The research reported in this paper was sponsored by the Advanced Research Projects Agency Information Processing Techniques Office and was monitored by the Electronic Systems Division, Air Force Systems Command under contract F1962867C0004, Information Processing Techniques, with the System Development Corporation.

TECH MEMO *a working paper*

System Development Corporation / 2500 Colorado Avenue / Santa Monica, California 90406
Information International Inc. / 11161 Pico Boulevard / Los Angeles, California 90064

TM-3417/340/00

AUTHOR Jeff Barnett
J. Barnett

TECHNICAL Jeff Barnett
J. Barnett

RELEASE Clark Weissman
C. Weissman, S.D.C.
[illegible]
D. Anschultz, I.I.I.

for J. I. Schwartz

DATE 4/26/67 PAGE 1 OF 26 PAGES

LISP 2 Compiler Context Resolver Language and Processor Specifications

ABSTRACT

This document describes the language and processor required for the Context Resolver pass of the LISP 2 compiler proposed for the IBM S/360 computer. The Context Resolver (pass II of the LISP 2 compiler) is used to macro-expand Intermediate Language inputs into a list of Context-Resolved Interlude Language function definitions.

TABLE OF CONTENTS

TABLE OF CONTENTS (Cont.)

TABLE OF CONTENTS (Cont.)

1. CONTEXT-RESOLVED INTERLUDE LANGUAGE

1.1 INTRODUCTION

The Context-Resolved Interlude Language (CRIL) is predicated on a computer with the following configuration:

1. A cluster of locatable and addressable fields
2. A "heaven-box" which is infinite in length

A field is a finite-length box or container that holds the surface of a datum, a locator, or an indirection. An indirection is a kind of locator generated by the system whose existence is not guaranteed by the language, e.g., the binding cell of a non-locative PUBLIC variable contains an indirection.

The heaven-box is a combination of general-purpose hardware registers and compiler passive-state variables. The heaven-box may hold anything a field may; the contents are assumed to be infinite in length, i.e., there are no sign-extension or precision-truncation problems when moving quantities through the heaven-box. All data constants are received through the heaven-box. It may not be "located" or "indirected" within the CRIL language, even though it may hold a locator or indirection.

1.2 MEMORY REFERENCE

1.2.1 Variables

```
variable  =  gname|ℓname

gname     =  (identifier . sname)

sname     =  identifier (id ≠ LEX.)

ℓname     =  (identifier . LEX.)

fname     =  (identifier . sname)
```

1.2.1.1 Sname

An sname is a section name for a variable. Any identifier with the exception of LEX. may be used.

1.2.1.2 Gname

A gname is the full name of a global variable. The identifier is its first and most used name. The sname is its section name.

1.2.1.3 ℓname

An ℓname is the name of a lexical variable, a variable found on either the Alist or Aplist. The pseudo-section LEX. is used for uniformity of variable formats.

1.2.1.4 Variable

A variable is either lexical (an ℓname) or global (a gname). The meaning of a CRIL variable is the location of the binding of a variable with the given name. It is like an immediate address with the name being a constant.

1.2.1.5 Fname

An fname is the name of a function, actually a constant. Because of peculiar naming conventions for functions, they look like variables; therefore an fname is handled syntactically as a variable.

1.2.2 Addresses

```
address      =  locator|indirection

indirection  =  ifield

ifield       =  (IFIELD coordinate address)

               (variable         )
locator      = < computed-address >
               ( locof            )
               ( afield           )

coordinate   =  (offset . length)

offset       =  integer

length       =  integer

afield       =  (AFIELD coordinate address)

               ( aplus              )
computed-      ( integer            )
address      = < loc-assignment     >
               ( address-expression )

aplus        =  (APLUS address integer-expression)
```

1.2.2.1 Address

An address is a rule for computing the location of an entity in the computer's memory.

1.2.2.2 Length

The length is the number of bits of an entity.

1.2.2.3 Offset

The offset is the number of bits that an entity lies to the left (maybe right) of an address.

1.2.2.4 Coordinate

Given a computable address, the coordinate gives explicit instructions on how to locate an entity in a computer memory.

1.2.2.5 Aplus

An aplus is used to compute run-time offsets for such things as array indexing. The integer expression is the offset from the address. Assume A is an array whose dimension is 10X10. Then the address of A(I, J) would be (APLUS L(A) (PLUS J (TIMES 10 (PLUS I -1)))) where L(A) is the locator of the array A.

