TM-5455/000/00
(DRAFT)

# CRISP: A PROGRAMMING LANGUAGE AND SYSTEM

31 DECEMBER 1974

J. A. BARNETT
D. L. PINTAR

THIS REPORT WAS PRODUCED BY SDC IN PERFORMANCE OF CONTRACT DAHC15-73-C-0080, ARPA ORDER NO. 2254, PROGRAM CODE NUMBER 5D30.



System Development Corporation
2500 Colorado Avenue • Santa Monica, California 90406

# TABLE OF CONTENTS

## List of Figures and Tables

# INTRODUCTION

CRISP is a programming language and system that will operate on an IBM 370/145 under VM. The goal is to provide an efficient implementation of and experimentation with programs that contain a mixture of numeric calculation and symbol manipulation such as speech and vision systems. The design of the language and system has been most strongly influenced by LISP 2, PL/I, SIMULA, and SDC and BBN LISPs.

The major system characteristics of CRISP are:

- Automatic memory management - garbage collection,
- An incremental compiler and assembler,
- Interactive and batch supervisors,
- System aids to better utilize virtual memory resources,
- Efficient arithmetic and symbol manipulation, and
- Aids to groups constructing large systems.

Some important language features are:

- Three language levels available to the user:
  (1) Source Language (SL) - an ALGOL-like language with infix operators.
  (2) Intermediate Language (IL) - a LISP-like, list structured, Polish prefix language.
  (3) CRISP Assembler Language (CAP) - a macro assembler.
- Block structuring in the normal LISP tradition.
- Standard LISP special variables in addition to local and own variables.
- Full availability of multi-dimensional arrays, ntuples with repeating elements, functionals, and process handles.
- Provision for giving global items first and last names,

thereby allowing name pooling and controlled access.

- Synonyms, both local and global.
- Availability of a wide variety of language extension capabilities.
- General ability to save, restore, and switch contexts of evaluation.

This document has been written concurrently with the design of the language and system. Therefore, it contains some inconsistencies. However, we are releasing it in its present state so that your comments and suggestions can be included in later revisions. It will serve eventually as the basis for the users' guide and system maintenance manual. Please send your critique to Jeff Barnett, System Development Corporation, 2500 Colorado Avenue, Santa Monica, California 90406, or to ARPAnet mailbox JAB at SDC-LAB.

The rest of this document is organized into two sections: The first section is the Language Description and the second section is the System Description.

## LANGUAGE DESCRIPTION

The first half of this document describes the CRISP language. There are three languages available in which to write CRISP programs: Source Language (SL) is an ALGOL-like language with infix and prefix operators. Intermediate Language (IL) is a Polish prefix, LISP-like language. CAP is an assembly language for the IBM 370 series computer; it features automatic stack manipulation and block structuring. The descriptions of SL and IL are presented together. However, all examples, unless otherwise stated, are given in SL. SL is always translated to IL before compilation. Therefore, a knowledge of IL is helpful to those users who employ macros and other compile-time substitution forms.

CRISP is a "syntaxy" language. That is, there are many forms with specialized formats and therefore many keywords. This may be contrasted with pure LISP and APL, where only a dozen or so forms are provided and everything else is constructed from these forms. The reasons for not taking that approach are many. Perhaps the most important is that CRISP is designed as a tool for the construction of very large programs by groups of programmers. In this context, there is no virtue in the ability to write one liners; you can't build very much of the program that way in any event. Also it is imperative that the system be able to compile high performance code. This task is simplified when forms properly signal the user's intent. For instance, it is much

easier to compile a good loop when it is introduced by the word FOR than when the same intent must be discovered in a LISP recursive nest. (Even special handling of MAPCAR does not help with nested loops.) Further, keywords tend to enhance the readability of programs, particularly to someone other than the original author. In the SDC LISP system, users have a choice between a form that resembles SL (called INFIX LISP) and the normal LISP. Almost without exception, when given the choice, programmers prefer INFIX. This is particularly true for large, complicated functions or for any calculation involving arithmetic. In CRISP, the choice is still available; if you are not content without the mother parenthesis, then write in IL.

Two language features have been planned but are not yet fully designed. Therefore, they are not described in this document. The first feature is embedded function and processor definitions. This would not give static scoping but would allow own variables to be shared. The other feature is a finite state machine (FSM) primitive to allow pattern-matching matching tc regular sequences of any data type.

The sections in this part of the document are: Syntax Specification Language; External Data Formats; Scoping and Denotation Rules; Declarations, Definitions, and Types; Blocks; Data Primitives and Presets; Expressions; Conditionals; FOR Loop; Processors and Processes; and The CAP Assembler.

# SYNTAX SPECIFICATION LANGUAGE

The syntax specification language used in this document to describe CRISP is standard BNF with the usual augmentation -- namely, operators that specify repeated occurrences. The terminal symbols in BNF are keywords and meta names. Keywords are normal symbols -- for example, BEGIN, +, and A. Examples of meta names are <block>, <expression>, and <function-def>. Meta names are enclosed in angle brackets. If a meta name consists of a sequence of more than one word, then the meta name has conventionally been concatenated by hyphens rather than spaces. Meta names are the names of syntax rules. For instance, this is a rule:

<a>::=<b><c>

<a> is defined as equivalent to a <b> followed by a <c>. Given this definition, the following two rules have the same meaning:

<x>::=<y><a><z> and <x>::=<y><b><c><z>

The general form of a rule is a meta name followed by ::=. This means that the meta name is defined as equivalent to the right side; the part following the ::=. Right sides are sequences of pattern parts. In operation, a rule is said to match (or generate) an input string if and only if each pattern part matches the input in the order in which the parts occur in the rule and if the matched parts are contiguous and do not overlap. The kinds of pattern parts are:

meta-variables - match the pattern defined by this meta name;

key-word - match exactly this keyword in the input;

{p1 ... pn} - match this sequence of pattern parts;

p1| ... |pn - match any pi, 1≤i≤n;

[p1 ... pn] - match this sequence of pattern parts or nothing e.g., the sequence is optional;

$p - match zero or more occurrences of p;

$=p1=p2 - match zero or more occurrences of p2 separated by p1s;

#p - match one or more occurrences of p;

#=p1=p2 - match one or more occurrences of p2 separated by p1s;

where p and pi are pattern parts. In the rules, concatenation of a sequence of pattern parts takes precedence over (binds more tightly than) alternation. Therefore, braces -- { } -- are often used to overcome the normal interpretation in much the same way in which parentheses are used in arithmetic expressions. Some examples of pattern parts and input strings that they match are:

| | |
|---|---|
| {abc} | abc |
| a\| b\| c | a b c |
| [abc] | nothing abc |
| $a | nothing a aa aaa etc. |
| $=+=a | nothing a a+a a+a+a etc. |
| #a | a aa aaa etc. |
| #=+=a | a a+a a+a+a etc. |

The characters, "{", "}", "|", "[", "]", "$", "=", ":", ">", "<", "#" and "=", have special meaning in the syntax specification language. If any of them is used as a keyword in CRISP itself, then its occurrence in a rule is underlined to avoid ambiguity.

There are two basic sets of syntax rules: rules for the syntax of tokens (described in the section on external data formats, page 16) and rules for the syntax of data structures and programs. The token syntax defines the members of the token classes such as <integer>, <identifier>, and <global-name>. The keywords in the token syntax are characters. The token syntax rules do not allow any implicit spaces between characters. In a few instances, liberties have been taken to improve the readability of the rules. For example, the pattern part, <character¬'>, means any <character> except prime.

The key words in the rules that describe data structures and programs are token class names and special symbols such as BEGIN, FOR, and +. Between the contiguous occurrence of any two tokens, there may be spaces. The delimitation rules given with the token syntax specify when this is necessary to avoid ambiguity. In any event, spaces are always legal between tokens.

## Character and Token Syntax

The data read from a symbolic input file (such as the user terminal) is a sequence of characters. The character sequence is normally segmented into a token sequence. The syntax of CRISP programs and the external format of data structures is described in terms of tokens. Examples of token classes are integer and identifier.

A character is an 8-bit byte viewed as an EBCDIC code. Because many character codes are not available on certain I/O devices, a special character mechanism is provided to allow all characters to be entered by their explicit codes.

<character>::=<special-character>|<regular-character>
<special-character>::=%%<hex-digit><hex-digit>
<hex-digit>::=<digit>|a ... f|A ...F
<digit>::=1 ... 9
<regular-character>::=any EBCDIC character

A four-byte <special-character> is translated by the character-reading program into the EBCDIC character with the specified code. In all subsequent processing of the input character sequence, no distinction is made between a regular character and its equivalent special character. Thus, if ij were the hex code for %, then %%ij%xy is equivalent to %%xy,

which in turn would be equivalent to the character whose EBCDIC code is xy.

The character sequence is segmented into the corresponding token sequence by the token parsing strategy. The strategy has two parts: (1) determine the boundaries of a token, and (2) convert the character sequence comprising the token into the appropriate internal format or representation.

The first part of the strategy applies the delimitation rules. The functions READTOKU and READTOK are used to implement the token strategy. (READTOKU parses signs as separate tokens from succeeding numbers.) They start at the next character position and scan until the token is delimited. If certain tokens are next to each other, determining the ending of the first and the beginning of the second might be ambiguous. In these cases, the tokens must be separated by a character string called a <spacer>. A <spacer> is not a token, and the characters making up the <spacer> are lost in the conversion of characters to tokens.

<spacer>::=<blanks>[<comment>][<spacer>]
<blanks>::=¢<blank>
<blank>::=a space
<comment>::= %'${<character¬'>|''}'

The next step performed by the token functions is the conversion of the character string into the internal representation. The major token categories and their

corresponding internal representations are: nil - NIL, fixed point - integer, floating point - float, string - string, global name - name pointer, delimiter - identifier, identifier - identifier, escape - identifier.

The function READTOKU uses the following token syntax equations for parsing the input stream.

<token>::=<nilt>|<unsigned-integer>|<unsigned-floating>|
<string>|<global-name>|<delimiter>|<identifier>|
<escape>

<nilt>::=NIL

<unsigned-integer>::=<decimal>|<hex>

<decimal>::=#<digit>

<hex>::=<digit>$<hex-digit>{x|X}

<unsigned-floating>::=<mantissa>[<characteristic>]

<mantissa>::=#<digit>.$<digit>

<characteristic>::={e|E}<integer>

<string>::='$'{<character¬'>|''}'

<global-name>::=<identifier>$<identifier>

<delimiter>::= ↓ | " | # | ( | ) | = | ' | @ | [ | ] | + |
- | : | * | : | ≤ | ≥ | , | ¬ | _ | ↓ | ↓

<identifier>::=<special-id>|<regular-id>

<special-id>::=$'$'{<character¬'>|''}'

<regular-id>::=<character¬{<delimiter>|<digit>|<blank>|$}>
$<character¬{<delimiter>|<blank>|$}>

<escape>::=%

The function READTOK uses the above set with the following substitutions:

<token>::=<nilt>|<integer>|<floating>|<string>|
<global-name>|<delimiter>|<identifier>|
<escape>
<integer>::=[+|-]<unsigned-integer>
<floating>::=[+|-]<unsigned-floating>

Thus, signs in front of numbers belong with the numbers. READTOKU is used by the SL read program and READTOK by nearly everything else. The function READCH is also provided. It inputs the next character and moves the pointer ahead. READCH also causes the next line to be input when the present line is exhausted and another character is requested. The value of READCH is the identifier with the one character name coresponding to the character input. When an end of file condition is encountered, READCH returns NIL. (See the section on the I/O facility, page 214, for more information.)

The following paragraphs briefly describe the individual token classes in more detail. In comments, strings, and special identifiers, a '' stands for a single ' and does not delimit the spelling.

<nilt> is the notation for the token representation of NIL. NIL is not an identifier while $'NIL' is. The character sequence "()" is divided into the two tokens "(" and ")". However, the function READ would treat NIL and () as equivalents.

[Unsigned] integer or floating-point tokens are converted to their equivalent internal integer or floating-point form. In either case, if the number is too large (or too small) to be representable in the computer in full word format, the error message "NUMERIC SIZE ERROR ON INPUT" is given. For hex integers, the X is required. If the leading digit of a hex integer is not a decimal digit, then it must be preceeded by a 0. (Thus, write 0A1X not A1X; the latter would be interpreted as an identifier.) Extra digits in the mantissa of a floating-point number are discarded and do not cause errors.

A string token is a convenient way of entering a one-dimensional character array. If this format is used, there must be fewer than 265 characters in the body of the string. If more are entered, the error message "STRING OVERFLOW ON INPUT" is issued.

A global name specifies the first and last name for a global object such as a global variable, function, etc. The first and last names must conform to the rules for identifiers. If a $ is to be used for purposes other than constructing a global name, it should be separated from a preceding identifier by a <spacer>.

Delimiter tokens are converted into the corresponding identifiers with one-character names. Delimiters are normally used to build SL operators.

Identifier tokens must not be longer than 255 characters. The $' form is provided to represent identifiers with unusual names.

The escape token is provided to allow an input to be aborted. The error message issued is "ESCAPE".

Examples of Tokens

Some examples of members of the various token classes are:

<nilt> - NIL

<integer> - 12, -1407, +2

<floating> - 0.92, 9.0 -13.4E-5, +0.17E+6, 1.0E1

<string> - 'THIS IS A STRING', 'CAN''T BE'

<global-name> - A$B, $'X Y'$$'123', XYZ$AB

<delimiter> - <, 6

<identifier> - HELLO, $'SPECIAL ID'

<escape> - %

## Structure Syntax

The function READ converts an input sequence of tokens into an internal structure representation. The bracketing pair "( )" is used to enclose binary node lists, and the pair "{ }" is used to enclose complex numbers, nodes, arrays or ntuple structures. Thus, the delimiters "(" and "{" are used to introduce special syntax for complex structures. To input them as identifiers, write $'(' and $'{'. Similarly, if ")" or "}" is used other than as a closer (balancer), the

error message "UNBALANCED )" or "UNBALANCED }" will be issued. To enter them as identifiers, write $')' and $'}'. The syntax of external structures is:

<external-data>::=<simple-external>|<composite-external>

<simple-external>::=<nil>|<number>|<global-name>|
<delimiter>|<identifier>

<nil>::=<nilt>|()

<number>::=<integer>|<floating>|<complex>

<complex>::=[COMPLEX {<float>|<integer>}
{<float>|<integer>}]

<composite-external>::=<string>|<array>|<ntuple>|
<list>|<noden>

<list>::=(#<external-data>[ . <external-data>])

<noden>::=[NODEi #<external-data>]

<ntuple>::=[<ntuple-type>$<external-data>]

<array>::=[<array-type>$<external-data>]

From the above, it is obvious that structures in several kinds of data spaces cannot be input by the READ function. The excluded spaces are: pdp, pdn, bps, handle, and heap. The meaning of pointers into these spaces (and hence the addressed structures) has a highly implementation-dependent usage. Therefore, their external and internal manipulation is the responsibility of the knowledgeable user or system programmer.

The external format for a list structure is the standard LISP syntax. The optional dot ending signifies a CONSed

pair. Otherwise, the final CDR is NIL.

The <ntuple-type> is any declared ntuple type. The external data following the type must be in one to one correspondence with the ntuple items. For repeating groups, the last (inner) subscript is varied most rapidly. The input items must be convertible to the specified item type. If not, the error message "INCORRECT NTUPLE ITEM TYPE ON INPUT" will be issued. If the ntuple type is unknown to the system, the message issued will be "INCORRECT NTUPLE TYPE ON INPUT".

The <array-type> may be any legal type of array including one with flattened items. If not, the message "INCORRECT ARRAY TYPE ON INPUT" will be issued. The specified dimensions must be positive and less than 32768. (The * option may not be used.) Otherwise, the error message "DIMENSION OUT OF BOUNDS ON INPUT" will appear. The external data following the dimension must be in one to one correspondence to the array elements. Last subscript varies most rapidly, etc. If fewer than the specified number of elements appear, then the missing elements are initialized to the standard default values. If extra elements appear, the error message issued is "TOO MANY ELEMENTS ON INPUT". If an element is not convertible to the required type, the error message "ILLEGAL ELEMENT ON INPUT" will be given.

Examples of Structures

The following are some examples of the format of structures that have multi-token printing representations.

```
{INTEGER ARRAY(2,3) 1 2 3 4 5 6}

{INTEGER ARRAY(*,*) ARRAY(2)
  {INTEGER ARRAY(2,3) 1 2 3 4 5 6}
  {INTEGER ARRAY(4,2) 1 2 3 4 5 6 7 8}}

(A B . C) and {NODE2 A {NODE2 B C}} are the same

{COMPLEX -4.7 18.1E-12}

{NODE4 17.5 NIL XYZ 14}

(THIS IS A LIST)
```

With the declaration, DEC A$B<X INT, Y FLOAT>;

```
{A$B -17 1.5}

{A$B 0 -8.E4}
```

See section on declarations, definitions, and types (page 45) for more information.

# SCOPING AND DENOTATION RULES

The tokens that make up a CRISP program are names. This section describes the rules that assign meaning to these names -- that is, the rules for deciding what object a name denotes. There are three major categories of names: constants, identifiers, and global names. As a category, identifiers include delimiters (normally used as operators), syntax keywords, and others.

## Structure of the Rules

Determining the correspondence of a name to an object depends upon lexical context, dynamic execution state, and the default tailing mechanism. Lexical context is determined by the program's block structure and the mode of name usage. The rules operate in two parts: scoping rules that are used at compile time, and denotation rules that operate at execution time.

The scoping rules convert a name to a proper name. The proper name of a constant is the data object to which it refers -- that is, itself. The proper name of a local object is an identifier, and the proper name of a global object is an ordered pair of identifiers separated by a $ (defined as a <global-name> in the section on external data formats, page 16). The scoping rules use lexical context

and the default tailing mechanism to assign proper names.

The denotation rules pair a proper name as determined by the scoping rules with an object. The denotation rules operate at execution time and make use of the dynamic state. Of course, the denotation of a constant is equally well determined at compile time by the scoping rules.

## Constants

Any data object may be used as a constant. However, the form of a datum may conflict syntactically with an evaluatable program part. As an example, the use of the identifier X could be interpreted as a variable name. For those cases where interpretation is ambiguous, the compiler assumes that the object is not a constant. To force the compiler to treat an object as a constant, the quote mechanism is provided.

*SL*

```
<constant>::=<unambiguous-data-object>|'<external-data>

<unambiguous-data-object>::=<nil>|<string>|<array>|<ntuple>|
                            <noden>|<unsigned-integer>|
                            <unsigned-floating>|<complex>
```

*IL*

```
<constant>::=<unambiguous-data-object>|
             (QUOTE <external-data>)

<unambiguous-data-object>::=<nil>|<string>|<array>|<ntuple>|
                            <noden>|<integer>|<floating>|
                            <complex>
```

For example, in SL, to use the identifier X as a constant, write "X. In IL, write (QUOTE X). Other examples of constants in SL are:

12, 8.2E-6, {INTEGER ARRAY(3) -17 18 -19}, "HELLO,
'THIS IS A CONSTANT STRING', {COMPLEX 3 -4.5}, NIL,
"(SAMPLE LIST), "{NODE2 SAMPLE {NODE2 LIST NIL}},
(), {NODE3 1 2 3}, "{NODE3 1 2 3}

Note, that whether a binary node (node2) constant is entered as a list or as a noden construct, it is ambiguous in a program definition and therefore must be quoted.

## Kinds of Objects

The kinds of objects that are referenced with non-constant names are synonyms, macros, transforms, functions, processors, generators, data spaces, code places (labels), and variables' bindings and values. For much of the discussion below, it is assumed that the appropriate substitutions have been made for synonyms and by macros and transforms.

In CRISP, there are no label-valued variables. For scoping and denotation resolution, labels are treated as if they were local names bound in the outermost block in which they are visible. One exception is that branching out of an expression is illegal. For example, in

```
BEGIN;
 L;A:=B;
   C:=BEGIN;
         IF X THEN GOTO L;
         RETURN 5;
      END;
END
```

the form "GOTO L" is illegal because the inner block (in which the GOTO resides) is used as an expression. See the section on statements and labels (page 115) for more information.

In CRISP, the object denoted by a variable name is called a binding. A binding is a proper name paired with a data object called its value. Several variable objects (bindings) may have the same name and/or the same value. The rules for determining which binding is referenced by a name are described in the following paragraphs. The legal operations with bindings are: creation (called binding), destruction (called unbinding), and retrieving or changing the value (called referencing and setting, respectively).

## Local and Global Names

A local name always refers to a variable binding whose place of creation may be determined by lexical inspection. That is, it is not possible to reference a variable object with a local name outside the function, processor, block, etc. that binds it. Thus local means local to a definition and a nested set of blocks within that definition.

