SETQ $g_v$ (PROG pv' s'))
special-restore
alist-restore
(RETURN $g_v$))

where $pv'' = (g_v\ coms^*\ spec^*\ free^*)$
$pv' = pv - [coms^* \cup spec^* \cup free^*]$

$trform[((LAMBDA\ (id^*)\ body)\ rand^*);b] = (trlam[(LABEL\ g\ (LAMBDA\ (id^*)\ body));b]\ trform[rand;b]^*)$

$trform[((LAMBDA\ id\ body)\ rand^*);b] = (trlam[(LABEL\ g\ (LAMBDA\ (id)\ body));b]\ (LIST\ trform\ rand;b\ ^*))$

$trform[((FLAMBDA\ (id^*)\ body)\ rand^*);b] = (trlam[(LABEL\ g\ (LAMBDA\ (id^*)\ body));b]\ (QUOTE\ rand)^*)$

$trform[((FLAMBDA\ id\ body)\ rand^*);b] = (trlam[(LABEL\ g\ (LAMBDA\ id\ body));b]\ (QUOTE\ (rand^*)))$

# APPENDIX C. PHASE2 code generations ---

comval [ id; stomap; name]

= (L AC locate [id] [stomap] )

where stomap = ( (id$_j$ (PDL 4k) ) (id$_{j-1}$ (PDL 4[k-1] ))...)

(PDL 4k) is the stack address for the local bound variable id$_j$ .

comval [(QUOTE s-exp); stomap; name]

= (L AC (QUOTE s-exp) )

comval [(SPECIAL id); stomap; name]

= (L AC (SPECIAL id) )

comval [(id rand$_i$*); stomap; name] for i > 2

= ( comval [ rand$_1$; stomap; g$_1$] :
(ST AC locate [g$_1$] [stomap'] ) :
comval [ rand$_2$; stomap; g$_2$] :
(ST AC locate [g$_2$] [stomap'] ) :
.
.
comval [ rand$_n$; stomap; g$_n$] :
(ST AC locate [g$_n$] [stomap'] ) :
(L AC locate [g$_1$] [stomap'] ) :
(L MQ locate [g$_2$] [stoamp'] ) :
(LA PDL 0 locate [g$_3$] [stomap'] ) :
(CALL id n) )

comval [ ( (LAMBDA (id$_1$... id$_n$) body) e$_1$ ... e$_n$); stomap; name]

= (comval [ e$_1$; stomap; g$_1$] :
(ST AC locate [id$_1$] [stomap'] ) :
.
.
comval [ e$_n$; stomap; g$_n$] :
(ST AC locate [ id$_n$] [stomap'] ) :
comval [ body; stomap'; name] )

where stomap = ( (id$_j$ (PDL 4k)) (id$_{j-1}$ (PDL 4[k-1] ) )... )

and stomap' =
( (id$_n$ (PDL 4[k+n] ) )... (id$_1$ (PDL 4[k+1] ) ) (id$_j$ (PDL 4k) )... )

comval [ (COND (p$_i$ e$_i$)* ); stomap; name]

= ( { comval [ p$_i$; stomap; g$_i$] :
(CR AC NIL) :
( {BNE / BE} 0 (LABEL g$_{i,1}$) ) :
comval [ e$_i$; stomap; name] :
(B 0 (LABEL name) ) :
g$_{i,1}$ }
.
.
for i = 1... * :
name )

comval [ (PROG pv s*); stomap; name]

where stomap = ( ($id_j$ (PDL 4k)) ($id_{j-1}$ (PDL 4[k-1])) ... )

= comprog [ (s*); pv; name]

I. (s'*) is created from (s*) by substitution of $g_i$ for each $label_i$ in (s*)

II. stomap' is created from pv = ($id_1$ $id_2$ ...) and stomap.

III. The code for each s' in (s'*) is created:

if atom [ s'] ⟶ s'

if s' = (GO $label_i$) ⟶ (B 0 (LABEL $g_i$))

if s' = (COND (p s)*) ⟶ comcond [ ((ps)*); NIL]

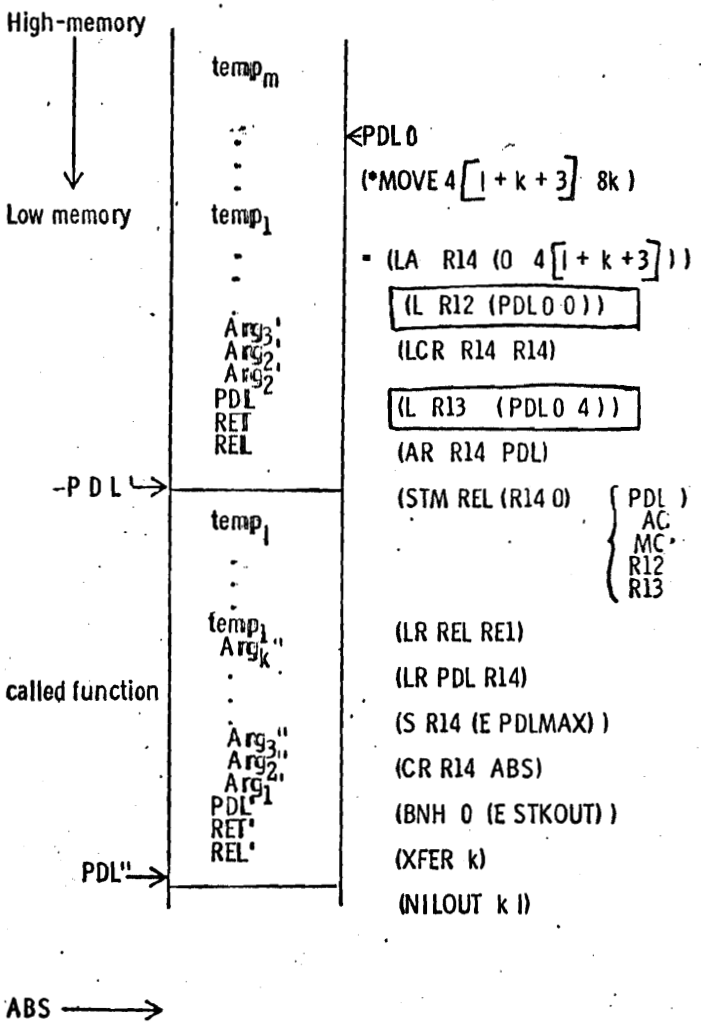
else comval [ s'; stomap'; g]

IV. The label name is placed at the end of all this code.

* ($c_1$ : $c_2$ : ... $c_n$ name) where $c_i$ is a list of code for $s_i$.

965 - IBM - 04

STACK PLAN 72 - 2



(XFER $\not{F}$ k )

if k < 5 ⟶ ( ) .

else (MVC ____(PDL 28) 4[k-4] (PDL 0 16) )

(NILOUT k $\not{l}$ )

if = 0 ⟶ ( )

= 1 ⟶ (ST NIL (PDL 4k + 12) )

= 2 ⟶ (ST NIL (PDL 4k + 12)) (ST NIL (PDL 4k + 16))

else ⟶ (ST NIL (PDL 4k + 12) )
(MVC _____(PDL 4k + 12) 4[$\not{l}$ -1] (PDL 4k - 16) )