1.2.2.6 Computed-address

A computed-address is a locator that by its nature must reside in the heaven-box. This mechanism is useful for such things as expansions of locatively valued functions, CAR, etc. The interpretation of an explicit integer used as a computed-address is as an intermediate address.

1.2.2.7 Ifield

An ifield specifies that the given address and coordinate are a rule for finding an address that resides in the computer's memory. The indirection implied by the ifield is needed because of something not part of the language specification. Hence, something located by an ifield is not locatable unless the thing located is locatable outside the context of the ifield. An example of this situation is that the binding cell of a non-locative fluid contains the address of the value. It is not legal to locset such a fluid. See explanation of afield for contrast.

1.2.2.8 Indirection

An indirection is a rule for computing a non-language-guaranteed locator.

1.2.2.9 Afield

An afield specifies that the given address and coordinate are a rule for finding an address that resides in the computer memory. The indirection implied by the afield is through a locator guaranteed by the semantics of the language translated into CRIL. An address located by an afield is a candidate for locsetting. An example of such an item is the contents of the binding of a locative variable.

1.2.2.10 Locator

A locator is a rule for finding anything in the computer's memory, the existence of which is guaranteed by language semantics.

1.2.2.11 Address-expression

An address-expression is interpreted as an unsigned integer that specifies a memory location. Some such form as (CALLIT ADDRESS expression) will be used to denote this entity.

1.2.3 Locatives

locative = field|address

field = (FIELD type coordinate address)

locof = (LOCOF field)

1.2.3.1 Field

A field is the description of a non-locator entity in the computer's memory. The coordinate and address specify the location of the datum, and the type specifies the format.

1.2.3.2 Locof

A locof is used to find the location of a field. The address part of the embedded field is the value of the embedding locof. (See Section 1.6.9 for a description of equivalent ℓfield-locof.)

1.2.3.3 Locative

Locatives are things that reside in the computer's memory or locate things there.

1.3 CONSTANTS

$$\text{constant} = \begin{Bmatrix} \text{datum} \\ \text{funarg} \end{Bmatrix}$$

datum = (DATUM type lap-address)

1.3.1 Datum

A data constant never stands alone in CRIL. Rather, its lap-address (see LAP document) is wrapped within a datum form. This allows for compile-time "cheats" and also alleviates the necessity for the continued use of "what-type-is-it" functions. The type specifies the format that the datum is assumed to be in. This mechanism may also be used to reference registers by using appropriate LAP mnemonics.

1.3.2 Funarg

Funargs are constants in the normal meaning. However, a discussion of funarg syntax and semantics is delayed until after the exposition on expressions. All constants are supplied through the heaven-box and therefore are not locatable, hence they are not "settable."

1.4 EXPRESSIONS

$$\text{expression} = \left\{\begin{array}{l}\text{constant}\\ \text{locative}\\ \text{complex}\end{array}\right\}$$

$$\text{complex} = \left\{\begin{array}{l}\text{transformation}\\ \text{arithmetic}\\ \text{assignment}\\ \text{terminal}\\ \text{pterm}\\ \text{compare}\\ \text{function call}\end{array}\right\}$$

An expression is an entity that produces a value, either a locator or a datum. Expressions are the basic building blocks of a language and their semantics describes the capabilities offered. Constants and locatives are obvious instances of expressions.

1.4.1 Complex

Complex expressions are expressions which are neither constants nor locatives.

1.4.2 Transformations

Transformations are mechanisms in CRIL to make type information explicit. They are used in the expansions of function calls, "cheaters" and the like.

1.4.2.1 Drive

A drive causes the following action. If the embedded expression evaluates to the type specified by the embedded type, then the valuation is exactly as if the expression had appeared without the embedding drive. If the natural type is different, then the drive results in a conversion of the value of the embedded expression to the homomorph in the driven type. For example,

```
(DRIVE REAL 3.0)  =  3.0

(DRIVE REAL 3  )  =  3.0
```

1.4.2.2 Callit

Callit evaluates the embedded expression with no driving of types. The datum produced is then assumed to be in the format specified by the embedded type. The value of CALLIT is locative or not as the embedded expression, and has the same coordinate (if applicable). For example, on the Q-32

```
          (CALLIT OCTAL 1.0)  =  2001400--Q
```

The use of callit is for expansion of "cheaters" from IL and for various other machine-dependent features.