A global name may denote a variable binding whose place of creation cannot be determined by lexical inspection. In general, the binding referenced by a global name may be created in the function containing the reference, another function, or even another process as described below.

## The Default Tailing Mechanism

The default tailing mechanism directs the compiler in assigning global proper names to identifiers that reference global objects. The transform, DEFAULT, and the function, DEFAULTX, are provided. Each specifies a default tail (last name) for identifiers being declared, defined, or bound. Also, an ordered set of possible tails to be used with an identifier appearing in reference mode (operator, left side of an assignment form, or as an expression for value) is given by DEFAULT and DEFAULTX.

```
*SL*
<default-form>::=<default>|<defaultx>
<default>::=DEFAULT <identifier>($*,*<identifier>)
<defaultx>::=DEFAULTX(<expression>,<expression>)

*IL*
<default-form>::=<default>|<defaultx>
<default>::=(DEFAULT <identifier> ($<identifier>))
<defaultx>::=(DEFAULTX <expression> <expression>)
```

The function, DEFAULTX, has two arguments. The value of the

first is the default tail and must be an identifier. The second argument is the ordered default list and must be a list of identifiers. The value of DEFAULTX is its first argument. When executed, DEFAULTX informs the compiler of the new default information that is to be used until changed by another usage of DEFAULTX.

DEFAULT is a transform that merely quotes its two arguments and generates a call on DEFAULTX. Thus, the following two SL forms are equivalent:

```
DEFAULT XYZ(XYZ,CRISP)
DEFAULTX("XYZ,"(XYZ CRISP))
```

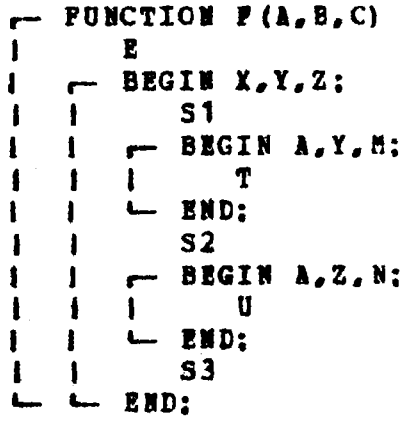
More information on the use of the default tail and the ordered default tailing list is given below in the section on scoping rules (page 32).

The system is initalized with the form

```
DEFAULT USER(USER,CRISP)
```

## Lexical Nesting

CRISP programs are made up of function, processor, and space definitions, variable declarations, and the compile time substitution mechanism (synonyms, macros, generators, and transforms). This subsection describes the nested structure of function, processor, macro, and generator definitions. To simplify the discussion, all such definitions will herein be called function definitions.




LEXICAL NESTING

Figure A

The outer level of the lexical nest is the function definition. The inner levels are blocks. Each level of the nest may bind variables. Figure A is an example. The function, F, binds the variables (parameters) A, B, and C. The body of F contains the block that binds the variables X, Y, and Z. Within this block are two other blocks: the first binds the variables A, Y, and M, and the second binds the variables A, Z, and N. The lines to the left of the figure show the (lexical) scope of the function definition and the blocks. S1, S2, and S3 are three groups of statements in the outer block that are not in any inner block. T is the group of statements in the first inner block, and U is the group of statements in the second inner block. E is a part of the body of F not in any block.

At any point in the definition of a function, there is a

lexical nest of the function definition and some blocks that properly contain that point. From these, a lexically derived search list may be formed. It consists of the name of the function, the function's parameters, and the block variables bound by the nest. This list is derived using the most nested bindings first. For example, in the part of F marked E, the lexically derived search list is:

C,B,A,F:

at the points called S1, S2, and S3:

Z,Y,X,C,B,A,F:

at the point called T:

M,Y,A,Z,Y,X,C,B,A,F:

and at the point called U:

M,Z,A,Z,Y,X,C,B,A,F.

The next subsection, on scoping rules, describes how the proper names put on this list are derived and how this list is used in assigning proper names to identifiers appearing in definitions. The lexical lists for the above example were constructed (for the sake of illustration) without regard for proper names.

## Scoping Rules

This paragraph details the scoping rules for transforming a name to a proper name. For constants and global names (identifier pairs separated by a $), the transformation is trivial: in both cases, the proper name and the original are identical. The remaining case, identifiers used as names,

is more complicated.

Determination of an identifier's proper name depends upon lexical context (nesting and mode of usage) and the default tailing mechanism. If the mode of usage equals declaration or definition, then the identifier is paired with the default tail to produce the global proper name. (Mode of usage equals declaration means that the name is an object whose attributes are being given by a declare form; mode of usage equals definition means that the name identifies a function, processor, macro, etc., that is being compiled.)

When an identifier is used in binding mode (parameter of a function, processor, macro, etc., or as a block variable), an optional scope attribute may be specified by the program writer. The attribute may be LOCAL or GLOBAL. (For use of OWN scope see the subsection on own variables, page 44). If the scope attribute is LOCAL or not specified, then the proper name is the identifier itself, a local name. Otherwise, if the value is GLOBAL, then the proper name is the identifier paired with the default tail.

The remaining case is an identifier used in reference mode, e.g. as an operator, on the left side of an assignment form or referenced as an expression for value. First, the lexically bound search list is examined to see if it contains any member whose name (if local) or whose first name (if global) is the identifier. If such an element exists, then the proper name of the first one found is used

as the proper name of the identifier. (Remember that the list is built in reverse order, most deeply nested bindings first: see preceding subsection.)

The rules above supply the proper name for an identifier that is bound lexically. If the name is not determined by this search, then the ordered default list of identifiers given by DEFAULTX is consulted. The identifier is paired with the first identifier in the ordered list as its last name (or tail). If a declaration or definition exists for a global object with that paired name, then the pair becomes the proper name of the identifier. If not, the next identifier in the list is tried as the tail. This procedure is repeated until a proper name is determined or the list is exhausted.

It has not yet been completely decided what to do when the above fails to produce a proper name for an identifier used in reference mode. The choices are: (1) give an error message, (2) guess a local declaration and force a binding *a la* FORTRAN, (3) guess a global declaration and use a global proper name *a la* LISP, and (4) ask the user. In any event, conflict in type between declaration of a name and its manner of use may produce compile or run time errors and diagnostics. See the section on declarations, definitions, and types (page 45) for more discussion.

Figure 8 shows a set of forms before and after the operation of the scoping rules. The first set of forms are in a

Before Scoping Rules

```
DEFAULT SYS(SYS,CRISP);

DECLARE GENERAL A, GENERAL B$SYS;

FUNCTION SUBST1(LOCAL X)
  WHEN B=X       THEN A
  WHEN NODEP(X)  THEN SUBST1(CAR(X))#SUBST1(CDR(X))
                 ELSE X;

FUNCTION SUBST$CRISP(GLOBAL A, B$SYS, C)
                        SUBST1(C);
```

After Scoping Rules

```
DEFAULT SYS(SYS,CRISP);

DECLARE GENERAL A$SYS, GENERAL B$SYS;

FUNCTION SUBST1$SYS(X)
  WHEN B$SYS=X THEN A$SYS
  WHEN NODEP$CRISP(X)
       THEN SUBST1$SYS(CAR$CRISP(X))#
            SUBST1$SYS(CDR$CRISP(X))
       ELSE X;

FUNCTION SUBST$CRISP(A$SYS, B$SYS, C)
                      SUBST1$SYS(C);
```

Figure B

EXAMPLE OF SCOPING RULES

format that might very well be written by the programmer; the second set has all names except those that function as special language names (DEFAULT, DECLARE, parentheses, comma, etc.) converted to the appropriate proper names. In the function SUBST1, the lexically bound search list is:

X, SUBST1$SYS

and in the function SUBST, the lexically bound search list is:

C, B$SYS, A$SYS, SUBST$CRISP

Both these lists are built from proper names.

## Dynamic Context

The dynamic state of execution is determined by an ordered set of process states and a set of top-level objects. Each variable with a global name has a top-level binding (and associated value) that is visible whenever the variable has not been explicitly bound. Global names of objects, other than variables, always refer to a single object that is called the name's top-level or its only value. The global proper names of functions, processors, macros, transforms, generators, spaces and global synonyms may not be bound. Therefore, these global names are like constants in that the object they denote may be unambiguously determined at compile time. Further, all global names, variable or other, always have exactly one top-level denotation.

As a process executes, functions and blocks are entered in

some order and exited in inverse order. On entry, variables (function parameters and block variables) are bound. That is, bindings for the bound variables are created and added to the front of a list called the process's variable context. On exit, the bindings added by the function or block are removed (unbound) from the process's variable context.

Consider the operation of

SUBST(1,2,"(1 (3 2) 4))

The definition of SUBST is given in Figure B (page 35). When SUBST is entered, the process's variable context is augmented by:

[C,(1 (2 3) 4)][B$SYS,2][A$SYS,1]

The square brackets denote a binding (which consists of a variable's proper name and a data object called the value.) If we trace the action of SUBST and SUBST1 to the point where 2 is passed as the argument to SUBST1, the total augmentation to the process's variable context will be

[X,2][X,(2)][X,(3 2)][X,((3 2) 4)][X,(1 (2 3) 4)]

[C,(1 (2 3) 4)][B$SYS,2][A$SYS,1]

Notice that this list is built in inverse order in much the same way as the lexically bound search list.

Associated with each process is another process (or the top-level set of objects) called its parent context. If process 1 is the parent of process 2, then we say that process 2 is embedded in process 1. The set of processes in the system at any moment form a tree with the processes

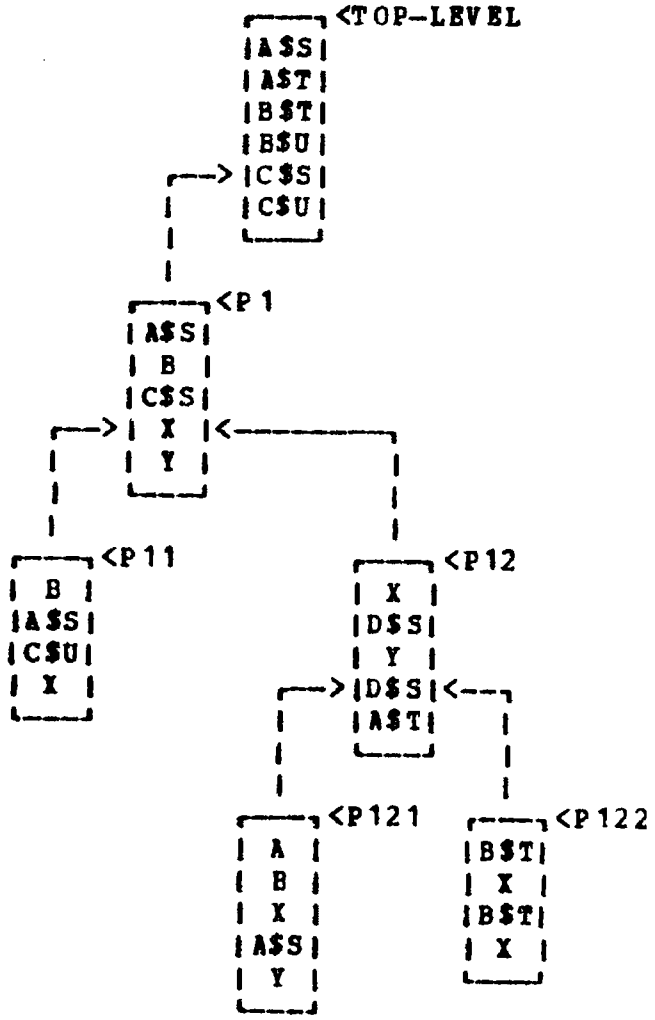



Figure C

PROCESS TREE
VARIABLE BINDINGS

being the nodes and leaves, the relation embedded-in (or parent-of) forming the arcs, and the top-level collection of objects forming the root node. Figure C shows such a configuration for the processes P1, P11, P12, P121, and P122. The boxes contain the processes' variable contexts.

The total variable context of a process is the concatenation of the (ordered) variable contexts of itself, its parent, its parent's parent, etc., up to and including the top-level set of objects. (Since there is no name duplication among the top-level objects, their ordering is immaterial.) From Figure C, the processes and their total variable contexts are:

P1 Y, X, C$S, B, A$S, C$U, C$S, B$U, B$T, A$T, A$S

P11 X, C$U, A$S, B, Y, X, C$S, B, A$S, C$U, C$S, B$U, B$T, A$T, A$S

P12 A$T, C$S, Y, C$S, X, Y, X, C$S, B, A$S, C$U, C$S, B$U, B$T, A$T, A$S

P121 Y, A$S, X, B, A, A$T, C$S, Y, C$S, X, Y, X, C$S, B, A$S, C$U, C$S, B$U, B$T, A$T, A$S

P122 X, B$T, X, B$T, A$T, C$S, Y, C$S, X, Y, X, C$S, B, A$S, C$U, C$S, B$U, B$T, A$T, A$S

A process may be moved from being embedded in some particular parent to being embedded in another parent so long as the result still forms a tree (not a forest and no loops). To ensure that the concept of a process's total variable context is well defined, one of the following two conditions is necessary: (1) only one process may execute at a given moment, or (2) the following operations are indivisible: binding, unbinding, referencing, setting, and

process embedding. Though the present system design satisfies the first criterion (because there are no explicit provisions for parallelism), the implementation will also satisfy the second criterion in style and spirit, so as to not exclude future possibilities.

## Denotation Rules

The denotation rules operate at execution time, pairing proper names (as determined by the scoping rules) with the appropriate data objects. The denotation of all names except variables may be determined at compile time by the scoping rules as described in previous subsections.

A variable appearing in binding mode (parameter or block variable) causes binding (adding of a name-value pair to a process's variable context) upon entry to a function or block and unbinding upon exit. The other possible use of a variable is in reference mode. In this case, the proper name of the variable is used to search the total variable context of the process containing the reference, for the first binding whose name part is the proper name of the variable. Depending on usage of the reference, either a copy of the value part is retrieved or the value part is set. The rules of the CRISP language guarantee that every binding contains a value part. Thus, there is no such run time diagnostic as "reference to unbound variable."

The following table shows the binding objects referenced by proper variable names when appearing in the processes as shown in Figure C.

| Variable | Referenced in Process | | | | |
|---|---|---|---|---|---|
| | P1 | P11 | P12 | P121 | P122 |
| A | ¬ | ¬ | ¬ | P121 | ¬ |
| B | P1 | P11 | ¬ | P121 | ¬ |
| X | P1 | P11 | P12 | P121 | ¬ |
| Y | P1 | ¬ | P12 | P121 | P122 |
| A$S | P1 | P11 | P1 | P121 | P1 |
| A$T | tl | tl | P12 | P12 | P12 |
| B$T | tl | tl | tl | tl | P122 |
| B$U | tl | tl | tl | tl | tl |
| C$S | P1 | P1 | P12 | P12 | P12 |
| C$U | tl | P11 | tl | tl | tl |

Process names refer to last binding of the appropriate name.
tl - top level object
¬ - local variables do not reference name outside of process containing reference.

Scoping and denotation strategies of the kind contained in this section are often called "dynamic scoping rules."

## Compile Time Substitutions

To enhance readability and compactness of representation in CRISP programs, an extensive compile time substitution mechanism is included in the system. A transform substitutes the forms used as its arguments for its parameters' names in its body. The transformation is done at compile time and the result is compiled in place of the original form. For example, DEFAULT may be coded as:

```
TRANSFORM DEFAULT(A,B) DEFAULTX("A,"B);
```

The parameter values to a transform are IL forms. They are substituted into the IL form of the body. The substitutions are done in "parallel" so that the actual values of the parameters do not cause strange effects. Thus,

```
DEFAULT X(A,Y) means
DEFAULTX("X,"(A Y)) not
DEFAULTX("X,"(X Y)).
```

Transforms substitute through all node2 structures including quotes.

Macros also perform compile time substitutions. A macro is a function of one argument, an IL form that has the macro's name as its form operator. The macro is called at compile time, and its value is used in place of the original form. To write DEFAULT as a macro,

```
MACRO DEFAULT(X)
      LIST("DEFAULTX,LIST("QUOTE,CADR(X)),
                     LIST("QUOTE,CADDR(X)));
```

A generator is used as a "bottom up" macro for forms that require special handling. Examples are IF, BEGIN, etc. A description of the operations of generators will appear in the document, CRISP Compiler and Assembler Structure.

Macros, transforms, and generators are global objects. They are used at compile time whenever their proper name, as determined by the scoping rules, appears as a form operator. Synonyms, on the other hand, may be local or global. At compile time, whenever the scoping rules produce a proper

name that is the name of a synonym, the value of the synonym is immediately substituted for the occurrence of the name. For example,

```
BEGIN SYN S:=CAR(FOO(X,Y)),X;
   A:=S+COS(S)
END
```

is equivalent to

```
BEGIN X;
   A:=CAR(FOO(X,Y))+COS(CAR(FOO(X,Y)));
END
```

Notice, a synonym merely does a substitution; it does not remember the context of its definition, and its use does not inhibit multiple evaluation. (A variable setting does all this.) Thus, if FOO has side effects, the two applications of FOO in the example may produce different values.

Local synonyms have the same visibility (scope) as a local variable appearing at the same spot. Global synonyms, introduced by a declare form, have the same visibility as function, macro, etc., defined at the same spot. Synonyms may be used anywhere in reference mode unless specifically stated otherwise. However, synonyms may not be used as substitutions for names appearing in declaration, definition, or binding mode. Also, synonyms may not be used as substitutes for keyword names that have syntactic significance or where they introduce ambiguity in translating SL to IL. All synonym substitution is performed after translation to IL.

## Own Variables

An own variable mechanism is provided as a convenience for programming certain kinds of algorithms in which communication is needed between different calls on the algorithm. An own variable has only one binding; that is, it may not be rebound. The declaration of an own is made when a block variable is given the scope attribute, own. The variable is visible only in places where a local variable appearing in the same spot would be visible: the block where the declaration occurs and nested blocks that do not rebind another variable with the same (first) name. At compile time, a preset value is computed. Entering the block during execution has no automatic effect on the variable's value. However, the operation of statements within the block may change its value. This use of own more closely resembles the PL/I STATIC attribute than the OWN of ALGOL.

An example use of an own variable is a program that generates random numbers. A seed has some preset value. Each time the generator is used, the seed receives a new value so that the generator will not return the same thing every time. The following is an example random number generator that uses an own:

```
FLOAT FUNCTION RANDOM()
       BEGIN INTEGER OWN SEED:=1;
        SEED:=(SEED*65536)&7FFFFFFFX;
        RETURN SEED*2.0**-31
        END;
```

# DECLARATIONS, DEFINITIONS, AND TYPES

This section describes the uses of data types in CRISP. The first subsection defines and explains the motivation for the inclusion of types in the language. Other subsections describe declare forms, definitions, item referencing, processing of declarations, and determination of expression types.

Types

A (data) type is a collection of objects. The value of a name with a type attribute is restricted to a member of that collection. The order of a type is the number of objects in that collection. The order may be indefinitely large, such as for the integers, or very small, such as for the boolean collection that contains only two objects. In most instances, the order of a type is somehow restricted by the size of the computer system or the organization of its memory -- 32-bit word, 8-bit byte, etc.

Most programming systems not only allow type attributes for names but require them (maybe implicitly like FORTRAN's first letter convention). Languages such as LISP without name attributes are the exception. They are sometimes incorrectly called "typeless" languages. But if that is correct, what is the meaning of such predicates as ATOM, NUMBERP, etc? In fact, these languages should be called "attributeless" languages. The advantages of giving type attributes to names are numerous. Some advantages are: (1) improved efficiency because the compiler knows more about the situation, (2) possibility of compile as well as run time error checking, (3) commentary -- improves readability of programs, (4) coercion -- automatic type conversion, (5) resolution of ambiguity, and (6) improved representation of abstractions.

An object may be atomic or composite. An atomic object is

not decomposable into elements by conventional techniques such as subscripting or name qualification. Said another way, a field in a composite data object may be changed by a simple assignment. For example, in most systems a floating point object is atomic even though its sign, characteristic, and mantissa are computable. The distinction between atomic and composite objects is not sharp and depends in a large measure upon the programming language and its set of primitives. A composite object is composed of elements that may be individually referenced (and/or set) by conventional techniques. It is normal that all composite elements of each object in a type collection have corresponding elements of each object that belong to a type collection associated with that element.

CRISP and most other programming languages use a data structure template as the method of describing the set of objects that belong to a type. A set of basic types are given, and new types are defined as composite structures whose element types are basic types or other user defined types. An example of a basic type is an integer, and a defined type would be an integer array or even an array of integer arrays, etc. The basic types provided are not necessarily atomic (e.g., nodes). However, when this is the case the motivation may be efficiency or some other aspect of implementation. On the other hand, almost no language allows the definition of new atomic types. Further, the kinds of composite objects that are definable is usually restricted. In CRISP, the only definable object structures

are ntuples and arrays.

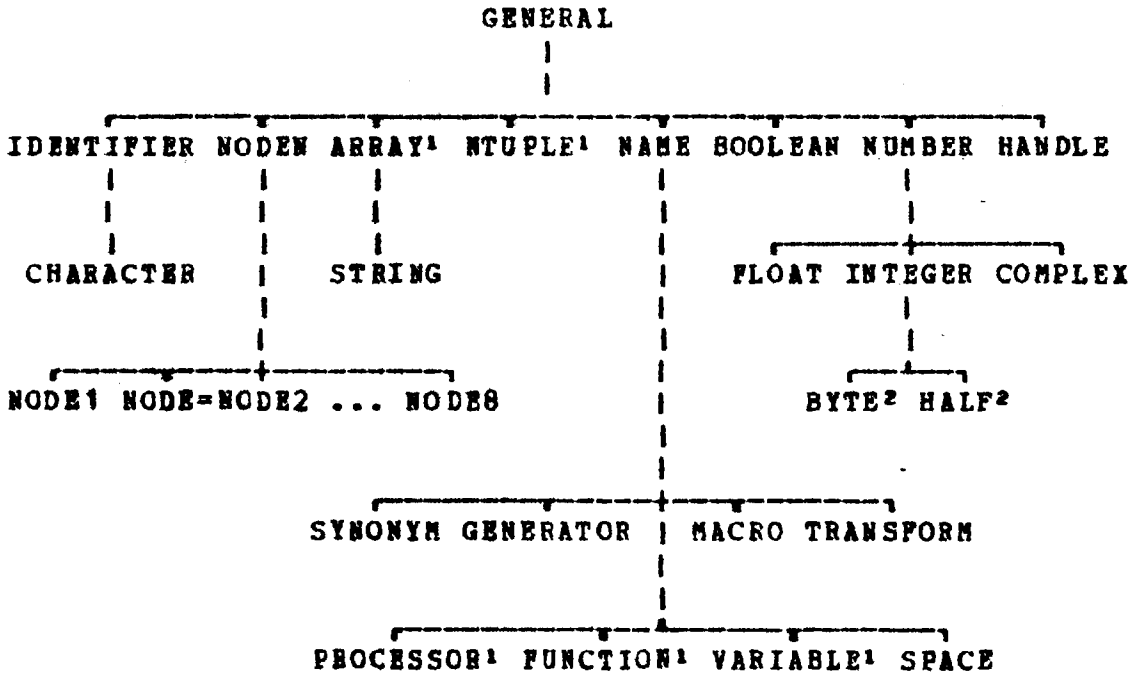
A set of type collections form a partial ordering (hierarchy) under the relationship of proper containment. For example, the collection of integer arrays is contained in the collection of all arrays. Figure D shows the type hierarchy available in CRISP. An object in a lower type is always convertible to a higher type without losing its identity (except for byte and half -- see below). That is, an integer being "kept" under the type general would still be detected by both the predicates INTEGERP and NUMBERP. (Corresponding to each type shown in Figure D is a predicate function that is the type name followed by the letter "P"). Downward conversion is not always possible. For instance, a name is not convertible to both a function and a variable. Conversion of one subtype of number to another is often done. However, these conversions lose the original identity of the object. Thus,

```
INTEGERP(FLOAT(4))
```

is false. (This does not mean that 4 has lost its identity as an integer. Rather, the same value of FLOAT could just as well be derived from FLOAT(4.0).) The following paragraphs briefly describe the individual type classes shown in Figure D.

Identifier types

Identifier objects are local names and the pieces used to build global names. The type, character, contains the 256 identifiers with one-character names. Identifier objects




TYPE HIERARCHY

Figure D

[1] Must be subspecified, e.g. INTEGER ARRAY(*).

[2] May only appear as element type in arrays and ntuples.

are unique. That is, there are never two separate identifier objects with the same printing representation. Associated with each identifier is a system property object and a user property object. The system property object is a link structure used by the compiler and the assembler to locate global names that have the identifier as their first name. The user property object is provided so that the programmer can build an associative memory using identifiers as search keys, as in LISP. The individual characters in an identifier's print name are not settable or referenceable as they would be in a string. When an ntuple or array element is declared of type character, storage is saved by storing only the 8-bit (one-byte) EBCDIC equivalent. (Identifier objects are stored as 32-bit pointers to the unique object.)

Name types

The type, name, consists of global names. The subtypes are: function, processor, (global) variable, macro, transform, generator, (global) synonym, and space. The types function, processor, and variable are further subspecified, i.e.,

INTEGER VARIABLE - variable with integer value

INTEGER FUNCTION(FLOAT) - integer valued function with one floating argument

PROCESSOR(NAME) - processor with one argument of type name.

Variables, function values, and array and ntuple elements may have a type attribute of name or a subspecified type of function, processor, or variable. When used as type attributes, the words FUNC, PROC, and VARB are used for

FUNCTION, PROCESSOR, and VARIABLE to avoid syntactic ambiguity. Thus, the type attribute of a variable with values of the type

INTEGER FUNCTION(FLOAT)

is written

INTEGER FUNC(FLOAT).

This differentiates a name that is a variable with functional values from a function name (which is for most intents and purposes a constant).

Boolean type

The type, boolean, contains only two objects, NIL and the identifier TRUE. When objects are converted to boolean, everything except NIL is changed to TRUE. When boolean is used as an ntuple or an array element type, an 8-bit field is used to conserve storage. Even though boolean could be considered as a subtype of identifier, it would be so unnatural that it is not.

Handle type

A member of the type, handle, is a process. That is, a processor that has been put into operation along with its control state and variable context.

Number types

The type, number, contains the three subtypes integer, float, and complex. Integer and floating objects are 32-bit quantities in the standard IBM 370 format. Complex objects are pairs of floating point numbers. (It should be noted

that complex objects are provided only as a user convenience and are handled very inefficiently. All computations that have complex arguments or produce complex values make function calls.) When an object is converted from one of number's subtypes to another, it loses its original identity. Thus, the value of

```
INTEGERP(FLOAT(4))
```

is false just the same as

```
INTEGERP(FLOAT(4.0)).
```

However, the conversion of an integer object to a number or a general object preserves the original type classification as an integer. (Similarly with float and complex.)

The integer subtypes,[1] byte and half, are provided for efficient storage of small quantities as elements of ntuples and arrays. A byte object is an integer in the range 0 through 255, and a half object is an integer in the range -32768 through 32767. When a byte or half element is accessed, it is immediately converted to integer, and its original identity is lost. Thus, there are no such predicates as BYTEP or HALFP.

Noden types

The type, noden, is made up of the eight subtypes node1, node2, node3, node4, node5, node6, node7, and node8. The

---

[1] These integer subtypes are really a combination of the type attribute, integer, and a precision attribute. Since the only precision control in CRISP is with integer elements and then to only full, half or quarter words, liberty has been taken.

type, node, is exactly equivalent to node2, the binary tree node of standard LISP. The type, nodei, is like an ntuple with i elements of type general. The elements are named FIRST, SECOND, THIRD, FOURTH, FIFTH, SIXTH, SEVENTH and EIGHTH. The fields of node2 objects may also be referenced with CAR - CDR primitives. The fields are referenced and set as if the noden object were an ntuple object with the above ordinals as item names. However, NTUPLEP of a noden object is false.

Array types

Array objects are subtyped by their number of dimensions and the type of their elements; for instance, INTEGER ARRAY(*) is a one dimensional integer array. In CRISP, the extents of an array's dimensions are not part of the array's type. All arrays carry dimensioning information with them. A string is a one dimensional array whose elements are characters.

Ntuple types

An ntuple is an ordered collection of elements. Each element has a type attribute, a name, and a repeat count associated with it. Ntuple subtypes are defined through the declare form.

General type

The type, general, is the collection of all objects in the CRISP system. The value of a variable with type attribute general may be any data object in the system.

Type predicates

For each type shown in Figure D (page 49) (except byte and half) there is a type predicate that returns the boolean value TRUE if its argument is of the specified type and returns NIL otherwise. The predicate names are formed by appending the letter "P" to the type name. The predicates are: IDENTIFIERP, CHARACTERP, NAMEP, FUNCTIONP, PROCESSORP, VARIABLEP, MACROP, TRANSFORMP, GENERATORP, SYNONYMP, SPACEP, BOOLEANP, HANDLEP, NODENP, NODE1P, NODE2P, NODEP, NODE3P, NODE4P, NODE5P, NODE6P, NODE7P, NODE8P, NTUPLEP, NUMBERP, INTEGERP, FLOATP, COMPLEXP, ARRAYP and STRINGP. A universal type predicate, TYPEP, is also provided. TYPEP has two arguments, a type and an expression. TYPEP returns TRUE if its argument is of the specified type and NIL otherwise.

*SL*

<typep>::=TYPEP(<type-ref>,<expression>)

*IL*

<typep>::=(TYPEP <type-ref> <expression>)

A <type-ref> is any specifiable type including an ntuple or a subspecified func, proc, varb, or array. See the subsection on the declare form (page 59) for formal definition of <type-ref>. The following two forms are equivalent:

NUMBERP(X) and

TYPEP(NUMBER,X).

## Data Object Formats

In the system, objects are represented by one, two, or four byte numbers. One and two byte numbers may only appear as elements of arrays and ntuples to represent byte, boolean, character and half objects. In all other cases, objects are represented by four-byte (32-bit) numbers in one of three formats: (1) 32-bit integer, (2) 32-bit floating, and (3) 32-bit pointer with high order 8 bits unused. A pointer is the byte address of the object. Except for integer, float, byte, half, and character or boolean (when used as an array or ntuple element) types, objects are implemented as pointers. When an integer (float) object is converted to number or general, a copy of the integer (float) is put in a special integer (float) space and a pointer at it is the "value". Conversion of an object from one pointer format to another (assuming the conversion is legal) is an identity transformation. Upward conversion (see Figure D, page 49) from one pointer type to another is always legal. Downward conversion may or may not be legal. For instance, a general pointer at a macro can be downward converted to a name but not to an array. In error checking mode, all downward conversions are diagnosed for possible type mismatches.

The object, NIL, may arise in any pointer type. For example:

```
DECLARE ILIST<X INTEGER, Y ILIST>
```

This declares ILIST to be a two element ntuple whose first element, X, is an integer. The second element, Y, is a pointer either at an object in the same ntuple subtype,

ILIST, or at the object NIL. If NIL were not allowed, then ILIST would have to be a looping structure (some kind of ring) and could not simply be a list of integers. However,

TYPEP(ILIST,NIL)

is false. In general, the object NIL appearing in a pointer type indicates either the "terminal condition" of recursion or partial initalization. When these cases can arise, the user's program should check for them. Inline structure access code produced by the compiler does not. (To do so would cause an unreasonable expansion in the size of the generated code.) The object, NIL, is represented by a pointer at address 0.

Type determination

This paragraph is a slight digression from a CRISP description. The question investigated is whether the type of a structure should be determined a priori by tagging or a posteriori by inspection. With a priori typing, a structure receives its identity (type) when it is created and carries that information with it as long as it exists; even when the value of an element is set. With a posteriori typing, it is assumed that (1) the type of an atomic object may be determined by inspection, (2) the elements of a composite structure may be distinguished (from each other), and (3) therefore some (total) algorithm exists that determines whether or not a particular object belongs to a particular type collection. The basic result is that if (1) recursive definitions of type collections are allowed and (2) the values of elements of composite objects can be set (for

instance to form rings, etc.), then a posteriori type determination is impossible.

The first question to be addressed is whether recursive type declarations are necessary or advantageous. Consider the following, which defines a list of integers:

INT.LIST=[INTEGER,INT.LIST|NIL]

Surely such definitions as this ought to be allowed by any general mechanism. The second capability, the ability to change the value of an element, is also necessary. Pure LISP is an example of a language that does not allow this (no RPLACA or RPLACD), but no one writes programs in pure LISP except to prove a point or to use as an example for a program correctness technique. All languages used for serious programming efforts allow element setting.

Now consider the above definition of an INT.LIST. It says that (with a posteriori typing) an object is an INT.LIST if and only if it is a composite structure with two elements, the first of which is an integer and the second of which is an INT.LIST or NIL. Let x be a two element composite structure whose first element is an integer, say 6, and whose second element is (a pointer at) the structure x itself. Then the a posteriori type predicate could determine that x is an INT.LIST if and only if x is an INT.LIST. Thus, it would be consistent to say either yes or no. Obviously, in this situation a convention could be adopted, probably to say yes. However, let us alter this example slightly:

XYZ=[ INTEGER,¬XYZ ]

This says that a structure is an XYZ if and only if it is a two element composite structure and the first element is an integer and the second element is not in the XYZ collection. Now consider the two element structure described above: the first element is the integer 6 and the second element is the structure x itself. Now the a posteriori type predicate can deduce that x is an XYZ if and only if x is not an XYZ.

To trace the steps that brought about this dilemma: (1) a recursive definition was allowed, (2) a "blank" two element structure was created (and named x), (3) the first element of x was set to the integer 6, and (4) the second element of x was set to x itself. (These operations are in no way different from creating a one element ring.) After these operations, we attempted to determine type membership.

Compare this to the a priori typing scheme using the same example: (1) make the definition of XYZ as above, (2) create a blank two element structure (named x) with type identification XYZ, (3) set the first element of x to the integer 6, and (4) attempt to set the second element of x to itself. An error is immediately detectable. Thus, the above contradiction is never generated.

## The Declare Form

The <declare> form is one of two major methods available to give global names type attributes. The other method is the definition mechanism described in the next subsection. The declare form creates top-level objects with the specified name and type attributes. For variables, the top-level value is generated as part of the binding. Data spaces are declared using the function NEWSPACE (page 293).

Many problems arise when the type attribute of a name is changed. Compiled code and data may already reference the name and make assumptions about the structure of the object associated with that name. If the name is given a new type attribute, the assumptions may no longer be correct and may lead to unrecoverable errors. The solution to this problem that has been adopted is described below in the paragraph on redeclarations. The rest of this section assumes that the declare form is not doing any redeclarations. See the section on data presets (page 127) for handling of initial value assignment to declared variables.

### Implicit typing

In many circumstances, the type attribute of a name is determined implicitly by examining the first character of the name in a manner similar to FORTRAN. The <implicit-form> details the relation between the first character of a name and its implicit type. The syntax of an <implicit-form> is:

```
*SL*
<implicit-form>::=IMPLICIT $[<imp-type>($=,=<imp-range>)]
<imp-type>::=GENERAL|INTEGER|FLOAT
<imp-range>::=<character>[THRU <character>]

*IL*
<implicit-form>::=(IMPLICIT $(<imp-type> $<imp-range>))
<imp-type>::=GENERAL|INTEGER|FLOAT
<imp-range>::=<character>|(<character><character>)
```

For example, to establish the normal FORTRAN conventions (where names can begin only with A-Z), use the following:

```
IMPLICIT FLOAT(A THRU H, O THRU Z)
         INTEGER(I THRU N);
```

When an implicit form is executed, the default type of all characters is initialized to general. Then the sub-phrases are interpreted in left to right order. A range of a single character makes the implicit type of that character the specified <imp-type>. A two character range makes the implicit type of all charcters whose EBCDIC code falls in the inclusive range the specified type. Thus, these two forms are equivalent:

```
IMPLICIT FLOAT(I THRU I);  and  IMPLICIT FLOAT(I);
```

Also, the system is initalized with either of the following two exactly equivalent forms that set the implicit type of all names to general:

```
IMPLICIT;  and  IMPLICIT GENERAL(%%00 THRU %%FF);
```

An implicit form may be used only at the top level; it may

not be embedded in other forms. The function

GENERAL FUNCTION IMPLICIT(ID,CHAR,CHAR)

is available for dynamic use. Its first argument is an <imp-type>, and its other arguments are the character extrema of the range. The value is NIL.

## Syntax of declares and types

A <declare> form gives type attributes to global names. The most common usages are to declare variables and synonyms. In addition, the <declare> form may be used to give type attributes to function and processor names. This is sometimes necessary when forward references are made to functions or processors that are not defined in the same file as the references. See the section on disk compiling (page 259) for more information. A declare form must appear on the top level; it may not be embedded in any other form. The syntax of the <declare> form and types is:

```
*SL*
<declare>::=DECLARE $e,*<declaration>
<declaration>::=<synonym-dec>|<like-dec>|<variable-dec>|
                <function-dec>|<processor-dec>
<synonym-dec>::=<syn-dec>|<synx-dec>
<syn-dec>::=SYN <name>:=<form>
<synx-dec>::=SYNX <name>:=<expression>
<like-dec>::=<name> LIKE <name>
<function-dec>::=[<value-type>] FUNCTION <name>
                                         <arg-type-list>
<value-type>::=NOVALUE|<type-ref>
```

```
<processor-dec>::=PROCESSOR <name><arg-type-list>

<arg-type-list>::=($*,*<type-ref>)|
                  ($[<type-ref>,]<type-ref> INDEF)|
                  ($[<type-ref>,]<type-ref> LIST)

<variable-dec>::=<var-dec>|<ntuple-dec>

<var-dec>::=[ <type-ref> ][ <global-scope> ][ VARIABLE ]
                          <name>[ <preset> ]

<ntuple-dec>::=[ NTUPLE ][ <global-scope> ][ VARIABLE ]<name>
                {<item-rep-count>[ <item-type> ]|<group-def>}

<name>::=<identifier>|<global-name>

<global-scope>::=GLOBAL

<preset>::= :=(<expression>|*)

<item-rep-count>::=(#*,* [[ <integer>/]<integer>])

<item-type>::=<type-ref>|<flat-type>|
               <group-def>|<short-type>

<group-def>::=≤#*,* (<item-name>[ <item-rep-count> ]
                    [ <item-type> ])≥

<item-name>::=<identifier>

<flat-type>::=FLAT <named-type>

<named-type>::=<name>

<short-type>::=BYTE|HALF

<type-ref>::=<named-type>|<array-type>|<simple-type>

<array-type>::=<element-type> ARRAY <array-rep-count>|
               STRING

<array-rep-count>::=($*,* [[ <integer>/]<integer>|*])

<element-type>::=<type-ref>|<flat-type>|<short-type>

<simple-type>::=<general-type>|<identifier-type>|
                <node#-type>|<name-type>|
                <boolean-type>|<number-type>|
                <handle-type>|<composite-type>

<general-type>::=GENERAL

<identifier-type>::=IDENTIFIER|CHARACTER

<node#-type>::=NODE#|NODE1|NODE2|NODE#|NODE3|
               NODE4|NODE5|NODE6|NODE7|NODE8

<name-type>::=NAME|SPACE|<proc-type>|
```

```
                <func-type>|<varb-type>

<proc-type>::=PROC <arg-type-list>

<func-type>::=<type-ref> FUNC <arg-type-list>

<varb-type>::=<type-ref> VARB

<boolean-type>::=BOOLEAN

<number-type>::=NUMBER|INTEGER|FLOAT|COMPLEX

<handle-type>::=HANDLE

<composite-type>::=ARRAY|NTUPLE


*IL*

<declare>::=(DECLARE $<declaration>)

<declaration>::=<synonym-dec>|<like-dec>|<variable-dec>|
                <function-dec>|<processor-dec>

<synonym-dec>::=<syn-dec>|<synx-dec>

<syn-dec>::=(<name> SYN <form>)

<synx-dec>::=(<name> SYNX <expression>)

<like-dec>::=(<name> LIKE <name>)

<function-dec>::=(<name> FUNCTION [<value-type>]
                        <arg-type-list>)

<value-type>::=NOVALUE|<type-ref>

<processor-type>::=(<name> PROCESSOR <arg-type-list>)

<arg-type-list>::=($<type-ref>
                   [(INDEF <type-ref>)|
                    (LIST <type-ref>)])

<variable-dec>::=<var-dec>|<ntuple-dec>

<var-dec>::=<name>|(<name>|[VARIABLE][<global-scope>]
                    [<type-ref>][<preset>])

<ntuple-dec>::=(<name>[VARIABLE][<global-scope>]
                {<item-rep-count>[<item-type>]|<group-def>}
                [<preset>])

<name>::=<identifier>|<global-name>

<global-scope>::=GLOBAL

<preset>::=(SET {<expression>|*})
```

```
<item-rep-count>::=(REP #{<integer>|(<integer><integer>)})

<item-type>::=<type-ref>|<flat>|
              <group-def>|<short-type>

<group-def>::=(GROUP #[<item-name>|
                       (<item-name>[ <item-rep-count>]
                        [<item-type>])])

<item-name>::=<identifier>

<flat-type>::=(FLAT <name-type>)

<named-type>::=<name>

<short-type>::=BYTE|HALF

<type-ref>::=<named-typed>|<array-type>|<simple-type>

<array-type>::=(ARRAY <array-rep-count><element-type>)|
               STRING

<array-rep-count>::=(REP $[<integer>|(<integer><integer>|*])

<element-type>::=<type-ref>|<flat-type>|<short-type>

<simple-type>::=<general-type>|<identifier-type>|
                <noden-type>|<name-type>|
                <boolean-type>|<number-type>|
                <handle-type>|<composite-type>|
                <func-type>|<proc-type>|
                <varb-type>

<general-type>::=GENERAL

<identifier-type>::=IDENTIFIER|CHARACTER

<noden-type>::=NODEN|NODE1|NODE2|NODE|NODE3|
               NODE4|NODE5|NODE6|NODE7|NODE8

<name-type>::=NAME|SPACE|<proc-type>|
              <func-type>|<varb-type>

<proc-type>::=(PROC <arg-type-list>)

<func-type>::=(FUNC <value-type><arg-type-list>)

<varb-type>::=(VARB <type-ref>)

<boolean-type>::=BOOLEAN

<number-type>::=NUMBER|INTEGER|FLOAT|COMPLEX

<handle-type>::HANDLE

<composite-type>::=ARRAY|NTUPLE
```

Declaration examples

By several examples, simple usage of the <declare> form will be demonstrated. Following that, several paragraphs will describe the syntax and usage in more detail. In most instances, shortened type names such as INT (for INTEGER) will be used. Also, the shortened name, DEC, will be used for DECLARE. See Appendix IV (page 317) for a complete list of legal abbreviations. Also see the section on scoping and denotation rules (page 25) for a description of the naming conventions. For this section, it is assumed that the <default-form>,

```
DEFAULT USER(USER,CRISP);
```

is in effect. Therefore, all names that are not explicitly tailed (identifiers) will be automatically tailed with the last name, USER. The default <implicit-form>,

```
IMPLICIT;
```

is in effect. Therefore, all names without an explicit type attribute are general.