1.4.3 Arithmetic Forms

```
arithmetic  = { minus
                plus
                times }

minus       =  (- numeric-expression)

plus        =  (+ numeric-expression*)

times       =  (* numeric expression*)
```

Arithmetics are a set of forms that do arithmetic computations and are fairly self-explanatory. The rules and algorithms governing the type of the produced value are described in TM-3417/200/00. All values produced by the arithmetics are unfielded, even (- (- variable)) or + variable.

1.4.4 Assignments

```
assignment         = { normal-assignment
                       loc-assignment    }

normal-assignment  =  (← field expression)

loc-assignment     =  (← (LOCOF(FIELD type coordinate (AFIELD coordinate address)))
                              address)
```

Assignments are the mechanism by which fields in the computer's memory may be altered. Both the IL "←" and "→" operators expand in these forms.

1.4.4.1 Normal-assignments

A normal-assignment is the expansion of the IL "←" operator. The value is an unfielded copy of the value of the embedded expression. The action is to take a copy of the value of the expression, convert it to the format prescribed by the field, and replace the contents of the field by the converted copy. The expression is evaluated only once.

1.4.4.2 Loc-assignment

A loc-assignment is the expression of the IL "→" operator. The effect is to place a copy of the embedded address (the right-most one) into the address field specified by the address part of the AFIELD. The value of the loc-assignment is the right-most address. If the original LOCSET was used as an expression, an appropriate field may embed the loc-assignment as an address.

1.4.5 <u>Comparisons</u>

compare	=	(COMPARE relational expression expression)
relational	=	arithmetic-relation|equality-relation
arithmetic-relation	=	$>\,\|\,<\,\|\,\geq\,\|\,\leq$
equality-relation	=	machine-equality|data-equality
machine-equality	=	≡
data-equality	=	=

A compare is a boolean-valued form. It yields value TRUE if the first embedded expression stands in the stated relationship to the second embedded expression, and FALSE otherwise.

1.4.5.1 Relational

Relationals are the set of possible comparisons to be done by a compare.

1.4.5.2 Arithmetic-relational

These do comparisons for numeric quantities of the same or different types. The meanings of the various comparisons are the normal ones.

1.4.5.3 Equality-relation

A machine-equality is a relation that tests TRUE if the two data being compared have precisely the same "substrate." What this means may vary from implementation to implementation. In general, this is the comparitor used for uniquely represented data. Data-equality is described in TM-3417/200/00 (see the semantic description of SL and IL).

1.4.6 Terminals

terminal = (TERMINAL statement)

A terminal is an expression that evaluates as follows: The embedded statement is executed. If the embedded statement is a return or if a return statement embedded in it is executed, the value of the terminal is the value of the expression body of the return. Labels in the terminal are invisible outside of it, and outside labels are invisible inside. (See the description of the return statement for more information.) If no return statement is evaluated within a terminal, a guess is made as to what value to retain: 0, 0Q, (), etc.

1.4.7 Pterm

pterm = (PTERM statement)

A pterm works similarly to a terminal and interacts with preturns instead of returns. A pterm is an indication that the evaluation is a boolean one where a data value is not needed, but rather placement of program control is conditional. The usage is for expansions of blocks in predicate context.

1.5 STATEMENTS

$$\text{statement} = \left\{\begin{array}{l}\text{conditional}\\ \text{return}\\ \text{preturn}\\ \text{go}\\ \text{computed-go}\\ \text{code}\\ \text{binder}\\ \text{expression}\\ \text{compound-statement}\\ \text{NIL}\end{array}\right\}$$

label = atom

A statement is a member of the class of language entities used for obtaining a side effect, not a value. In fact, if an ordinary expression is used as a statement the value produced is discarded.

1.5.1 Label

A label is a marker in a program. Several language facilities may request transfer of "program control" to a label. This means "continue execution in the program at the lexical right side of the label". If several labels are given in a row, the transfer of control to any of them is computationally equivalent to transferring to the right-most label.

1.5.2 Compounds

compound-statement = (COMPOUND $\{$statement|label$\}_*$)

The statements embedded in a compound are executed left to right unless a control-transferring statement is executed. If such is encountered, control is passed to the appropriate label, and left-to-right execution proceeds from there. If program control passes through the right-most statement, the execution of the compound statement is completed. What happens next depends upon what the compound is embedded in. Not all compound-statements complete execution, as embedded transfer statements may move control out of this compound-statement. All labels visible directly outside a compound-statement are visible inside. Labels directly embedded in a compound-statement are markers and may be seen outside.