Example 1:

```
DEC INT A, GEN B, FLOAT C:=17;
```

Three global variables, A, B, and C, are declared of type integer, general, and float, respectively. A receives an initial value of 0, B receives an initial value of NIL, and C receives an initial value of 17.0 (forced conversion).

Example 2:

```
DEC GLOBAL A, VARIABLE B, GLOBAL VARIABLE C;
```

The three variables, A, B, and C, are all declared of type

general with a preset of NIL. The scope, GLOBAL, and the class name, VARIABLE, are redundant but may be used.

Example 3:

```
DEC INT ARRAY(*,*) A, A ARRAY(*) B;
```

In this example, the global variable, A, is declared as a two dimensional integer array. The variable, B, is declared as a one dimensional array whose elements are the same type as A. That is, B is a one dimensional array, each of whose elements is a two dimensional integer array.

Example 4:

```
DEC INT ARRAY(*,*) ARRAY(*) B;
```

B has the same declaration as in example 3 -- namely, a one dimensional array whose elements are two dimensional integer arrays.

Example 5:

```
DEC SYN X:=A+B*C;
```

The expression form, "A+B*C" will be substituted for subsequent occurrences of the proper name X$USER.

Example 6:

```
DEC FLOAT FUNC(FLOAT) TRIGF:=COS;
```

TRIGF is declared as a variable whose value is a functional that returns a floating value and receives a floating argument. The preset is the function, COS.

Synonym declarations

A <synonym-dec> defines a global name as a synonym. An example is 5 above. In the following, a <synx-dec> form is used:

```
DEC SYNX X:=A+B*C;
```

The keyword, SYNX, specifies that the value of the expression "A+B*C" should be immediately computed (at compile time). The value is the form that will be substituted for appearances of X. Recall that synonym substitution works on IL forms (or SL forms after translation to IL). The following two pairs of forms are equivalent:

```
DEC SYN X:=A+B*C;
DEC SYNX X:="(PLUS A (TIMES B C));

DEC SYN PI:=3.14159;
DEC SYNX PI:=3.14159;
```

Like declarations

A <like-dec> specifies that the first name should be declared with exactly the same attributes as the second name. Thus:

```
DEC X LIKE COS;
```

means that X is the name of a function that receives a floating argument and returns a floating value. This should be contrasted to the use of a named type:

```
DEC COS X;
```

In this case, X is declared as a global variable whose type attribute is:

```
FLOAT FUNC(FLOAT)
```

This would be the case when one wished to write such forms as:

```
X:=COS or X:=SIN
```

etc. The second name in the like declaration (the type "sender") must be a variable, a function, or a processor; otherwise, an error diagnostic will be issued.

Function and Processor Decs

A <function-dec> and a <processor-dec> give a type attribute to a name as a subspecified function or processor, respectively. In a <function-dec> declaration, the optional <type-ref>, if present, specifies the type of value returned by the function. If not present, then the value is determined as the implicit type of the name. Thus,

```
DEC FUNCTION X(), INT FUNCTION Y();
```

declares X to be a function of no arguments that returns a general value and Y to be a function of no arguments that returns an integer value. A function that returns no value (like a FORTRAN subroutine) may be declared with a <value-type> of NOVALUE, in which case the function may be called only as a statement or in other places where a value is not needed. Since a processor has no value, none is declared.

An <arg-type-list> gives the type of each argument to a function or processor. An <arg-type-list> may specify that there is an indefinite number of arguments. For example:

```
DEC FLOAT FUNCTION Z(GEN,INT INDEF);
```

The function, Z, is declared to return a floating value. Its first argument is of type general, and there are zero or more arguments of type integer. See the section on

definitions (page 97) for more information on indef arguments.

Type refs

A <type-ref> is the major syntax mechanism for specifying type attributes in CRISP. There are three kinds of <type-ref>s: (1) <simple-type>, (2) <named-type>, and (3) <array-type>. The simple types are shown in Figure D (page 49). A <named-type> is a name. The type of that name is "borrowed". If a <type-ref> name is a global name, it is used as is. If it is an identifier, then the standard scoping rules are used to turn it into a proper name. The most common use of a <type-ref> is to borrow a subspecified ntuple type.

Some conversion is performed in this type borrowing if the name is a function, procesor, or variable. By example:

```
DEC INT X, FLOAT FUNCTION Y(), PROCESSOR Z(GEN);
DEC X A, Y B, Z C;
```

A, B, and C are variables with types borrowed from X, Y, and Z, respectively. The second declare form is equivalent to:

```
DEC INT A, FLOAT FUNC() B, PROC(GEN) C;
```

That is, the type attribute of a variable is borrowed, not the type of its name. Borrowed function and processor types are converted to the corresponding func and proc types. See the <like-dec> form for the other possibility (page 67). In the above, B is a variable whose values are (the names of) functions of no arguments that return float values. The type derived from a <type-ref> must be a legitimate simple

type, array type, or a subspecified ntuple type. If the <type-ref> name is any of the excluded subcategories of name (macro, generator, transform, synonym, space), then the <type-ref> type will be converted to name.

<type-ref>s try to substitute immediately. Thus, in

```
DEC INT X, X Y, Y Z;
```

X, Y, and Z are integer variables. Y loses its identity as an "X", and Z looses its identity as a "Y". Therefore, a change in the declaration of X does not affect Y or Z and a change in the declaration of Y does not affect Z. The exception to this is when a subspecified ntuple type is borrowed. Consider:

```
DEC ILIST<X INT, Y ILIST>, ILIST A, A B;
```

ILIST is a subspecified ntuple whose type is also called ILIST. The variables A and B are also subspecified ntuples of type ILIST. That is, if the value of ILIST, A, or B is printed, the type identification would be ntuple of subtype ILIST (actually ILIST$USER.) A redeclaration of A would have no affect on B. However, a redeclaration of ILIST would affect both A and B. Their type would now be the hidden version of ILIST. See the section on declarations and redeclarations (page 84) for more information.

Assume that the value of the variable V is an ntuple of the subtype ILIST; then the value of all three of the following forms is TRUE:

```
TYPEP(ILIST$USER, V)
TYPEP(A$USER, V)
```

```
TYPEP(B$USER, V)
```

(It is necessary to explicitly tail the names in these forms only in contexts where ILIST, A, or B have been locally bound with other meanings.) The <type-ref>s in the <typep> form also undergo immediate substitution by the compiler. Array types and their uses in <type-ref>s are described in the following paragraph.

Array types

An <array-type> defines the type of the array's elements and the number of dimensions. An array declaration may or may not include information on the extents of the individual dimensions. However, this information is not considered a part of the array's type; it is used only for allocation. When not specified, the lowest subscript value is assumed to be the value of the variable LOWSUB$CRISP when the type definition is compiled. The system is initalized with

```
LOWSUB:=1
```

in effect. Thus,

```
DEC INT ARRAY(*) A;
```

means that A is a one dimensional integer array whose first element is A[1] and that A has an unspecified number of elements. To specify the actual range of subscripts, use a pair of integers separated by "/".

```
DEC INT ARRAY(0/256,-4/17) A;
```

The first number of the pair is the lowest subscript value; the second is the extent of the subscript (not the highest subscript value). Thus, A is a 256 by 17 integer array. The first subscript ranges between 0 and 255 and the second

between -4 and 12 (both ranges are inclusive). If a single number is used as a dimension, then that number is the extent of the dimension.

```
DEC INT ARRAY(14,17) A;
```

means that A is a 14 by 17 integer array. The first subscript ranges between 1 and 14 inclusive and the second between 1 and 17 inclusive. The three kinds of dimension specification may be mixed in a single declaration.

```
DEC INT ARRAY(0/256,10,*) A;
```

This declares A to be a three dimensional integer array. The first subscript ranges between 0 and 255, the second between 1 and 10, and the third between 1 and the actual extent of the array's third dimension (all ranges inclusive).

As stated previously, the element type of an array can be byte or half. A byte array is an array whose elements are 8-bit unsigned integers, and a half array is an array whose elements are 16-bit (half word) signed integers. Using byte and half arrays saves storage. However, there are penalties: (1) it takes more code (both space and time) to access (reference or set) byte elements, and (2) most operations with half quantities are slower than the corresponding operations with integers. For these reasons, it is suggested that INTEGER arrays be used unless a large space savings will be achieved using the short types, byte and half.

The array subtype, string, is exactly equivalent to

```
CHARACTER ARRAY(*)
```

That is, a string is a one dimensional array whose elements are characters. The individual elements may be referenced and set by the standard subscripting mechanism.

The type of an array's elements may be declared flat. The word, FLAT, has no meaning unless the <named-type> is an ntuple. Thus, these two declare forms are equivalent:

```
DEC INT X, FLAT X ARRAY(*) A;

DEC INT X, X ARRAY(*) A;
```

In both cases, A is a one dimensional integer array. The following two forms are different:

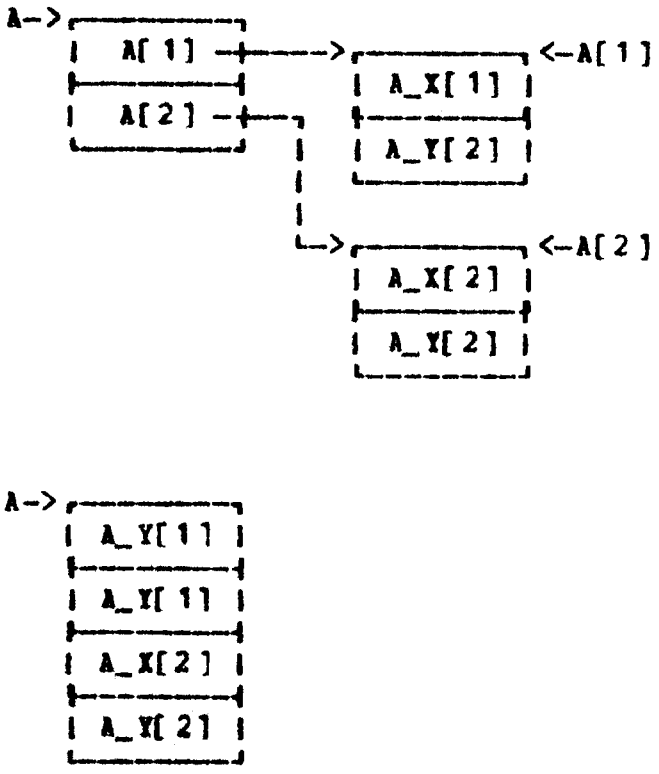
```
DEC FOO<X INT, Y INT>,
    FOO ARRAY(2) A;

DEC FOO<X INT, Y INT>,
    FLAT FOO ARRAY(2) A;
```

Figure E shows the difference. The top diagram corresponds to the first declare form with the unflat elements. The bottom diagram corresponds to the flat element case. The flattened representation obviously conserves storage (8 words vs. 14 words including the array and ntuple headers). However, the unflattened representation allows more general processing. The array elements in the unflattened case may be processed individually. For example,

```
B:=A[1]
```

where B is a variable with type attribute FOO, would make the value of B a pointer at the same structure as A[1]. If A is flat, as in the second diagram, the above assignment would be illegal. A copy of the pair, A_X[1] and A_Y[1] could be made using the ntuple copy primitives. See the




FLAT vs. NON-FLAT ARRAYS
Figure E

section on copiers (page 255) for more details.

Ntuple declarations

An <ntuple-dec> gives a variable name a subspecified ntuple type attribute. Further, a name with such an attribute may be used, as a <type-ref>, to confer the same attribute on another name. Ntuples are composite objects. An object in a particular ntuple subtype has a fixed number of elements. Each element has a name and a type attribute, and the elements occur in a particular, specified order. The name of an element is a combination of item qualifiers (other item names) and subscripts. The formation of an element

name is described in detail in the section on item referencing and subscripts (page 86). (The terms "item"and "element" are used interchangeably in this section and throughout this document.) The term field refers to the occurrence of an element.

Ntuples may be structured in a variety of ways. In the simplest case, an ntuple is merely an ordered tuple with simple elements. For example:

```
DEC N<X INT, Y FLOAT, Z NODE2>;
```

N is declared to be an ntuple with type N. (Actually, the variable, N$USER, is declared to be an ntuple with type attribute N$USER, assuming that the original default form is in effect.) N is an ordered 3-tuple; the first element (named X) is an integer, the second element (named Y) is a float, and the third element (named Z) is a binary node. Example values of the variable N are:

```
{N$USER 27 -13.6E-4 (X Y Z)}
{N$USER -13 1701.2 (14 . 17)}
```

When an ntuple structure of type N is created, the initial value is:

```
{N$USER 0 0.0 NIL}
```

The elements of an ntuple may be repeated. However, the number and extent of the repetitions must be specified when the declaration is compiled. (Therefore, there is no option to use "*" as with array declarations.) For example:

```
DEC N<X INT, Y(3) FLOAT, Z NODE2>;
```

N is declared to be a 5-tuple. The elements are X, Y[1], Y[2], Y[3], AND Z, where X is an integer, Y[1] - Y[3] are

floats, and Z is a binary node. Note, Y itself may not be independently referenced. An example value of N is:

```
{N$USER 1 3.7 -4.5 1.0 (A B)}
```

Repeating element declarations resemble array declarations in that either the extent or the lowest value subscript and the extent of a repetition may be specified. If the lowest value subscript is not specified, then the value is assumed to be the value of the variable, LOWSUB$CRISP, when the declaration is compiled. The initial value of LOWSUB is 1. For example:

```
DEC N<X(0/2,2) INT, Y FLOAT>;
```

N is declared as a 5-tuple with the elements X[0,1], X[0,2], X[1,1], X[1,2], and Y. An example value of N is:

```
{N$USER 1 2 3 4 5.0}
```

Ntuples may also be partitioned into groups. A group is a collection of elements and other groups. For example:

```
DEC COUPLE<MAN<AGE INT, NAME ID>,
           WOMAN<AGE INT, NAME ID>>;
```

In this example, COUPLE is a 4-tuple with the elements MAN_AGE, MAN_NAME, WOMAN_AGE, and WOMAN_NAME. An example COUPLE is

```
{COUPLE$USER {MAN 37 FRANK}
             {WOMAN 13 MARY}}
```

Groups may also be repeated. This has the effect of repeating all the elements and groups that make up the group. For example:

```
DEC CORNER<STOPLIGHT BOOLEAN,
           STREET(2)<NAME ID, SURFACE ID>>;
```

CORNER is a 5-tuple with the elements STOPLIGHT, STREET_NAME[1], STREET_SURFACE[1], STREET_NAME[2], and STREET_SURFACE[2]. An example value is:

```
{CORNER$USER TRUE
             {STREET MAPLE CONCRETE}
             {STREET MAIN TAR}}
```

When the extent of repetitions is not known, an array may be used as an ntuple element, and of course, if necessary, the elements of the array may be ntuples. For example:

```
DEC WORD<PRINT ID, SPELL PHONE ARRAY(*)>,
    PHONE<GRAPHIC ID, FEATURES MODE>;
```

WORD is a 2-tuple with the elements PRINT and SPELL. An example value of WORD is:

```
{WORD$USER TEEN
           {PHONE$USER ARRAY(3)
            {PHONE$USER T (ALVEOLAR PLOSIVE)}
            {PHONE$USER IY (VOWEL BACK)}
            {PHONE$USER N (NASAL ALVEOLAR)}}}
```

Ntuples may contain groups nested to any depth, any of which may specify any number of repetitions. The net result is that to access an element, it is necessary to use the total number of subscripts specified by all groups containing that element and the element itself. For example:

```
DEC N<A INT,
      B(5) GEN,
      C(3,2)<D FLOAT,
             E(4) ID,
             F(17)<G INT, H(6) MODE3>>>;
```

N is a 750-tuple. The elements are (not in their order of occurrence) A, B[1] - B[5], C_D[1,1] - C_D[3,2], C_E[1,1,1] - C_E[3,2,4], C_F_G[1,1,1] - C_F_G[3,2,17], and C_F_H[1,1,1,1] - C_F_H[3,2,17,6]. The order of occurrence of the elements is

```
A,
B[1] - B[5]
  D[1,1]
    E[1,1,1] - E[1,1,4]
      G[1,1,1]
      H[1,1,1,1] - H[1,1,1,6]
      .
      .
      G[1,1,1,17]
      H[1,1,17,1] - H[1,1,17,6]
  .
  .
  D[3,2]
    E[3,2,1] - E[3,2,4]
      G[3,2,1]
      H[3,2,1,1] - H[3,2,1,6]
      .
      .
      G[3,2,17]
      H[3,2,17,1] - H[3,2,17,6]
```

The elements and groups are repeated in their lexical order of appearance, with the last (inner) subscript varying most rapidly. When a group is repeated, all of its elements and groups are also repeated. It is necessary to know the order of occurrence of elements only if binary data are exchanged with programs that are not written in CRISP.

An item's type may be omitted. When this is done, the item's type is the implicit type of the item's name. Thus:

```
IMPLICIT FLOAT(A TO H) INT(I TO N);
DEC N<A, I, O>;
```

is equivalent to:

```
DEC N<A FLOAT, I INT, O GEN>;
```

A group may be created in an ntuple by using a flat type. (As with arrays, the word, FLAT, does nothing unless the named type is a subspecified ntuple.) The following two declare forms define N as an ntuple with identical structure.

```
DEC N<X INT, Y FLAT M>, M<Z FLOAT, Q ID>;
DEC N<X INT, Y<Z FLOAT, Q ID>>;
```

In both cases, the elements of N are X, Y_Z, and Y_Q. With the following declaration, the elements of N are X and Y:

```
DEC N<X INT, Y M>, M<Z FLOAT, Q ID>;
```

However, the elements of M may still be referenced with the same qualifier sequences as in the above two examples, namely Y_X and Y_Q. See the section on item referencing and subscripts (page 88) for more details.