1.5.3 Direct Transfers

go = (GO label)

computed-go = (CASEGO integer-expression label$_{*+1}$)

1.5.3.1 Go

A go transfers control to the specified label. If no label of the given "name" is visible, an error condition exists. The label in a go is not a marker, only a reference to a marker.

1.5.3.2 Computed-go

The embedded integer-expression is evaluated. If the value is one, control is transferred to the first label; if the value is two, control is transferred to the second label, etc. If the value is less than one or greater than the number of labels given, control is transferred to the last label. If one or more of the embedded labels are not visible, an error exists. The labels in a computed-go are not markers, only references to markers.

1.5.4 Conditional

conditional = (COND boolean-expression statement)

The embedded boolean-expression is evaluated. If its value is FALSE, execution of the conditional is completed. If the value is TRUE, the embedded statement is executed. Completion of execution of the embedded statement completes execution of the conditional. Visibility rules for labels and control transfer for the embedded statement are precisely the same as if the statement had not been embedded in the conditional.

1.5.5 Binder

binder = (BIND variable-declaration-with-preset statement)

variable-declaration-with-preset = {undefined}

A binder causes the variables specified by the variable-declaration-with-preset to be bound and visible during the execution of the embedded statement. Upon completing execution of the statement, or program control leaving the binder for any reason, the variables are unbound. (For full variable visibility rules, see the SL and IL descriptions.) Labels visible directly outside the binder are visible inside. Labels appearing within a binder are not visible outside. A visible label from outside a binder may have the same name as a label inside the embedded statement. References, in this case, are transfers to the inner label. If two labels' definitions of the same name are encountered at the same binder "level," an error condition exists.

For example,

```
(COMPOUND L(BIND ()

             (COMPOUND

                (IF P (GO L))

                (SET A B)

              L (SET B C))))
```

In this example, (GO L) transfers control to the label L in front of the assignment, not to the label L in front of the binder. If the inner L was not there, control would pass to the outer L. The use of the two L labels is not an error, but rather a feature of block-structuring.

1.5.6 Return

return = (RETURN expression)

A return is a statement assumed to be embedded in one or more terminals. If it is not, an error exists. The execution of a return statement involves evaluation of the embedded expression. The value of the expression is used as the value of the innermost terminal embedding the return. The execution of the return completes execution of everything embedded in the terminal, i.e., the return does not "fall through."

1.5.7 Preturn

preturn = (PRETURN expression)

A preturn is a statement that interacts with a pterm in much the same way that a return interacts with a terminal.

1.5.8 Function Calls

function-call = (FNCALL type functional-expression $\{\text{drive}|\text{locof}\}_*$)

A function-call is the mechanism used to direct linkage to an external subroutine. The type describes the value returned and the necessary information to pass the arguments. The functional-expression gives the rule for computing the form operator. It may be anything from a datum (a function name) to a complex calculation needed to save context for a funarg. The string of drives or locofs is the CRIL representation of the arguments. The order of expansion of the form operator and arguments is not guaranteed.

1.6 TOP OF THE LANGUAGE

```
cril-form = function-definition|macro-definition|run-form

function-definition = function-with-value|procedure

function-with-value = (FUNCTION (gname v-type)
                      variable-declaration-without-preset expression
                      external-reflist)

procedure = (FUNCTION (gname NOVALUE)
            variable-declaration-without-preset statement
            external-reflist)

macro-definition = (MACRO gname (variable) expression external-reflist)

run-form = (RUN expression external-reflist)

v-type = {undefined}

variable-declaration-without-preset = {undefined}
```

1.6.1 Function-with-value

A function-with-value is the mechanism for introducing a function definition to the system. The gname is the name of the function. The v-type is the description of the value produced. The variable-declaration-without-preset is a list specifying various things about the arguments. The expression body is the rule of computation to apply to the arguments. The value of the function is the value of the expression when evaluated.

1.6.2 Procedure

A procedure is a definition of a function that produces no value and therefore is executed for its side effects only. The body of the procedure is any statement. However, some statements have no meaning in this context; go, return, etc. will be diagnosed as errors.

1.6.3 Function-definition

Function-definitions are forms absorbed by the system that state rules of computation. The functions so defined can be called from any place in the system.

1.6.4 Macro-definition

Macro-definitions are functions to be used by the macro-expander at compile time. Their actions will not be specified here.

1.6.5 Run-form

A run-form is the representation of an expression to be operated for its value at the end of the compilation process. The value produced is handed to the supervisory program and probably printed.

1.6.6 Funargs

funarg-definition	=	(FUNARG v-type variable-declaration-without-preset expression external-reflist internal-reflist)
funarg	=	(DATUM functional-type funarg-definition)
external-reflist	=	{undefined}
internal-reflist	=	{undefined}

1.6.7 Syntactic Start Variable

A cril-form is the start variable for the CRIL syntax equations. They are the entities upon which further passes of the compiler act.

1.6.8 Miscellaneous Terms

The terms "identifier", "integer", "lap-address", "type", "v-type", and "atom" have been used in this document without definition, since they are assumed to be understood by those familiar with LISP. A code statement is a mechanism for introducing LAP code into a CRIL-expressed form. The semantics and syntax have not yet been worked out.

1.6.9 Simplification

An additional form in the language is ℓfield.

```
(LFIELD type coordinate address) ≡ (locof (FIELD type coordinate
                                          address))
```

Any place a locof may legitimately appear, the corresponding ℓfield may also appear.

Several CRIL language entities are self-typing, e.g., a field. Therefore

```
(DRIVE T (FIELD T coordinate address))
```

may be reduced to (FIELD T coordinate address) and so on, for driving of other self-typing things. A similar situation exists for callit:

```
(CALLIT T (FIELD S coordinate address))
```

reduces to (FIELD T coordinate address).

2. CONTEXT RESOLVER

2.1 INTRODUCTION

The Context Resolver expands LISP 2 Intermediate Language into a list of CRIL function definitions. Programs in IL format are rewritten as CRIL programs. Error diagnostics are issued when such rewrites are impossible or prohibited by either syntactic or semantic restrictions of IL or CRIL. All legitimate IL programs may be transformed into a computationally equivalent CRIL program. The converse is not true.

Figure 1 shows an overview of the function structure of the Context Resolver. EXPSWITCH is the controller of the recursion and acts as a master switch through which all expansion passes. The functions EXPSTAT, EXPLSA, EXPRSA, EXPARI, EXPRSLA, EXPLSLA, EXPRED, EXPNOVAL, and EXPFNAME are contextual top drivers and form feeders to EXPSWITCH. The switch diverts the compilation through various handlers such as the FUNARG expander, EXPFUNARG, etc. Listed below are the functions in Figure 1, the PUBLIC variables they bind, and a short description of their intended use.

2.2 EXPAND (COMLST, ERRFLG, ERRVAL)

EXPAND is handed either a function or expression in IL format. If an expression, it is changed into a run-form. COMLST is bound to NIL and the function is passed to EXPFUNC for expansion. ERRFLG is checked, and if TRUE, all expansions are scrapped. The value of EXPAND is COMLST, a list of all CRIL expansions for both embedded and embedding functions (not funargs). If the value is NIL, this signifies an error in expansion.

2.3 EXPFUNC (EXTREF, FUNAME)

EXPFUNC performs the expansion of one function. The global reflist EXTREF is bound, and external references are accumulated on it by the declaration logic. EXTREF is tacked on to the CRIL equivalent of the IL function-definition to form the complete CRIL function-definition. The function name is extracted from the IL form before expansion and is bound to FUNAME. The value of the expansion of the function-definition argument is CONS'ed onto COMLST. EXPFUNC has as its value the function name.

2.4 EXPLEX (ALIST, APLIST, FUNTYP, ECLASS, VKIND, RSIDE, LOCNTX, LABDEF, LABREF)

The arguments to EXPLEX are a list of lexically available variables (bound to APLIST) and an IL function-definition to be expanded. ALIST is bound to NIL and FUNTYP is bound to the type of the function to be expanded. The name and type of the functions are composed in CRIL format and placed in the resulting CRIL. The equivalent CRIL form of the IL input is the value of EXPLEX. The context variables are rebound to reflect the value type that is produced by the compiled function. VKIND is bound to NOTERM for procedural compilations (NOVALUE functions).