Assume that M has been declared by:

```
DEC M<Y GEN, Z INT>;
```

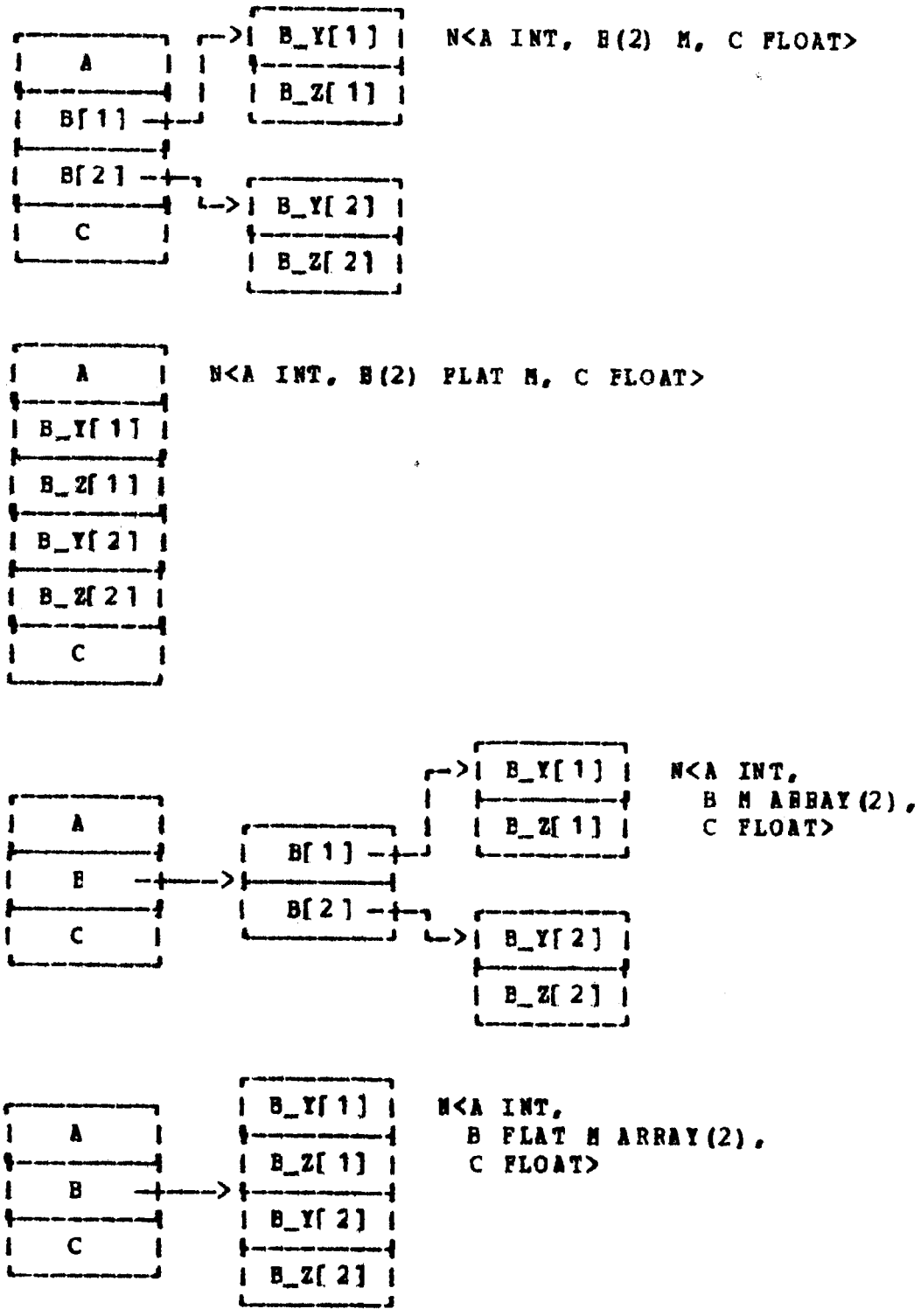
then Figure F shows the structure for each of these four declarations of N:

```
I.   DEC N<A INT, B(2) M, C FLOAT>;
II.  DEC N<A INT, B(2) FLAT M, C FLOAT>;
III. DEC N<A INT, B M ARRAY(2), C FLOAT>;
IV.  DEC N<A INT, B FLAT M ARRAY(2), C FLOAT>;
```

The following describes the properties of each of the four N's. The size computations are given, assuming that the repeat count had been "n" and specifically for n=2 (as in the actual examples). Sizes include all array and ntuple header information.

```
I.   A 3-tuple
     Elements in N - A, B[1], B[2], C
     Total size=5n+4, n=2 size=14 words
     Referenceable fields - A, B[1], B_Y[1], B_Z[1], B[2],
                            B_Y[2], B_Z[2], C

II.  A 6-tuple
     Elements of N - A, B_Y[1], B_Z[1], B_Y[2], B_Z[2], C
     Total size=2n+4, n=2 size=8 words
     Referenceable fields - A, B_Y[1], B_Z[1], B_Y[2],
                            B_Z[2], C
```




NTUPLE and ARRAY STRUCTURES

Figure F

```
III.  A 3-tuple
      Elements of M - A, B, C
      Total size=5n+9, n=2 size=19 words
      Referenceable fields - A, B, B[1], B_Y[1], B_Z[1],
                             B[2], B_Y[2], B_Z[2], C

IV.   A 3-tuple
      Elements of M - A, B, C
      Total size=2n+9, n=2 size= 13 words
      Referenceable fields - A, B, B_Y[1], B_Z[1], B_Y[2],
                             B_Z[2], C
```

As can be seen from Figure F and the above, there are a variety of sizes, shapes, and referenceable fields, depending on which of the four declaration forms is used. A virtue of making B an array is that the number of repetitions need not be known in advance. Declarations III and IV are then written:

```
DEC M<A INT, B M ARRAY(*), C FLOAT>;

DEC M<A INT, B FLAT M ARRAY(*), C FLOAT>;
```

When conservation of storage is a factor, then an analysis of which fields need to be referenced should be made to determine which usable representation is the most compact. When the number of repetitions is known, use a repeating element or group instead of an array. (This effects a minor savings of 5 words per structure for single repeats.) Also, when possible, use a flattened representation. (This saves 3n words per structure when there is a single repeat of a group of order n.) Further, to aid in parameterizing storage declarations, synonyms with numerical values may be used to specify the extent and lowest subscript of a repetition in an array or an ntuple. However, if the value of the synonym is changed, then it is necessary to recompile the declarations and the code that references them.

Declaration III above is the most flexible. Whenever storage conservation is not a major concern, or when it is hard to predict in advance the ways in which a structure will be decomposed, do not use flat groups but do use arrays for repeats. The referencing and subscripting mechanism has been designed to allow convenient switching among the four styles of declarations without change to programs that reference the fields that are still reachable. Thus, the storage layout may be optimized later, after all the evidence is in.

Ntuple types may be defined recursively. For example:

```
DEC ILIST<I INT, L ILIST>;
```

This defines ILIST as a 2-tuple with the elements I and L. The value of the element L must be either an ntuple of type ILIST or NIL. Some examples of ILIST structures are shown in Figure G. The first diagram is a list of integers. Its external, printing representation is:

```
{ILIST$USER 1 {ILIST$USER 2 {ILIST$USER 3 NIL}}}
```

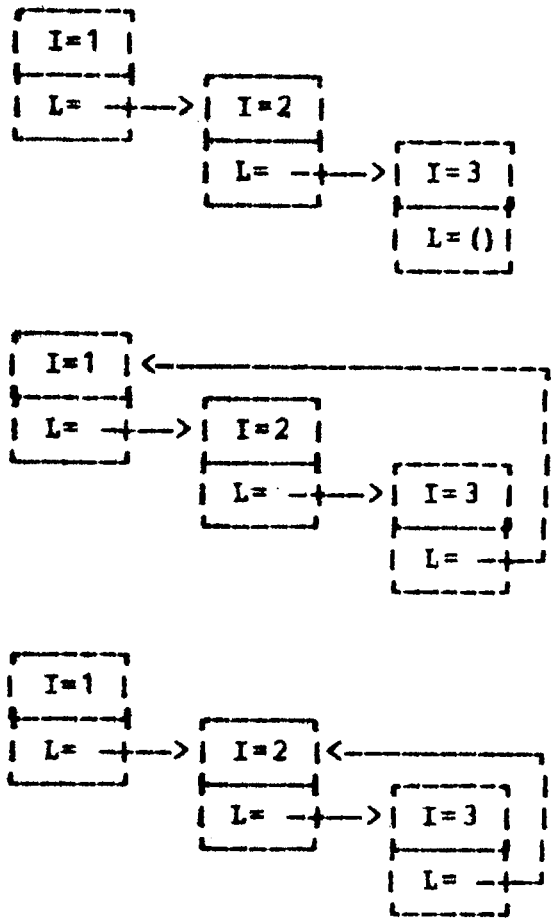
The second diagram is a simple three element ring. The third diagram shows a sort of combination of a list and a ring. The printing representation of both the second and third example is:

```
{ILIST$USER 1 {ILIST$USER 2 {ILIST$USER 3 *CIRCULAR*}}}
```

In some instances, a flat type and a recursive definition are incompatible. An obvious example is:

```
DEC ILIST<I INT, L FLAT ILIST>;
```

Other examples can occur when definitions are mutually




ILIST EXAMPLES

Figure G

recursive. For instance:

DEC A<I INT, L FLAT B>, B<F FLOAT, L FLAT A>;

The problem, in both cases, arises with the recursive (circular) definition because, at some point, a flattened ntuple must be inserted into "itself" an indefinite number of times. This is obviously an ill-defined situation. Three legal examples are:

```
DEC A<I INT, L B>, B<F FLOAT, L A>;

DEC A<I INT, L FLAT B>, B<F FLOAT, L A>;

DEC A<I INT, L B>, B<F FLOAT, L FLAT A>;
```

Figure 8 portrays these three examples. The left box in the top diagram is an A ntuple and the box on the right is a B ntuple. The field L_L may either point at an A ntuple or be NIL. The second diagram corresponds to A in the second example, where B is flattened in A. The field L_L is a pointer at another A or is NIL. The left box in the third diagram corresponds to A in the third example, where A is flattened in B. The right box corresponds to a B ntuple. The field L_L_I either points to a B ntuple (like the right box) or is NIL.

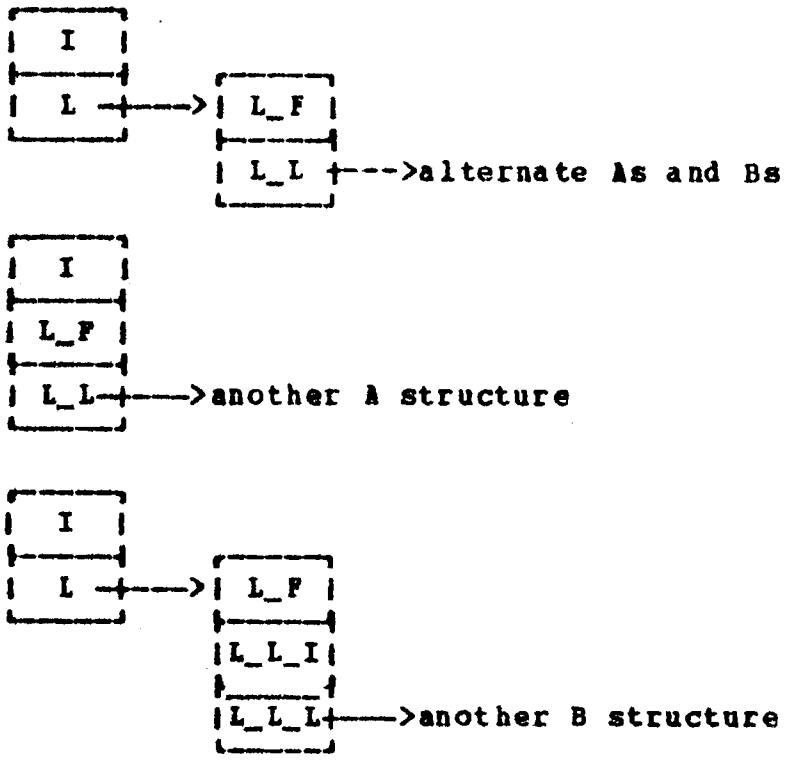
Declarations and Redeclarations

As a general rule, all names that make up a CRISP declaration or definition must have been given type attributes before they are used. The exceptions are the keywords in the language such as DECLARE, +, BLOCK, and names that appear in a position where they are given attributes such as V, W, X, and Y in this example:

```
DEC V INT;

INT FUNCTION W(X)
     BEGIN FLOAT Y;
      RETURN X+Y
      END;
```

This may cause problems when several functions call each other recursively or when declared types make circular references to each other. Also, there may be a desire to write functions in a "top down" manner, which would cause forward references. One method of handling these cases is




RECURSIVE FLATTENING

Figure H

to write a <declare> form that gives the attributes of all function names before the definition of the functions. This is very cumbersome. The compiler helps alleviate this by operating a declaration pass on each file. When a file is compiled, all SL is translated to IL. All <declare> forms are then joined together with all definitions. At this point, all the names are given their attributes simultaneously. In Figure 8 (page 35) a definition of the functions SUBST and SUBST1 was given along with a <declare>

form. The definition pass processes the derived form:

```
DEC GEN A$SYS, GEN B$SYS,
    GEN FUNCTION SUBST1$SYS(GEN),
    GEN FUNCTION SUBST$CRISP(GEN,GEN,GEN);
```

In interactive mode, because there is no file to process, forms are processed one at a time. However, all names given attributes by the same <declare> form are handled in the same fashion, simultaneously. Therefore, up to the order of preset computation, it never matters in what order the declarations appear in a <declare> form.

Another problem arises when names are redeclared with different attributes. Compiled code and data structures may reference the name and make assumptions about the name's value. To simply redeclare the name could lead to unrecoverable program checks. To avoid this, a "hiding" mechanism has been created.

Associated with each (global) name is a binding (with a value cell) and a symbol table entry. The symbol table entry includes, among other things, the name's first name, last name, type attribute, and some descriptor bits. Also, all (unhidden) name structures with the same first name are linked together. When a name with a type attribute is redeclared with a different attribute, the present structure associated with that name is hidden and a new name structure, with the same first and last name, is created with the new attributes. All code and data references that existed before now point to the hidden structure. All new references point to the new structure.

Part of the operation of the declaration pass is determining which names are to be hidden because of redeclaration. Before making the declarations, all of these names are hidden, and all references to them in the IL are changed to point at the new structure with the same name. Then all declarations are made. After this, the representations of all constant ntuples and arrays with ntuple elements are made into the proper data structures. Compilation of IL follows the same general pattern: a declaration pass followed by constant conversion. For user programs that generate IL programs and cause them to be compiled, some primitives are available. However, they do not operate in as general a manner. That is, they will not tolerate redeclarations. When necessary, these generating programs must explicitly hide the name being redeclared using the function HIDENAME (see below).

The process of hiding a name is simple. First, a bit in its symbol table entry is set to indicate that the name is hidden: second, the name structure is removed from the list of all structures with the same first name. Thus, the name is no longer found by the normal searching algorithms. To explicitly remove a name and its associated value object from the system, use the function:

```
NAME FUNCTION HIDENAME(NAME)
```

This hides the name. When there are no longer code or data references to the hidden name, it will be reclaimed.

## Item Referencing and Subscripts

This section describes the syntax and mechanism for referencing and setting ntuple and array fields. A field is an element in a particular ntuple or array. The syntax has been designed to permit declarations to be modified from flattened to unflattened form and from repeating groups and elements to arrays without changing programs that have already been written. Of course, the programs containing such references would need to be recompiled because the field references would now have a different meaning. The syntax for referencing ntuple fields when not subscripting is:

*SL*

<field>::=<ntuple-expression>$_<item-name>

*IL*

<field>::=([_]<ntuple-expression> $<item-name>)

An <ntuple-expression> is any expression whose value is an ntuple and whose exact subtype is known at compile time. For example, given:

```
DEC N<X INT, Y FLOAT>, M N;
```

the variables N and M would be instances of <ntuple-expression>s with type N. In this section, variables will be used as <ntuple-expression>s. For more complicated examples, see the section on expressions (page 130).

In the above example, the elements X and Y may be referenced in the ntuple N as the fields N_X and N_Y, respectively. Similarly, for the ntuple M, write M_X and M_Y.

In this 4-tuple:

```
DEC N<X, Y, Z<A, X>>;
```

the fields are N_X, N_Y, N_Z_A, and N_Z_X. It is also permissible to reference the field N_Z_A as N_A. But the field N_Z_X may not be referenced as N_X because there is another field known by that exact name. For another example, consider:

```
DEC N<A, A<X, A>, X<A, X>, X>;
```

The fields of N are N_A, N_A_X, N_A_A, N_X_A, N_X_X, and N_X. No shortening of any of these names is permitted. All short forms would conflict with other names. In:

```
DEC N<A, B<A, B<A, C, D>>, C>;
```

the fields are N_A, N_B_A, N_B_B_A, N_B_B_C, N_B_B_D, and N_C. N_B_B_C may be shortened to N_B_C and N_B_B_D may be shortened to N_B_D or N_D. The shortening rule is: (1) the ntuple expression may not be deleted, (2) the last item name may not be deleted, (3) any subset of the interior (remaining) item names may be deleted as long as the remaining sequence is not another field's name or another field's short name.

In order to handle subscripting of array and ntuple elements, the above syntax must be extended. Note, the actual specification of legal field forms is highly context dependent; these BNF forms serve only as the starting point

for the description:

```
*SL*
<field>::=<composite-expression>#{_<item-name>|
                                  <subscripts>}
<subscripts>::=[#*,*<subscript>_]
<subscript>::=<integer-expression>

*IL*
<field>::=([_]<composite-expression>#{<item-name>|
                                      <subscripts>})
<subscripts>::=($<subscript>)
<subscript>::=<integer-expression>
```

A <composite-expression> is an expression whose value is an ntuple or an array. The full subspecified type must be known at compile time. An <integer-expression> is any expression whose value is integer or is convertible to integer. The simplest case of subscripting is an array with simple elements. For instance,

```
DEC INT ARRAY(*) A;
```

Typical fields are A[1] and A[X-3*Y]. A more complicated example is an array with array elements.

```
DEC INT ARRAY(*,*) ARRAY(*) A;
```

As above, typical fields of A are A[1] and A[X-3*Y]. One way to subscript the fields in the fields of A is with forms like A[1][I,J+2] and A[Z][3,1]. Another way is to combine all the subscripts as in A[1,I,J+2] and A[Z,3,1]. These latter two forms are equivalent to the prior two forms. Yet another way of rewriting the forms with the same meaning is A[1,I][J+2] and A[Z,3][1]. Also, with the declaration:

```
DEC INT ARRAY(*,*) B;
```

the I Jth field could equally well be written as B[I,J] or B[I][J]. In general, the subscripts in a field reference may be arbitrarily grouped and placed. The compiler gathers them all together, preserving their lexical order of appearance, and then regroups and allocates them as appropriate. In any of the equivalent groupings and placements of subscripts, their relative order of lexical appearance will be identical.

The next example is an ntuple with a repeat:

```
DEC CORNER<STOPLIGHT BOOL,
           STREET(2)<NAME ID, SURFACE ID>>;
```

The fields are CORNER_STOPLIGHT, CORNER_STREET[I]_NAME and CORNER_STREET[I]_SURFACE where I=1 or 2. The following are equivalent under the item name deletion and the subscript movement rules:

```
CORNER_STREET[I]_NAME
CORNER[I]_STREET_NAME
CORNER_STREET_NAME[I]
CORNER[I]_NAME
CORNER_NAME[I]
```

A more complicated example is:

```
DEC N(2)<A(3), B<C(4,5)<A, D>, E>, F>;
```

The following are from among 40 equivalent field forms:

```
N_[I]_B_C[J,K]_A
N_B_C_A[I,J,K]
N_C[I,J]_A[K]
N[I,J,K]_B_A
```

The following is not an equivalent:

```
N_A[I,J,K]
```

That is, ambiguity is not resolved by the number of subscripts alone. (There is a field, N_A[X,Y], in N.) When

the compiler searches a type definition for the meaning of a field form, it first determines whether the form gives the full name of an element (all containing group names present). If so, that element of the value of the composite expression is the field denoted. If a full element name has not been given, then the compiler searches for the first element (in the lexical order of the declaration) that can have this shortened name. If the value of the variable CHKFIELD$CRISP is TRUE, then the compiler will continue the search to see whether this shortened name is ambiguous. If it is, a warning diagnostic is issued. In any event, the first element with the shortened name is used. The system is initialized with the form:

```
CHKFIELD$CRISP:=TRUE
```

So far, the field referencing mechanism is virtually identical to PL/I. However, CRISP allows structure items to be arrays, ntuples, or even functionals (funcs and procs). Therefore, many cases that do not arise in PL/I can occur in CRISP. The rest of this section describes these cases and the mechanism for handling them.

The next example shows some ntuples, flattened and unflattened, that are included in other declarations:

```
DEC M<A N, B FLAT N, C O, D FLAT P>,
    N<X, Y>, O<Q, R>, P<R, S>;
```

M is declared as a 6-tuple with the following fields in the value of M: M_A, M_B_X, M_B_Y, M_C, M_D_R, and M_D_S. In addition, the following fields may be reached from M: M_A_X, M_A_Y, M_C_Q, and M_C_R. M_A_X and M_A_Y are the

fields in the ntuple of type N pointed at by the field M_A. M_C_O and M_C_B are the fields of the ntuple of type O pointed at by the field M_C. In total, ten fields are reachable from M in this example. All may be accessed using the field forms shown above. Only two of the field forms may be shortened: M_D_S may be written M_S, and M_C_Q may be written M_Q. Thus, part 3 of the shortening rule refers to all reachable fields. (This restriction is slightly relaxed for recursive definitions. See below.)