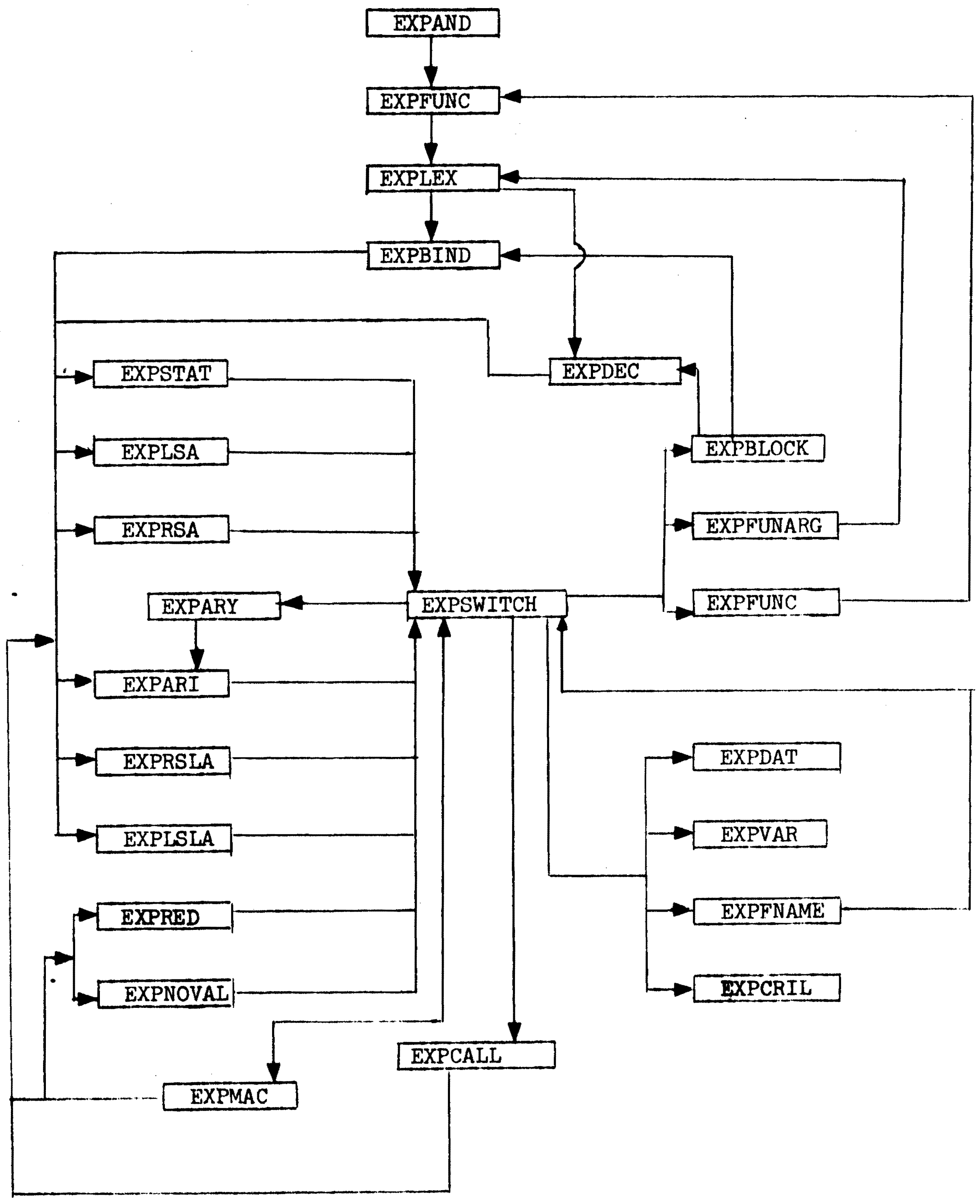

Figure 1. LISP 2 Context Resolver Block Diagram

2.5 EXPBIND (ALIST)

EXPBIND has two arguments: a set of additions to the ALIST, and a functional to continue the expression. The value of the functional (of no argument) is the value of EXPBIND. ALIST is rebound to the APPEND of the addition and the old value of ALIST.

2.6 EXPDEC

EXPDEC has two arguments: a declarative sequence in IL format, and a boolean stating whether or not presets are allowed. The value of EXPDEC is the declaration sequence in CRIL format with the presets (if any) expanded.

2.7 EXPSTAT (ECLASS)

EXPSTAT has one argument, a statement to be expanded. ECLASS is bound to TRUE to indicate that the context of compilation is a statement.

2.8 EXPLSA (ECLASS, VKIND, RSIDE, LOCNTX)

EXPLSA has one argument, an expression to be evaluated in the context of X in the expression (X ← Y). A validity check is made on the expansion of the argument to ascertain legality in this context. The value of EXPLSA is the resultant CRIL form. See Table 1 for values bound to PUBLIC variables during the expansion.

2.9 EXPRSA (ECLASS, VKIND, RSIDE, LOCNTX)

EXPRSA has one argument, an expression to be evaluated in the context of Y in the expression (X ← Y) or (CAR Y). The value of EXPRSA is the resultant CRIL form. (See Table 1.)

2.10 EXPLSLA (ECLASS, VKIND, RSIDE, LOCNTX)

EXPLSLA has one argument, an expression to be evaluated in the context of X in the expression (X → Y). A validity check is made on the expansion of the argument to ascertain legality in this context. (See Table 1.)

2.11 EXPRSLA (ECLASS, VKIND, RSIDE, LOCNTX)

EXPRSLA has one argument, an expression to be evaluated in the context of Y in the expression (X → Y). A validity check is made on the expansion of the argument to ascertain legality in this context. The value of EXPRSLA is the resultant CRIL form. (See Table 1.)

2.12 EXPRED (ECLASS, VKIND, RSIDE, LOCNTX)

EXPRED has one argument, an expression to be compiled in the context of P in the statement (IF P THEN S). The value of EXPRED is the resultant CRIL expansion of its argument. (See Table 1.)

Table 1. Contextual PUBLICS

	ECLASS[*]	VKIND	RSIDE	LOCNTX
EXPSTAT	FALSE	-	-	-
EXPLSA	TRUE	'VALUE	FALSE	FALSE
EXPRSA	TRUE	'VALUE	TRUE	FALSE
EXPLSLA	TRUE	'VALUE	FALSE	TRUE
EXPRSLA	TRUE	'VALUE	TRUE	TRUE
EXPREP	TRUE	'PRED	TRUE	FALSE
EXPNOVAL	TRUE	'NVALUE	TRUE	FALSE
EXPARI	TRUE	'ARITH	TRUE	FALSE
EXPFNAME	TRUE	'FNAME	TRUE	FALSE

[*] ECLASS, RSIDE, and LOCNTX are PUBLIC BOOLEAN; VKIND is PUBLIC GENERAL.

2.13 EXARI (ECLASS, VKIND, RSIDE, LOCNTX)

EXARI works exactly as does EXPRSA, with the exception that VKIND is bound to reflect the expansion of a form used in an arithmetic expression. (See Table 1.)

2.14 EXPNOVAL (ECLASS, VKIND, RSIDE, LOCNTX)

EXPNOVAL has one argument, an expression to be compiled in the context of X in the expression (DRIVE NOVALUE X). A validity check is made to assure legality of the CRIL expansion of the argument in this context. The expansion is the value of EXPNOVAL. (See Table 1.)

2.15 EXPSWITCH (EXP)

EXPSWITCH has one argument, an S-expression representation of an IL form, variable or constant. An IL form may contain embedded CRIL-formatted things but not conversely. If the argument is a variable or constant, an appropriate function changes it to its CRIL counterpart. Otherwise, the form name is processed and--depending on the kind of name--further action is taken by various handler functions. The argument is bound to EXP. If the argument is not a variable or datum, it is checked for being a "top-level true-list."

2.16 EXPDAT

EXPDAT has one argument, a data constant. The value of EXPDAT is the constant in CRIL.

2.17 EXPVAR

EXPVAR has one argument, a variable in IL format. The value of EXPVAR is the field expansion of the variable properly tailed. The expression takes note of the context of compilation variables, LOCNTX, etc.

2.18 EXPFNAME

EXPFNAME handles the expansion of form operators, and as such functions much like a context top driver. Appropriate error checks are made. A communication mechanism with EXPSWITCH is yet to be worked out.

2.19 EXPCRIL

EXPCRIL is a "fiction" used to illustrate the expansion. Its implication is that an argument to EXPSWITCH already translated to CRIL will cause no further transformation.

2.20 EXPCALL

EXPCALL expands function(al) calls. The function(al) declaration is used to indicate which of the contextual top drivers to use for expansion of the arguments of the called function(al). The arguments, before expansion, are wrapped with appropriate combinations of drives and locof's. The value of EXPCALL is the CRIL expansion of the function(al) call.

2.21 EXPARY

EXPARY has one argument, a subscripted array. The subscripts are expanded and combined using field forms. This conglomeration is the value of EXPARY. The context variables, ECLASS, etc. are examined to determine whether the value or a locator is requested.

2.22 EXPMAC

EXPMAC has one argument, a form whose form-name is a macro. The macro is invoked with the form as an argument. The resultant S-expression is fed to EXPSWITCH. This is done because the result of the macro may be an IL form and the expansion must continue until CRIL is produced.

2.23 EXPFUNARG AND EXPFUNC

These two functions are really just macros. They are shown separately in Figure 1 because of the nature of the recursion needed to expand them.