The next example is an array with ntuple elements:

```
DEC N<X, Y, Z>,
    N ARRAY(*) A,
    FLAT N ARRAY(*) B;
```

The fields in A are writen A[I] and each is a pointer at an ntuple of type N (or NIL). The fields in the fields of A are: A[I]_X, A[I]_Y, and A[I]_Z, or with the subscript moved, A_X[I], etc. In B, there is no such field as B[I]. The fields in B are B[I]_X, B[I]_Y and B[I]_Z, or the versions with the subscripts moved such as B_X[I].

A more complicated example is given next:

```
DEC WORD ARRAY(*) SENT,
    WORD<PRINT ID, SPELL PHONE ARRAY(*)>,
    PHONE<GRAPHIC ID, FEATURES MODE>;
```

The fields in SENT are of the form SENT[I] and are ntuples of type WORD. The fields in the WORD ntuples may be referenced by writing SENT[I]_PRINT or SENT[I]_SPELL, an array with fields of the type PHONE. To reference the Jth ntuple of type PHONE in the Ith WORD, write SENT[I]_SPELL[J] or one of the equivalent forms, SENT[I,J]_SPELL or

SENT_SPELL[I,J]. To reference the field, GRAPHIC, in the PHONE ntuple, write any of these equivalent forms:

```
SENT[I,J]_SPELL_GRAPHIC
SENT[I]_SPELL[J]_GRAPHIC
SENT[I]_SPELL_GRAPHIC[J]
SENT_SPELL[I,J]_GRAPHIC
SENT_SPELL[I]_GRAPHIC[J]
SENT_SPELL_GRAPHIC[I,J]
SENT[I,J]_GRAPHIC
SENT[I]_GRAPHIC[J]
SENT_GRAPHIC[I,J]
```

The last three forms are shortened names. For purposes of resolving the denotation of field forms, the compiler does not distinguish groups and repeating elements from pointer types at arrays and ntuples. The FEATURES fields are referenced in a manner similar to the above. Since the type of this field is node (node2), a _FIRST or _SECOND may be appended to access the CAR and CDR portions. For examples:

```
SENT_SPELL_FEATURES_FIRST[I,J]
SENT[I,J]_FIRST
```

Both of these forms would retrieve the CAR field: the first feature of the Jth phone in the Ith word of the sentenence.

The next topic to be discussed is referencing of fields from structures that have recursively defined types, or structures that eventually point at structures with recursively defined types. The problem arises because the set of fields referenceable from such a structure is indefinitely large and because for all fields in a recursively defined structure, there are arbitrarily many other fields with the same short names. To handle these cases (and to keep searching routines from going into infinite loops when processing illegal names), the compiler keeps a list of all definitions and groups in these

definitions that have been examined while looking for the i+1st item name since locating a tentative candidate for the ith item name. The searching routines will terminate this portion of the search tree if an attempt is made to pass through any member of this list again. A simple example is:

```
DEC X<N, L X>;
```

In processing X_L_Y, an error, the compiler matches the item named L and sets the list with the single element X_L. The definition of X is then recursively examined for Y. N is not Y, and L is not Y, so an attempt is then initiated to search from the element X_L. But this is already on the list. Therefore, processing stops, the recursion unwraps and, finding no other possibilities, the compiler reports an error. Examples of legal field forms are X_N, X_L, X_L_N, X_L_L, etc.

So far, all the information and examples above have been about SL <field> forms. The IL <field> form is virtually identical. The first thing in an IL <field> form may be _, which is optional. If the <field> form has a legal <composite-expression> (that is, if the compiler can determine the array or the ntuple subtype), then the _ serves no purpose. However, if there is an error in the composite expression or if its subtype can not be properly determined, the use of the _ leads to better diagnostics. Assume that the variable N is undeclared, then the IL form:

```
(N A (I J))
```

is in error. The diagnostic mechanism will interpret this as an illegal function call form with the arguments A and (I

J). The second argument looks like a call on the function I with argument J. If the form had instead been written:

(_ N A (I J))

the form would still be in error (N's type is still unknown) but the diagnostics would know that A was an item name rather than a variable and so would ignore it. Also, I and J would be treated as subscripts, hence variables, and a diagnostic about I being an illegal form operator would not result.

## Definitions

This section gives a syntax and a prose description of the CRISP mechanism for defining functions, processors, macros, transforms, and generators. See the section on declarations and redeclarations (page 84), for information on what happens when the definition redefines a name with a different type attribute. The syntax of definitions is:

```
*SL*

<definition>::=<function-def>|<processor-def>|<macro-def>|
               <transform-def>|<generator-def>

<function-def>::=[<value-type>] FUNCTION <name>
                    <arg-list><expression>

<processor-def>::=PROCESSOR <name><arg-list><expression>

<macro-def>::=MACRO <name>([<arg-scope>]<name>)<expression>

<transform-def>::=TRANSFORM <name>($=,=<identifier>)
                           [<gen-form>]<form>

<gen-form>::=GENID $=,=<identifier>

<generator-def>::=GENERATOR <name>([<arg-scope>]<name>)
                           <expression>

<arg-list>::=($=,=<arg-dec>)|($[<arg-dec>,]<indef-arg-dec>)

<arg-dec>::=[<type-ref>][<arg-scope>][VARIABLE]<name>

<indef-arg-dec>::=[<type-ref>] INDEF <identifier>|
                 [<type-ref>][<arg-scope>] LIST <name>

<arg-scope>::=<global-scope>|<local-scope>

<local-scope>::=LOCAL


*IL*

<definition>::=<function-def>|<processor-def>|<macro-def>|
               <transform-def>|<generator-def>

<function-def>::={FUNCTION {<name>|(<name><value-type>)}
                    <arg-list><expression>)
```

```
<processor-def>::=(PROCESSOR <name><arg-list><expression>)
<macro-def>::=(MACRO <name>(<name>[ <arg-scope>])
                    <expression>)

<transform-def>::=(TRANSFORM <name>($<identifier>)
                              [ <gen-form>]<form>)

<gen-form>::=(GENID #<identifier>)

<generator-def>::=(GENERATOR <name>(<name>*<arg-scope>*)
                              <expression>)

<arg-list>::=($<arg-dec>[ <indef-arg-dec>])

<arg-dec>::=<name>|(<name>[ <arg-scope>][ <type-ref>])

<indef-arg-dec>::=(<identifier> INDEF [ <type-ref>])|
                  (<name> [ <arg-scope>] LIST [ <type-ref>])

<arg-scope>::=<global-scope>|<local-scope>

<local-scope>::=LOCAL
```

For all the above, the name being defined is global. If it is an identifier, then it is paired with the default tail to derive the proper name. The section on scoping and denotation rules (page 25) gives more information on this.

Arg list

An <arg-list> specifies the parameter names for a function or processor. It also determines each parameter's type attribute. First, the proper name of each parameter is determined by the rules given in the section on scoping and denotation rules (page 25). To repeat, there are six cases: the <arg-scope> may be LOCAL, GLOBAL, or omitted, and the parameter name may be either an identifier or a global name. This table summarizes the action in each of the six cases for determining a parameter's proper name.

| Scope | Used Name: I | Used Name: A$B |
|---|---|---|
| LOCAL | I | error |
| GLOBAL | I$default | A$B |
| omitted | I | A$B |

I and A$B are arbitrary examples of, respectively, an identifier name and a global name. I$default is the identifier name paired with the default tail to form a global name.

After the parameters' proper names have been determined, their type attributes are determined. There are six cases: the proper name may be local (an identifier), a global name with a type attribute, or a global name without a type attribute, and a <type-ref> may be present or omitted as part of the <arg-dec>. The following table summarizes the action in determining each parameter's type attribute given its proper name. Ltr is the <type-ref> given in the <arg-dec> if present. Gtr is the global name's type attribute if it exists. Global name 1 is a global name with type attribute, and global name 2 is a global name without type attribute.

| type ref | Proper Name: local name | Proper Name: global name 1 | Proper Name: global name 2 |
|---|---|---|---|
| present | ltr | error unless ltr=gtr | make dec using gtr |
| omitted | use implicit type | gtr | make dec using implicit |

If a global name is used that has a type attribute, an error

occurs if the <arg-dec> attempts to redeclare it. Also, if a global name does not have a type attribute, the compiler forces a declaration for it. If a <type-ref> is given with the <arg-dec>, then that becomes the name's type attribute. If the <type-ref> is omitted, then the declaration of the name is made using the name's implicit type.

Essentially, the above applies to an <arg-dec> or a list arg (an <indef-arg-dec> with LIST specified). A normal <indef-arg-dec> is handled slightly differently. The indef arg name is always an identifier and is always treated as a local variable with the type attribute integer. When the function is called, the initial value of the variable is the number of values of the indef. The type of the indef values is determined by the <type-ref> if present in the <indef-arg-dec>. Otherwise, the type is the implicit type of the identifier. To retrieve the value of the Nth indefinite, use the pseudo function, ARGN, with an integer argument. The legal value of ARGN's argument is N where $1 \leq N \leq L$ and L is the number of arguments. A simple example of a function with an indefinite number of arguments is:

```
NODE FUNCTION FOO(INT X, FLOAT INDEF L)
      FOR I:=1 THRU L,
          LIST ARGN(I)**X
          ENDF;
```

FOO returns a list of its floating, indef arguments raised to the power of its integer argument. For example, the value of

```
FOO(2, 4.0, -1.5, 6)
```

is

```
(16.0 2.25 36.0)
```

A list arg is another method by which a function may receive an indefinite number of arguments. The indef args are CONSed together into a node2 list. The list arg name naturaly has the type attribute, node2. The <type-ref> is the type of the list elements. The function FOO above, could be rewritten as

```
NODE2 FUNCTION FOO(INT X, FLOAT LIST L)
        FOR M ON L, VALUE L
            DO CAR(M):=CAR(M)**X;
               ENDF;
```

Function defs

When a function (or processor) is called, the arguments are converted to the types specified by the <arg-list>. All such conversions are error checked for legality at compile time when possible, otherwise at run time. If error checking mode has been turned off (by setting the variable ERRCHK$CRISP to NIL) compile time error checking is still done but at run time there is no error checking when conversions are made from one type with pointer values to another type with pointer values. Conversions from general and number to integer or float are diagnosed at run time whether or not error checking mode is used. All argument passing is by value.

After a function is entered and its argument variables bound, its expression body is evaluated. If the function's <value-type> is NOVALUE, then the function simply returns. Otherwise, the value of the expression is converted, if necessary, to the type specified by the value type. As with argument type conversion, error checking is done. If the

function is compiled with error checking mode turned off, then pointer type to pointer type error checking is not done at run time. See the section on data object formats (page 55) for more information on types with pointer representations.

Processor defs

A <processor-def> has an <arg-list> that is handled exactly as is a <function-def>'s arg list; see above paragraphs. When a processor is entered, a new process is created and the calling process is suspended. For information on process evaluation, see the section on processors and processes (page 173). The expression body of a process is not responsible for producing a value, because a processor has no value.

Macro defs

A macro is just like a function with the type:

```
GEN FUNCTION(NODE2)
```

A macro operates at compile time. Its argument is an IL form being compiled, whose operator is the name of the macro. The value of the expression body of the macro is compiled in place of the original form. The argument and value of a macro are IL forms. An example is:

```
MACRO IMPLY(F)
   LIST("OR, LIST("NOT, CADR(F)), CADDR(F));
```

Example argument-value pairs are:

```
(IMPLY A B)
(OR (NOT A) B)
```

```
(IMPLY (NOT A) (AND B C))
(OR (NOT (NOT A)) (AND B C))
```

The SL forms corresponding to these two are:

```
IMPLY(A, B)   or   A>>B

IMPLY(¬A, B&C)   or   ¬A>>B&C
```

Although this example of a macro is so simple that it could be replaced by a transform, macros may be extremely complex, involving conditional computation. For instance, the <for-loop> form is handled through macro expansion. It turns loop forms into forms made up of language primitives like BEGIN, PLUS, GOTO, etc.

Transform defs

Transforms are described in the section on compile time substitutions (page 41). Another example would be rewriting the IMPLY macro above as a transform.

```
TRANSFORM IMPLY(A,B) ¬A|B;
```

A transform may use an optional <gen-form>. A <gen-form> specifies a set of identifier symbols for which unique identifiers should be substituted each time the transform is used. The new identifiers are created using GENID. This is particularly useful for creating labels when the transform will be used more than once in a single function.

Generator defs

Generators are described in the forthcoming document, CRISP Compiler and Assembler Structure.

## Expression Typing

The topic of this section is how the compiler determines the type (of the value) computed by an expression. Expressions are defined recursively: first as a set of primitive forms and second as forms made up from other expressions. The rules for determining an expression's type follow the same pattern. It is assumed that the reader is familiar with other parts of this document that deal with constants, variables, function calls, arithmetic forms, expressions and blocks. Throughout this section, reference is made to the type hierarchy shown in Figure D (page 49).

### Primitive forms

The type of a constant form is the lowest type in the hierarchy that contains that object. Thus, 3.7 is typed as float rather than number or general. The type of a variable form is its type attribute. The type of a function call form is its value type.

### Arithmetic forms

Some arithmetic forms act like function calls in that they always return the same type of value. For instance, the infix operator "//" returns an integer value from integer division. However, the infix operators +, -, *, /, and ** return a value whose type depends upon the types of their arguments. The possible argument and value types are integer, float, complex, and number. If any argument is complex, then the value is complex. Otherwise, if any

argument is number, then the value is number. Otherwise, if any argument is float, the value is float. Otherwise, all arguments are integer and the value is integer.

The type of value produced by the unary + and - is the same type as its argument. The pseudo functions ABS and NABS produce the same value type as their argument unless the argument is complex. In this case the value type produced is float (the norm or the negative norm).

Assignment typing

An assignment form may be used as an expression (as opposed to a statement), in which case it produces a value. The type of the assignment form is the type of its left side, the receiving field. Thus, if I is an integer variable, then the form "I:=5.7" has the value "5". If the types of the left and right sides are different, and if both are types represented by pointers, then the type is the most restrictive that can contain both sets of values (lowest in Figure D, page 49). For instance, if A has the type attribute INTEGER ARRAY(*) and G has the type attribute general, then the value type of both these forms is INTEGER ARRAY(*):

A:=G and G:=A

Multi-terminal forms

Several language forms may compute their values in more than one spot. Each such spot is called a terminal computation. Examples are A and B in the IF form

```
IF P THEN A ELSE B
```

Other forms that may compute values in more than one place are selectors and blocks. (Each return form in a block is a terminal.) In all these cases, the type of each terminal is determined. Then, the type of the parent form is the lowest type (in Figure D, page 49) that contains all the objects in the terminals' types. For example, the value type of

```
IF P THEN 1 ELSE 1.0
```

is number. The type of

```
IF P THEN "X ELSE "ABC
```

is identifier.

## Type Conversion

Type conversion is always possible upwards as defined by Figure D (page 49). Sidewards and downwards conversions are not always possible. Whenever it makes sense, it is allowed. Boolean is always convertible to identifier. Integer is always convertible to float and complex, even though precision (low order bits) may be lost. Float is always convertible to complex. Complex conversion to float or complex and float conversion to integer may not be possible. In these cases, a run time error will occur if the value of the boolean variable PRECISIONP is TRUE; otherwise, some such value as 0, "very large" positive, or "very large" negative will result. The system is initialized with

```
PRECISIONP:=TRUE;
```

An interesting case occurs when a func (proc) of one type is converted to a func (proc) of another type. Assume that it is desired to convert a value of the type

v1 FUNC(a11 ... a1n) to the type

v2 FUNC(a21 ... a2m)

This is permissible if and only if:

(1) n=m

(2) a1n and a2m must both be of the same form -- e.g., both indefs, lists, simple, or non-existent

(3) v1=v2 or v1 and v2 are represented by pointers and v1<v2

(4) for $1 \le i \le n$, a1i=a2i or a1i and a2i are represented by pointers and a2i<a1i.

Similarly for proc values. If it is desired to convert a

value of the type v1 VARB to the type v2 VARB, then either v1=v2 or v1 and v2 must both be represented by pointers and v1<v2. The operator "<" is the order relation in Figure D (page 49).

# BLOCKS

There are three kinds of <block> forms in CRISP: <binding-block>s, <do-block>s, and <multi-form-block>s. A binding block can bind variables; the other two kinds of blocks cannot. <binding-block>s and <do-block>s have a body consisting of zero or more statements and labels, while a <multi-form-block> is an ordered collection of forms. All types of blocks may be used either as statements or as expressions (in which case they produce values). The sections on scoping and denotation rules (page 25); declarations, definitions, and types (page 45); and data presets (page 127) should be reread along with this section for better comprehension.

This section gives the syntax of blocks, statements, and related forms, then describes them with the subsections Multi-form blocks, Do blocks, Bind blocks, and Statements and Labels.

The syntax is:

```
*SL*

<block>::=<binding-block>|<do-block>|
          <multi-form-block>

<binding-block>::=BEGIN <block-bind-list>;
                        [<attribute-list>]
                        <block-body>
                        END

<do-block>::=DO <block-body> END
```

```
<multi-form-block>::=<multi-statement-block>|
                     <multi-expression-block>

<multi-statement-block>::=($*,*<statement>)

<multi-expression-block>::=($*,*<expression>)

<block-bind-list>::=$*,*{<block-var-dec>|<local-syn-dec>}

<block-var-dec>::=[<type-ref>][<block-scope>][VARIABLE]
                  <name>[<preset>]

<block-scope>::=<global-scope>|<local-scope>|<own-scope>

<own-scope>::=OWN

<local-syn-dec>::=SYN <identifier>:=<form>


<attribute-list>::=${<attributes>;}

<attributes>::=ATTRIBUTE $<block-attr>

<block-attr>::={<type-ref>[<block-scope>]|<block-scope>}
                    ($*,*<name>)

<block-body>::=$*;*{<statement>|<label>}[;]

<statement>::=<expression>|<statement-only>

<statement-only>::=<go>|<return>|<leave>

<go>::=<simple-go>|<computed-go>

<simple-go>::={GO|GOTO} <label>

<computed-go>::={GO|GOTO} ($*,*<label>) [ON]
                  <integer-expression>

<return>::=RETURN <expression>

<leave>::=LEAVE [<integer>]

<label>::=<identifier>


*IL*

<block>::=<binding-block>|<do-block>|
          <multi-form-block>

<binding-block>::=(BEGIN <block-bind-list>
                         <block-body>)

<do-block>::=(DO <block-body>)

<multi-form-block>::=<multi-statement-block>|
                     <multi-expression-block>
```

```
<multi-statement-block>::=(MULTI $<statement>)

<multi-expression-block>::=(MULTI $<expression>)

<block-bind-list>::=($[<block-var-dec>|<local-syn-dec>])

<block-var-dec>::=<name>|
                  (<name>[<block-scope>][<type-ref>]
                     [<preset>])

<block-scope>::=<global-scope>|<local-scope>|<own-scope>

<own-scope>::=OWN

<local-syn-dec>::=(<identifier> SYN <form>)

<block-body>::=$[<statment>|<label>]

<statement>::=<expression>|<statement-only>

<statement-only>::=<go>|<return>|<leave>

<go>::=<simple-go>|<computed-go>

<simple-go>::=(GO <label>)

<computed-go>::(GO (#<label>) <integer-expression>)

<return>::=(RETURN <expression>)

<leave>::=(LEAVE [<integer>])

<label>::=<identifier>
```

Multi-form blocks

A multi-form block allows the insertion of several expressions (statements) wherever the syntax allows a single expression (statement) to appear. The forms in a multi-form block are evaluated one at a time, in a left to right order. When the multi-form block is used as an expression, then the value of the last expression is the value of the form. The most trivial usage is with only one expression to break the normal precedence relation of operators. Thus, (A+B)*C may be used to overcome A+(B*C) as the normal interpretation of A+B*C. Some other examples of <multi-expression-block>s

are:

```
A:=(B:=C, D)

(F(G), H(I), J(K))
```

In IL, the <multi-form-block> is equivalent to the LISP PROGN. A <multi-form-block> of no arguments, e.g. (MULTI), is the same as NIL.

When used as an expression, all except the last expression may produce no value -- for instance, by calling a novalue function. However, the last expression must produce a value that becomes the value of the form. Also, the type of the last expression's value is the type of the multi-expression block.