2.24 EXPBLOCK (LABDEF, LABREF)

The expansion of blocks is handled by EXPBLOCK. The ALIST binding and expansion is handled by EXPBIND. The validity of label references and definitions is handled by EXPBLOCK. If the block is an expression, LABDEF is rebound to (()), or else is rebound to the old value of LABDEF with a () CONS'ed on the front. For expression blocks, LABREF is rebound to (()), otherwise to the old value of LABREF with () CONS'ed on the front. When the block expansion is finished, label references in the CAR of LABREF are pruned if they occur in the CAR of LABDEF. If the expansion is of an expression and LABREF is not completely deleted, an undefined labels message is issued. If the expansion is not of an expression, the CAR of LABREF (after pruning) is appended to the CADR of LABREF. The functions adding labels to LABDEF check for ambiguous label definitions. This process will do the only label error checking during the entire process of compilation.

2.25 PROGRAMMING CONVENTIONS AND TECHNIQUES

1. All functions shown in Figure 1, including macros that produce CRIL-formatted output must guarantee completely unique list structure as their value. This is because subsequent compiler passes will change (RPLACA and RPLACD) CRIL input in any manner desired.

2. All CRIL form names will be tailed into section CRIL.. This means that CRIL. is a reserved section name in the LISP 2 system.

3. It is the Context Resolver's task (by use of contextual information) to decide whether special macros are to be used or equivalent functional definitions are to be provided. For example, the Context Resolver decides whether MINUS is the open-coded version or the GENERAL FUNCTION with GENERAL argument.

4. The following function will be available as a diagnostic aid:

```
(FUNCTION (LENCHK BOOLEAN) ((NUMARG INTEGER))

   (IF (EQ (LENGTH (CDR EXP)) NUMARG) FALSE

       (BLOCK ()

          (COMER2 (CAR EXP)

             (QUOTE(WRONG NUMBER OF ARGUMENTS)))

          (SET ERRVAL (BSTGES))

          (RETURN TRUE))))
```

5. The function SIMCHK of one GENERAL argument, a pattern for SIM to match against EXP, will work in the same way as LENCHK using SIM instead of LENGTH for the checking.

6. A function named BSTGES will be coded. BSTGES looks at the context variables and trys to make a guess at a CRIL expression which may be used. This facility is used in attempting to continue expansion when an input is incorrectly formatted.

7. A function named COMERR will exist and will have the following job:

   1. set ERRFLG to TRUE;

   2. send its one argument, a compilation diagnostic, to the supervisor error file.

8. COMER2 has the following definition:

```
(FUNCTION COMER2 (X Y)

   (COMERR (CONS X Y)))
```

9. Whenever it makes sense, the names generated by macros (systems type) must be tailed.

2.26 EXAMPLE MACROS

```
MACRO LABEL (X) :
   IF LENCHK(2) THEN ERRVAL
      ELSE '(DO . LISP) . CDR(X);
MACRO PLUS (X) : ('+ . 'CRIL.) . MAPCAR(CDR (X), COMARI);
```

Table 2. Context Resolver Public Variables

Name	Type	Description
ALIST	GENERAL	List of bound lexical variables and their declarations that are visible at this point in the expansion.
APLIST	GENERAL	List in the same format as ALIST; the variables on this list are the lexicals available from an embedding function.
COMLST	GENERAL	List of CRIL outputs of expansion of all embedding and embedded function definitions.
ECLASS	BOOLEAN	Context variable; TRUE if expression expansion; FALSE if statement.
ERRFLG	BOOLEAN	Set TRUE whenever an error diagnostic is issued.
ERRVAL	GENERAL	Used by error mechanism to hold guess at an acceptable value with which to continue expansion.
EXP	GENERAL	The expression (statement) being expanded.
EXTREF	GENERAL	List of global variables and functions referenced by this compilation.
FUNAME	GENERAL	The name of the function being expanded.
FUNTYP	GENERAL	The type of the function being expanded.
INTREF	GENERAL	List of lexical variables referenced and on the APLIST.
LABDEF	GENERAL	A list of lists of labels visible at this stage of the compilation.
LABREF	GENERAL	A list of referenced labels, definitions for which have not been found.
LOCNTX	BOOLEAN	Context variable that is TRUE if the expansion is for the left or right side of a locset.
RSIDE	BOOLEAN	Context variable that is TRUE if the expansion is for the right side of any assignment (X qualifies in +X).
VKIND	GENERAL	Context variable that states the kind of value expected from an expansion. Possibilities are VALUE, PRED, ARITH, NVALUE, NOTERM, and FNAME.