When a multi-form block is used as a statement, the body of the block is a sequence of statements. There are no provisions for placing labels in a multi-form statement.

Do blocks

A <do-block> is exactly equivalent to a <binding-block> with the same block body, a null <block-bind-list>, and no attribute forms. The <do-block> is provided only as a convenience to the user.

Binding blocks

A <binding-block> is one of the most powerful and useful forms in the language. It allows local and global variable bindings, local synonyms and own variables to be created, and statement mode to be used.

A <block-bind-list> specifies the variables to be bound when the block is executed, along with their type attributes, scopes, and initial (preset) values. In SL, <attribute-list>s may be used to specify the variables' attributes and scopes. (There is no attribute form in IL.) The attribute form is a purely syntactic mechanism that operates as SL is translated to IL. The name specified in an attribute form must be exactly a name appearing in the <block-bind-list>. The following is an error:

```
BEGIN A GLOBAL;
      ATTRIBUTE INT(A$B);
```

That is, the information given by an attribute form is distributed before proper names are determined. The following is ambiguous, and therefore an error:

```
BEGIN A, A GLOBAL;
      ATTRIBUTE INT(A);
```

Redundant type or scope specification is permissible. However, if there is a conflict among the attributes and <block-bind-list>, then it is an error. For instance,

```
BEGIN INT A;
      ATTRIBUTE FLOAT A;
```

and

```
BEGIN A;
      ATTRIBUTE INT(A);
      ATTRIBUTE FLOAT(A);
```

are both errors.

Once the attribute forms' information has been distributed, local and global variables' (not local synonyms' and owns') proper names and type attributes are determined in a manner identical to that used for processing a function's or a processor's argument list. See the section on arg list

(page 98) for a complete description.

A scope of OWN may be specified for a block variable with an identifier name. For such a variable, the preset value is computed at compile time. An own has only one binding and thus is not rebound when the block is entered. The scope (visibility) of an own variable is the same as that of a local variable bound in the same <block-bind-list> -- that is, in this block and other blocks nested in the one with the own declaration. See the section on own variables (page 44) for more information.

A local synonym specifies a language form that is to be substituted for occurrences of the synonym's name within its scope. The scope of a local synonym is the same as the scope of a local variable bound in the same <block-bind-list>. See the section on compile time substitutions (page 41) for more information.

When a block is entered, all presets are evaluated in the left to right order in which they appear in the <block-bind-list>. All the block variables are then bound simultaneously. This is similar to making a function call where arguments are all evaluated, then the function is entered and the parameters are bound. This means that the preset computations cannot use the other variables in the same <block-bind-list>.

```
BEGIN A:=1, B:=2;
      BEGIN A:=B, B:=A;
```

In the inner block, the initial values of the variables A

and B are, respectively, 2 and 1 (not 2 and 2).

Statements and labels

The body of a binding or do block is an ordered sequence of statements and labels separated by semicolons and terminated by the word END. It is not necessary, but is permissible, to include a final semicolon before the END. Statements may be any expression except an identifier, which is interpreted as a label. Also, there are several language forms that may be used only as statements, not expressions. All <statement-only> forms cause some sort of control transfer. Most forms in the language may appear as either statements or expressions.

That a form appears in statement or expression mode is an hereditary property of many forms. If an if form is a statement, then its terminals (then and else clauses) are also statements. If an if form is used as an expression, then its terminals are also expressions. Similarly, for the selector forms, the terminals are statements or expressions as the parent is a statement or expression.

As can other forms, do and binding blocks (hereafter simply called blocks) may be used either as statements or as expressions. The <return> form is used to generate the value of a block used as an expression. When executed, the expression body of the return form is evaluated, and its value becomes the value of the most closely nested expression block that contains the return. The bound

variables in all the inner blocks and the expression block are unbound. For example,

```
A:=BEGIN X;
         IF P THEN RETURN F(X);
         X:=G();
         BEGIN Y:=17;
               RETURN H(X,Y);
               END
         END
```

This assignment form sets the value of the variable A to the value of the outer block: first the outer block is entered, then X is bound, and P is tested. If the value of P is TRUE, then F(X) is evaluated, X is unbound, and the value of F(X) becomes the value of the outer block and is stuffed in A. Note, this outer block is used as an expression. If the value of P is not TRUE, then X is set to the value of G() and the inner block is entered with the local variable Y initialized to 17. The inner block is a statement block because it appears as a statement in the outer block. The form H(X,Y) is then evaluated and becomes the value of the outer, expression block. After H(X,Y) is evaluated and before the value is stuffed in A, the variables X and Y are unbound.

The <leave> form is used to terminate execution of a set of statement blocks. The form LEAVE is identical to LEAVE(1). The integer part of the <leave> form specifies how many of the most closely nested blocks (that contain the <leave> form) to terminate. If all the terminated blocks are not statement blocks, a compile time error diagnostic will be issued. When a set of blocks is terminated, control "falls through" the outermost terminated block. The <leave> form

is provided as a user convenience to avoid dreaming up label names for this purpose. A caution when using macros and <leave> forms together should be remembered. If a macro (like FOR) expands in terms of a block, and a <leave> form is in the interior of that expansion, then the leave count must be increased by one to count for the invisible block.

A <label> marks a place in a program that may be addressed by a <go> form. In CRISP, there are no label variables. If L is a label and B is the block in which L is defined, then the following rules define the spots where L is visible, that is, the places where forms such as "GO L" may appear.

1. L is visible to the top level statements in B.
2. If L is visible to an IF, multi-statement block, or selector form, then it is visible to the terminals of that form.
3. If L is visible to a block, then it is visible to the top level statements of that block.
4. If the block in which L appears (B), or a block to which L is visible by this rule (part 4) is a statement block that binds no local or global variables, then L is visible to the next outer block.

The following example shows a label, L, and a variety of legally placed branches to L.

```
BEGIN X;
  IF P THEN BEGIN A; GO L END;
  IF Q
     THEN BEGIN;
            L:F(X);
              BEGIN A;
                IF R THEN GO L;
                BLOCK B; GO L END;
                GO L
              END
              GO L
            END;
  GO L
END
```

There are two types of go forms: <simple-go> and <computed-go>. In both cases, if the go branches out of one or more (statement) blocks that bind variables, then the variables bound by the terminated blocks are unbound before program control resumes at the label. A <computed-go> specifies a list of labels. The value of the integer expression selects which label to transfer to. If the value of the expression is less than or equal to 1, then the first label in the list is transferred to. If the value is greater or equal to the number of labels in the list, then the last label is transferred to. In all other cases, if the value of the integer expression is i, then the ith label in the list is transferred to.

In any one function, processor, macro, or generator definition, there may not be any duplicate label names.

# DATA PRIMITIVES and PRESETS

This section describes the available primitives for allocating data structures, accessing their fields, the mechanism that initalizes variables appearing in declare forms, and the mechanism that computes the presets (initial values) for block variables when a block is entered.

## Data Primitives

Many forms in the language automatically allocate data structures. For instance, evaluation of "A+B" sums the current values of the variables A and B and allocates a data structure of the appropriate type to hold the value. However, this section describes only those primitives that are explicitly used for allocation. It should be noted that the various read primitives also create structures. Each of the paragraphs following in this section describes the allocation and accessing primitives associated with a particular kind of data object. All primitives have the last name (or tail), CRISP.

### Identifier and character primitives

An identifier object can be created with the function,

```
ID FUNCTION COMPRESS(NODE2)
```

The argument to COMPRESS is a list of character identifier objects. The value is the identifier whose name is that

string of characters. Thus,

```
COMPRESS("(A B C))  is  ABC
COMPRESS("(A $' ' B))  is  $'A B'
```

The function,

```
MODE2 FUNCTION EXPLODE(ID)
```

takes an identifier as an argument and returns a list of character objects that are the id's print name. Thus,

```
EXPLODE("ABC) is (A B C)
```

GENID may be used to create a unique, new identifier. The identifier is created without a name, just a structure. If a genid (generated identifier) is printed or exploded, then it is given a name of the form Gxxxxxxx, where x is a digit. When the name is generated, it is guaranteed to be different from the name of any existing identifier. The declaration of GENID is

```
ID FUNCTION GENID()
```

The implicit type of a genid, before it receives a name, is general: if it receives a name, then it is treated as an ordinary identifier.

The functions CHAR2INT and INT2CHAR convert a character identifier to its integer EBCDIC equivalent and vice versa. The integer must lie in the range 0 to 255. Thus,

```
CHAR2INT("A)  is  0C1X
INT2CHAR(0C1X)  is  A
```

The declarations of these pseudo functions are

```
CHAR FUNCTION INT2CHAR(INT)
INT FUNCTION CHAR2INT(CHAR)
```

Each identifier has a property object of type general. The property object may be accessed using a form such as I_PROP, where I is a variable with type attribute ID or CHAR. Such a form as

INT2CHAR(X)_PROP

is also legal and may even appear as the left side of an assignment form. It is normal to maintain the property object as a set of pairs. One element of the pair is an identifier that is used as a search key, and the other member of the pair serves as the value of that property. To facilitate this usage, three functions are provided:

GENERAL FUNCTION SETPROP(ID,ID,GENERAL)

GENERAL FUNCTION GETPROP(ID,ID)

GENERAL FUNCTION REMPROP(ID,ID)

In all three, the first argument is the identifier whose property set is to be manipulated and the second argument is the property name (or search key).

SETPROP gives the named property the value of its third argument. If a property with that name already exists, then it is replaced; otherwise, a new pair is added to the property set. The value is the third argument.

SETPROP("AB,"P,12)

The value is 12. The value of the property P under the identifier AB is now 12. GETPROP retrieves the value of the specified property. If none exists, then NIL is returned. After execution of the above SETPROP,

GETPROP("AB,"P) is 12

REMPROP removes the specified property pair. Thus, after

REMPROP("AB,"P),

GETPROP("AB,"P) is NIL.

The value of REMPROP is its second argument.

Caution: if any of the above functions are used, use of the id_PROP form should be avoided because it may corrupt the assumed, standard format of the property object.

An additional set of property functions is being considered but has not yet been designed. These new functions would allow the use of an ordered sequence of property names instead of just one. This, in effect, would make the property object into a tree with the identifier and property names identifying the nodes, thereby allowing private property sets.

Node primitives

The allocation primitive for all node objects is the pseudo function CONS. CONS may have one to eight arguments of type general. The value is a node with as many fields as there are arguments. For example,

the value of CONS("A,"B,"C) is {NODE3 A B C},

the value of CONS(1,"X,3.7,"AB) is {NODE4 1 X 3.7 AB} and

the value of CONS(17) is {NODE1 17}.

In SL, the infix operator, #, may be used for consing node2 structures, and the infix operator, ##, may be used for consing node2, node3, ... node8 structures. See the section on expressions (page 130) for details.

The fields in a node object may be accessed by the pseudo item names FIRST, SECOND, THIRD, FOURTH, FIFTH, SIXTH, SEVENTH, and EIGHTH. For example, if N is a variable with type attribute node3, and the value of N is (NODE3 A B C), then

N_FIRST is A

N_SECOND is B

N_THIRD is C

These forms may also appear on the left side of an assignment form.

The pseudo functions, CAR, CDR, ... CDDDDR, are available for accessing the fields of node2 objects in the LISP tradition. The form is a C followed by one to four As or Ds followed by an R appearing as an operator. By example,

CADDR(X)

is exactly equivalent to

CAR(CDR(CDR(X)))

CAR(X) is like X_FIRST and CDR(X) is like X_SECOND. If the CAR - CDR forms have a general argument, then they assume that the argument should be of type node2. The pseudo item name forms make no such assumption. The compiler must be able to determine that the expression part is some kind of node or an error diagnostic is issued. CAR and CDR forms may appear on the left side of an assignment.

Name primitives

Objects of type name (or one of its subspecified types) may be created or "found" by the function,

```
NAME FUNCTION MAKENAME(ID,ID)
```

The two arguments are the name's first and last name (or tail). If that name already exists, then MAKENAME returns it as its value. Otherwise, an object by that name is created (with the type attribute name) and returned. The function,

```
NAME FUNCTION FINDNAME(ID,ID)
```

is also available and is similar to MAKENAME. However, if the name does not already exist, then none is created and the value is NIL. A name may be hidden by using

```
NAME FUNCTION HIDENAME(NAME)
```

See the section on declarations and redeclarations (page 84) for use and meaning of name hiding. The value is the argument.

The accessors to name objects are the pseudo functions FIRSTNAME and LASTNAME. Their argument is a name and their value is the id first or last name.

```
FIRSTNAME("ABC$XYZ)  is  ABC
LASTNAME("ABC$XYZ)  is  XYZ
```

These forms may not appear on the left side of an assignment form.

The functions,

```
NAME FUNCTION LOCKNAME(NAME)
NAME FUNCTION UNLOCKNAME(NAME)
```

are used to protect and unprotect, respectively, a name from inadvertent redeclaration or redefinition. Their value is their argument. Most functions with the last name, CRISP,

that are used by the system have been protected by LOCKNAME.

### Numeric primitives

Most allocation of and access to numerical objects is implicitly through use of various operators such as + or COS. The few explicit primitives are used for creating and accessing complex numbers. The function,

COMPLEX FUNCTION CMPLX(FLOAT,FLOAT)

creates a complex number. The first argument is the real part and the second argument is the imaginary part. The value of

CMPLX(18.5,-13.6) is {COMPLEX 18.5 -13.6}

The accessors of complex objects are the pseudo functions REAL and IMAGINARY. Their argument is a complex object and their value is a float object.

REAL({COMPLEX 18.5 -13.6}) is 18.5

IMAGINARY({COMPLEX 18.5 -13.6}) is -13.6

These forms may not appear on the left side of an assignment form.

### Boolean primitives

There are no allocators or accessors for boolean objects.

### Handle primitives

The primitives for manipulating handle objects are described in the section on processors and processes (page 173).

Table I
INITIAL FIELD VALUES

| FIELD TYPE | CREATEBLANK | CREATE |
|---|---|---|
| ID | NIL | $'' |
| CHAR | %%00 | %%00 |
| NODE0 | NIL | NIL |
| NODE1 | NIL | {NODE1 NIL} |
| .. | NIL | .. |
| NODE8 | NIL | {NODE8 NIL ... NIL} |
| NAME | NIL | NIL |
| PROC[1] | NIL | trap[2] |
| FUNC[1] | NIL | trap[2] |
| VARB[1] | NIL | NIL |
| BOOL | NIL | NIL |
| NUMBER | NIL | 0 |
| FLOAT | 0.0 | 0.0 |
| COMPLEX | NIL | {COMPLEX 0.0 0.0} |
| INT | 0 | 0 |
| BYTE | 0 | 0 |
| HALF | 0 | 0 |
| HANDLE | NIL | NIL |
| ARRAY | NIL | NIL |
| ARRAY[1] | NIL | ARRAY[3] |
| NTUPLE | NIL | NIL |
| NTUPLE[1] | NIL | NTUPLE[3] |

[1] Subspecified type.
[2] Error trap routine.
[3] Array or ntuple allocated by CREATE.

Array and ntuple primitives

The section on item referencing and subscripts (Page 88) describes the mechanism for accessing fields in arrays and ntuples. The two forms used to allocate arrays and ntuples are CREATE and CREATEBLANK. Table I shows the initial value, given each type of field, when the structure is allocated. The argument to CREATEBLANK or CREATE is a

<type-ref>; the value is a new structure of the specified kind. If the <type-ref> is not a subspecified array or ntuple type, the value is NIL, 0, or 0.0, as appropriate. Also, if the argument is an array type with any *'s in its outer dimension, then the value is NIL.

CREATEBLANK operates much faster than CREATE. When the former is used, the structure is allocated and zeroed. This makes the pointer fields NIL. CREATE, on the other hand, examines the type of each field and initializes it to a default value as shown by the table. Because subarrays and ntuples are allocated, a problem could arise with recursive definitions. The tactic adopted is simple; when CREATE runs into a field that involves a recursive definition, it uses the value that would have been generated by CREATEBLANK, a NIL. For instance, given the declaration,

```
DEC A<X INT, Y B, Z C>,
    B<M ID, N NODE>,
    C<Q FLOAT, R A>;
```

The value of CREATE(A) is

```
{A$USER 0 {B$USER 0'' (NIL)} {C$USER 0.0 NIL}}
```

and the value of CREATEBLANK(A) is

```
{A$USER 0 NIL NIL}
```

## Presets

Variables in a <declare> form and variables bound by blocks may have a preset expression associated with them. The preset may be omitted, be an explict expression, or be *

(which means use CREATEBLANK to compute the initial value). Table J gives the initial value of a variable for each of its possible type attributes when the preset is either omitted or is *. Recall that every time a block is entered, the presets for its variables are recalculated (except for own variables whose presets are calculated once, at compile time). The real use of the * option is to cut down the amount of writing when allocating arrays and ntuples. To allocate using CREATE instead of CREATEBLANK, call CREATE explicitly as a preset expression. For instance, after compilation of

```
DEC INT ARRAY(3) ARRAY(2) A:=CREATE(A);
```

the value of A is

```
{INT ARRAY(3) ARRAY(2)
  {INT ARRAY(3) 0 0 0}
  {INT ARRAY(3) 0 0 0}}
```

If * had been used instead of CREATE(A), the value of A would be

```
{INT ARRAY(3) ARRAY(2) NIL NIL}
```

Table J
INITIAL VARIABLE VALUES

| TYPE | PRESET FORM | |
|---|---|---|
| | OMITTED | * |
| ID | NIL | $'' |
| CHAR | NIL | %%00 |
| NODE0 | NIL | NIL |
| NODE1 | NIL | {NODE1 NIL} |
| .. | NIL | .. |
| NODE8 | NIL | {NODE8 NIL ... NIL} |
| NAME | NIL | NIL |
| PROC[1] | NIL | trap[2] |
| FUNC[1] | NIL | trap[2] |
| VARB[1] | NIL | NIL |
| BOOL | NIL | NIL |
| NUMBER | NIL | 0 |
| FLOAT | 0.0 | 0.0 |
| COMPLEX | NIL | {COMPLEX 0.0 0.0} |
| INT | 0 | 0 |
| HANDLE | NIL | NIL |
| ARRAY | NIL | NIL |
| ARRAY[1] | NIL | ARRAY[3] |
| NTUPLE | NIL | NIL |
| NTUPLE[1] | NIL | NTUPLE[3] |

[1] Subspecified type.
[2] Error trap routine.
[3] Array or ntuple allocated by CREATEBLANK.

# EXPRESSIONS

Expressions are the basic building blocks of CRISP programs. They are used anywhere a value is required or needed. There is also a class of no-value expressions, e.g. the invocation of a function with NOVALUE <value-type>. No-value expressions may appear as the non-final forms in a <multi-expression-block> or as statements. Any expressions may also be used as statements for side effects. The syntax of expressions is:

```
*SL*

<expression>::=$=<infix-operator>=<operand>

<operand>::=<constant>|<locative>|<control>|<block>|
            <for-loop>|<typep>|<special-operand>

<locative>::=<name>|<field>|<machine-field>

<machine-field>::=<byte>|<core>|<cheat>

<byte>::=BYTE(<integer-expression>,<integer>,
                <integer-expression>)

<core>::=CORE(<integer-expression>[ ,<integer>])

<cheat>::=CHEAT(<type-ref>,<type-ref>,<expression>)

<control>::=<conditional>|<processing-primitive>|
            <function-call>

<function-call>::=<func-expression>($<expression>)|
                  <pseudo-func-name>($<expresion>)

<special-operand>::=<drive>|<prefix-operand>|
                    <IL-form>|<CAP-form>

<drive>::=DRIVE(<type-ref>,<expression>)

<prefix-operand>::=¬<operand>|
                   {+|-|ABS|NABS}<number-operand>

<IL-form>::=IL <external-data>

<CAP-form>::=CAP {<type-ref>|NOVALUE}
```

```
                $e ;=<CAP-SL-form>
                [; ] END
```

```
*IL*

<expression>::=<constant>|<locative>|<control>|<block>|
               <for-loop>|<typep>|<special-operand>

<locative>::=<name>|<field>|<machine-field>

<machine-field>::=<byte>|<core>|<cheat>

<byte>::=(BYTE <integer-expression><integer>
               <integer-expression>)

<core>::=(CORE <integer-expression>[<integer>])

<cheat>::=(CHEAT <type-ref><type-ref><expression>)

<control>::=<conditional>|<processing-primitive>|
            <function-call>

<function-call>::=(<func-expression> $<expression>)|
                  (<pseudo-func-name> $<expression>)

<special-operand>::=<drive>|<prefix-operand>|<CAP-form>

<drive>::=(DRIVE <type-ref><expression>)

<prefix-operand>::=(NOT <expression>)|
                   ([MINUS|ABS|NABS] <number-expression>)

<CAP-form>::=(CAP [<type-ref>|NOVALUE] $<CAP-IL-form>)
```

The SL and IL syntaxes are very different. In IL there is no concept of an operand as opposed to an expression. In SL, the concept of an operand arises because of the use of infix operators and their relative binding strengths. The following sections describe the SL precedence scheme and various kinds of SL operands (IL expressions) that are referenced by the syntax. The operands not described below are detailed in other sections of this document.

## SL Infix Expressions

If an expression consists of more than one operand, then it is a sequence of operands separated by infix operators. For example, in the form

A*B+C/D

A, B, C, and D are the operands and *, + and / are the infix operators. Each infix operator, which must be an identifier or sequence of delimiters, is defined by a 4-tuple. The fields are: (1) the left binding strength, (2) the right binding strength, (3) the indef flag, and (4) the IL (prefix) form operator. Table K gives the values of these fields for each of the SL infix operators.

The left and right strengths are used to determine to which operator an operand "belongs". The algorithm that fully parenthesizes an input expression is straightforward. Basically, the operand - operator sequence is input until an operator is found whose left strength is less than the right strength of the previous operator. When this happens, the last two operands and their included operator are enclosed in parentheses and act as a single operand for the duration of this processing. When no more operators are found, the remaining operands are parenthesized in a right to left order. Whenever possible, the operands of an operator that is marked indef are strung together. (This can happen only if the intervening operators are all of higher strength.) Thus,

A+B*C+D is (A+(B*C)+D)

Table K

SL INFIX OPERATOR DEFINITIONS

| OPERATOR | RIGHT | LEFT | INDEF | PREFIX |
|---|---|---|---|---|
| := | 1000 | 0 | | SET |
| ** | 901 | 900 | | EXP |
| * | 810 | 810 | X | TIMES |
| / | 800 | 801 | | QUO |
| // | 800 | 801 | | IQUO |
| + | 700 | 700 | X | PLUS |
| - | 710 | 711 | | DIFFER |
| && | 620 | 620 | X | BAND |
| \|\| | 610 | 610 | X | BOR |
| BXOR | 600 | 600 | X | BXOR |
| > | 500 | 500 | | GR |
| ¬> | 500 | 500 | | LQ |
| < | 500 | 500 | | LS |
| ¬< | 500 | 500 | | GQ |
| >= | 500 | 500 | | GQ |
| ≥ | 500 | 500 | | GQ |
| ¬>= | 500 | 500 | | LS |
| <= | 500 | 500 | | LQ |
| ≤ | 500 | 500 | | LQ |
| ¬<= | 500 | 500 | | GR |
| INTER | 490 | 490 | | INTER |
| UNION | 481 | 480 | | UNION |
| @ | 471 | 470 | | APPEND |
| @@ | 471 | 470 | | DAPPEND |
| ## | 465 | 465 | X | CONS |
| # | 460 | 461 | | CONS |
| IN | 301 | 300 | | IN |
| ¬IN | 301 | 300 | | NIN |
| ON | 301 | 300 | | ON |
| ¬ON | 301 | 300 | | NON |
| = | 200 | 200 | | EQ |
| ¬= | 200 | 200 | | NQ |
| == | 200 | 200 | | EQUAL |
| ¬== | 200 | 200 | | NEQUAL |
| ¬& | 150 | 150 | X | NAND |
| & | 140 | 140 | X | AND |
| ¬\| | 130 | 130 | x | NOR |
| \| | 120 | 120 | X | OR |
| >> | 110 | 111 | | IMPLY |
| << | 110 | 111 | | IMPLIED |

but

A*B+C*D is ((A*B)+(C*D))

Consider the expression

A*B/C*D##E##F+G*H+I#J

The following lines show the steps in processing the above (refer to Table K for the operators' binding strengths).

```
A*B/
(A*B)/
(A*B)/C*D##
(A*B)/(C*D)##
((A*B)/(C*D))##
((A*B)/(C*D))##E##F+G*H+
((A*B)/(C*D))##E##F+(G*H)+
((A*B)/(C*D))##E##F+(G*H)+I#
((A*B)/(C*D))##E##(F+(G*H)+I)#
(((A*B)/(C*D))##E##(F+(G*H)+I))#
(((A*B)/(C*D))##E##(F+(G*H)+I))#J
((((A*B)/(C*D))##E##(F+(G*H)+I))#J)
```

After the expression is completely parenthesized, the operators are replaced with their prefix equivalents and moved to the first (operator) position in the list. This produces an IL expression. Thus, the IL for the above example is

```
(CONS
     (CONS (QUO (TIMES A B) (TIMES C D))
           E
           (PLUS F (TIMES G H) I))
     J)
```

Note, the inner CONS (because it is marked indef) ends up with three arguments in this instance. Therefore, it will produce a node3 object when it is executed. The operator, #, is not marked indef, so it will always grab precisely two operands and produce a node2 object when it is executed.

The _ used to make field forms and the $ used to make global names appear to be infix operators but are not. Both are

associated with specialized syntax and meanings and are therefore handled separately. The interpretation of

A$B_C_D[I]_E(J)_F

is

{{{A$B_C_D_E[I]}(J)}_F}

Presumably, the following is true: A$B is a variable with an ntuple value. {A$B_C_D_E[I]} is a field that is accessed (in A$B) with a single subscript, I. The value of that field is a func that takes one argument -- in this instance, J. The value of the function call is an ntuple and the field whose item name is F is being accessed in that ntuple. Thus, A$B_C_D[I]_F(J) is an ntuple expression and the entire expression is a field. (It may, therefore, appear on the left side of an assignment form.)

## Locatives and Assignments

An assignment form is used to set (change) the value of a variable or a field. In SL, an assignment is specified with the infix operator, :=. In IL, the operator is SET. A <locative> is the syntax description of the legal forms that may appear as the left side (receiver) of an assignment. In operation, the right side of an assignment is evaluated, and its value is converted to the type of the left side. Then the value is set into the left side. The value type of an assignment form is the type of the left side. If both the left side and the right side are types with objects represented by pointers, then the type will be the most

restrictive. For instance, if either side is a character and the other side is an identifier, then the value is character.

Not all forms that are locatives syntactically can appear as receiving fields. All fields can, but the only names that can are variables. The <machine-field>s are described below along with the conditions under which they can appear as a receiver.

Byte

A <byte> form is used to reference a contiguous string of bytes in an integer object. The first argument specifies the offset in bytes from the left of the integer object. The second argument is the length of the byte string in bytes. Evaluation of the third argument produces the integer object in which the bytes are referenced. Some examples are:

```
BYTE(0,4,01020304X) is 01020304X
BYTE(0,1,01020304X) is 01X
BYTE(1,2,01020304X) is 0203X
BYTE(3,1,01020304X) is 04X
```

A <byte> form may be used as a receiving field if and only if (1) the third argument is a locative that can be a receiving field and (2) the type of the third argument is integer without any conversion. This specifically excludes the types, number and general. The value type of the form is integer.

Core

A <core> form treats memory as if it were a byte array. Any

core form may appear as a receiving field. The first argument is the byte address (subscript), and the second argument, which is optional, is the length of the field in bytes. If the second argument is omitted, it is assumed to be 4. The value type of a core form is integer.

Cheat

CHEAT evaluates its expression body and converts its value to the type of the first <type-ref>. Then, without further conversion, the value is assumed to be of the type specified by the second type ref. For instance,

CHEAT(FLOAT,INTEGER,1.0) is 40100000X

Similarly,

CHEAT(FLOAT,INTEGER,1) is 40100000X

because 1 is floated before it is cheated. Cheating from a type with pointer objects works on the pointer. Similarly, cheating to a type with pointer objects creates a pointer. Thus, if G is a variable with type attribute general, then the value of

CHEAT(GEN,INT,G)

is the byte address of the structure pointed at by G. Using core, cheat, and byte forms together, many language forms may be simulated. For instance, CDR, the second field of a node2 object, could be represented as

CHEAT(INT,GEN,CORE(CHEAT(NODE2,INT,x)+4))

where x is an arbitrary expression that produces a value that can be converted to node2. The value of the inner cheat is the byte address of the node2 object. The four byte field starting four bytes to the right in that object

is the CDR field. The core form references this field, and the outer cheat specifies that the type of the field is general.

A <cheat> form may be used as a receiving field whenever (1) its third argument is a locative that may appear as a receiving field and (2) the value of the third argument is the same as the first <type-ref> without conversion. Thus, the above equivalent of CDR is a legal receiving field, as is CDR itself.

WARNING: Use of BYTE, CORE, and CHEAT can lead to unrecoverable program checks. Their use is intended only for the programmer who is very knowledgable in the system storage conventions. Therefore, use them at your own risk.

## Function Calls

A <function-call> is the mechanism used to instantiate a function. As opposed to a process start, the activity remains in the currently active process. A function call evaluates its arguments and places their values on the stack. The <func-expression> is then evaluated, and the function, which is its value, is entered. At this point, the argument values are paired with the parameter names, creating variable bindings. The expression body of the function is then evaluated, the variables are unbound, and the value of the expression is returned as the value of the

call. The value returned is converted to the value type of the function (except when the value type is novalue). The order of argument evaluation is not guaranteed.

A <func-expression> is any expression whose value type is known to the compiler as a subspecified type of function or func. The simplest examples are a function name or a variable with a subspecified func type attribute. A function call with a pseudo func name as its operator resembles a standard function call in most respects. The name may be a macro or transform name, or the name of some pseudo function that the compiler handles specially. Some examples are TIMES, ABS, and QUO, for which the compiler may produce open code. Many forms in the language, for which the syntax has been specifically spelled out, could be handled as function calls with pseudo func name operators. When this has not been done, it has been to present a more organized description.

All arguments to functions and processors are passed by value. Assume that this function definition has been made:

```
NOVALUE FUNCTION FOO(INT I, INT ARRAY(*) A)
                    (I:=1, A[I]:=5);
```

Then operate this block:

```
BEGIN INT J:=4, INT ARRAY(4) B:=*;
      FOO(J, B);
      PRINT(J);
      PRINT(B);
      END;
```

The printed value of J is 4, and the printed value of B is (INT ARRAY(4) 5 0 0 0). When values are passed to a function or processor they are copied. However, a value

that is an array, ntuple, node, identifier, etc., is a pointer at a structure; therefore, the thing copied is the pointer, not the pointee. When FOO is entered, the variable A in FOO and the variable B in the block point at the same array. But an assignment form such as "A:=array expression" in FOO has no effect on B in the block. It merely "points" A at another array. The call by value technique used in CRISP is identical to the argument passing regime used in LISP: it has sometimes been called the weak form of call by copy. In CRISP there is no equivalent of call by location (or call by reference). Call by name can be simulated by using the process control primitives in combination with func, proc, and handle arguments that are passed by value.

## Special Operands

The operands described in this section are a miscellaneous collection that are not described elsewhere or forms that have a special syntax.

### Drive

A <drive> form evaluates its argument and converts it to the type specified by the type ref. If error checking mode is not being used, then no run time check is made when an object in a type represented by pointers is converted to any other type with objects that are represented by pointers. Some examples of <drive> forms are:

DRIVE(INT,1) is 1

DRIVE(FLOAT,1) is 1.0

DRIVE(FLOAT,3.5) is 3.5

DRIVE(INT,3.5) is 3

DRIVE(FLOAT,DRIVE(INT,3.5)) is 3.0

All type conversion that is performed in the system is done either explicitly or implicitly by a <drive> form.

Not

NOT returns TRUE if its argument is NIL and returns NIL otherwise. In SL, the prefix operator, ¬, applies only to the next operand. Thus,

¬A&B is (AND (NOT A) B) in IL

To "negate" the entire conjunction, write

¬(A&B)

Arithmetic prefix operands

The operators +, -, ABS, and NABS apply to the following operand. If it is desired to apply them to a more complicated expression, then enclose that expression in parentheses, making it into a single operand. The unary + and - do the usual thing. ABS returns the absolute value of its argument and NABS is equivalent to -ABS. The type of value produced by unary + and - is the same type as its argument. The value produced by ABS and NABS is the same type as its argument except when the argument is complex. In this case, the value is the norm (or negative norm) of the argument and is an object of type float.

CAP and IL forms

<CAP-form>s and <IL-form>s are used to insert forms written in CAP or IL into programs that are written in SL and IL. The IL operand is an external datum, normally a node2 object, that is used as an operand in the SL program without further translation. A <CAP-form> may be used to drop into machine language at any point in an SL program at which an operand is expected and at any point in an IL program at which an expression is expected. The <type-ref> in a <CAP-form> is the type of value that is returned by the machine code. The value is assumed to be in register F0 if floating, R5 otherwise. An SL or IL program may not branch into a CAP form. Also, a CAP form must not branch to labels placed by the higher level language form. This restriction will be lessened after the proper communication with the compiler's flow analysis has been worked out.

## Order of Evaluation

Unlike LISP and many other languages, CRISP guarantees order of evaluation only in a few places. This section summarizes the rules that pertain to ordering. See the sections that specifically describe the particular forms for more information.

Order of evaluation is guaranteed when:

(1) Preset computations for block variables are calculated in the left to right order of their appearance.

(2) Statements (or expressions) appearing as the top level forms in blocks are evaluated in their order of occurrence except as modified by <go>, <leave>, <return>, and <exit> forms.

(3) AND, OR, NAND, and NOR evaluate their arguments in a left to right order. Evaluation ceases as soon as the value of the form is known.

(4) The generators of a FOR loop are evaluated in their order of occurrence except as modified by conditional generators.

(5) Conditionals evaluate their embedded forms in the order dictated by their definitions. For instance, the predicate in an if form is evaluated before the evaluation of the then or else clause selected by the value of the predicate.

Some specific areas where order of evaluation is not guaranteed are:

(1) The order of evaluation of subscript expressions is not guaranteed in a field form.

(2) The order of evaluation of function and processor arguments is not guaranteed.

(3) The order of evaluation and combination of expressions involving arithmetic is not guaranteed.

# CONDITIONALS

This section describes several forms that allow for conditional evaluation. In all these forms, one of several embedded forms is selected for evaluation. The form selected depends upon the results of predicate tests. The kind of test depends upon the kind of <conditional> form used. The embedded forms are statements if the parent form is a statement and are expressions if the parent form is an expression. The embedded forms are called terminal computations or, more simply, terminals. The type of value produced by a conditional expression is the minimal type that contains the values of all the terminals. See Figure D (page 49). The syntax of <conditional>s is:

```
*SL*

<conditional>::=<if>|<select>|<selectq>|<selectn>|<selectt>

<if>::=[IF|WHEN] <predicate> THEN <form>
          $[[OR IF|WHEN] <predicate> THEN <form>]
          [ELSE <form>]

<predicate>::=<boolean-expression>

<select>::=SELECT <expression>
             $[WHEN ≠=,=<expression> THEN <form>]
             [ELSE <form>]

<selectq>::=SELECTQ <expression>
              $[WHEN ≠=,=<external-data> THEN <form>]
              [ELSE <form>]

<selectn>::=SELECTN <number-expression>
              $[WHEN ≠=,=<nselx> THEN <form>]
              [ELSE <form>]

<nselx>::=<number-expression>|
          [≥|≤|=|¬=|>=|<=|<|>]<number-expression>|
```

```
          [≥|≥=]<number-expression>
                {TO|THRU}<number-expression>

<selectt>::=SELECTT {<name>|<identifier>:=<expression>}
              $(WHEN {<type-ref>|NIL} THEN <form>)
              [ELSE <form>]


*IL*

<conditional>::=<if>|<select>|<selectq>|<selectn>|<selectt>

<if>::=(IF <predicate> <form> [<form>])

<predicate>::=<boolean-expression>

<select>::=(SELECT <expression>
                   $((≠<expression>) <form>)
                   <form>)

<selectq>::=(SELECTQ <expression>
                     $((≠<external-data>) <form>)
                     <form>)

<selectn>::=(SELECTN <number-expression>
                     $((≠<nselx>) <form>)
                     <form>)

<nselx>::=<number-expression>|
          ({EQ|NQ|GQ|LQ|GR|LS} <number-expression>)|
          ({THRU|TO} [GR|GQ] <number-expression>
                             <number-expression>)

<selectt>::=(SELECTT {<name>|(<identifier> <expression>)}
                     $({<type-ref>|NIL} <form>)
                     <form>)
```

For all the (SL) <conditional> forms there is an optional else clause. If omitted, then the clause "ELSE NIL" is assumed. Also, nested conditionals with some of the else clauses omitted could be ambiguous. The SL to IL translator uses the ALGOL rule in these instances; namely, when an else clause is found, it is attached to the most deeply nested conditional to which it could belong. For instance,

```
IF IF P THEN A THEN IF Q THEN B ELSE C
```

is interpreted as

```
IF (IF P THEN A) THEN (IF Q THEN B ELSE C)
```

which also could have been written

```
IF P&A THEN (IF P THEN B ELSE C) ELSE NIL
```

## IF

<if> forms provide the normal if-then and if-then-else conditional primitives. In the following, p1 ... pn are predicates and f1 ... fn are forms. <predicate>s are expressions that are evaluated to determine a boolean value. If the value of a predicate is anything other than NIL, then the value will be treated as if it were TRUE. The meaning of

```
IF p1 THEN f1
```

is that if the value of p1 is TRUE then evaluate f1. If the if is an expression, then the value of f1 is the value of the if. If the if is a statement and f1 is not a goto, a leave, or a return, then after f1 is evaluated, control falls through the if. If the value of p1 is NIL, then nothing more is evaluated. If the if is a statement, then control falls through. Otherwise, the if is an expression and its value is NIL. The form,

```
IF p1 THEN f1 ELSE f2
```

behaves identically to the first example when the value of p1 is TRUE. When the value of p1 is NIL, then f2 is evaluated. If the if is a statement and f2 is not a goto, a leave, or a return, then after f2 is evaluated, control falls through the if. Otherwise, the if is an expression and the value of f2 is the value of the if. Recall that

these two forms are equivalent.

IF p1 THEN f1 and IF p1 THEN f1 ELSE NIL

Therefore, we can talk as if all <if> forms have both a then and an else clause. Thus, an if evaluates either the then or the else clause, not both. The form evaluated depends upon the value of the predicate; the else clause is evaluated if the predicate is NIL and the then clause is evaluated otherwise.

It is natural to write nested if forms using an indentation scheme like the following.

```
IF p1 THEN f1
   ELSE IF p2 THEN f2 ELSE f3

IF p1 THEN f1
   ELSE IF p2 THEN f2
            ELSE IF p3 THEN f3 ELSE f4
```

However, with long or complicated forms, one soon runs off the right side of the listing. As a lexical convenience, several alternative methods of representing if-then and if-then-else logic are provided. The above two examples can be rewritten as

```
WHEN p1 THEN f1      |     IF   p1 THEN f1
WHEN p2 THEN f2      |     ORIF p2 THEN f2
          ELSE f3    |              ELSE f3

WHEN p1 THEN f1      |     IF   p1 THEN f1
WHEN p2 THEN f2      |     ORIF p2 THEN f2
WHEN p3 THEN f3      |     ORIF p3 THEN f3
          ELSE f4    |              ELSE f4
```

Also, the various forms could be mixed as in

```
WHEN p1 THEN f1
ORIF p2 THEN f2
          else f3
```

which is equivalent to the first forms. The WHEN and ORIF keywords (appearing where ELSE could appear) are just

equivalents to the two-word sequence, ELSE IF. They give the language no new power, rather they provide a "pretty" format for involved conditional logic. However, there is a case to watch out for. Contrast the meaning of these two examples.

```
BEGIN;                    |     BEGIN;
  WHEN p1 THEN f1         |       WHEN p1 THEN f1;
  WHEN p2 THEN f2         |       WHEN p2 THEN f2
  END                     |       END
```

The only difference is the semicolon appearing after f1 in the second example. The first block has one statement in its body. The second block has two separate statements in its body. In the first case, either f1 or f2 or neither will be evaluated. In the second case, it is possible for both f1 and f2 to be evaluated. Thus, a very subtle error may be introduced by an added or a deleted semicolon.

## SELECT and SELECTQ

<select> and <selectq> forms are very similar. They have an embedded expression called their selectrix. The selectrix is evaluated precisely once per evaluation of its parent form. The when clauses are introduced by the word WHEN and followed by one or more expressions. In the case of <selectq>, the expressions are external data that are implicitly quoted. The expressions in the when clauses are called selectors.

The evaluation of a select or a selectq proceeds along the following lines: Evaluate the selectrix and call its value

x. Evaluate the first selector in the first when clause. If its value is EQ (=) to x, then perform the terminal associated with the first when clause (i.e., the first then clause). Otherwise, evaluate the second selector in the first when clause and see whether its value is EQ to x. Continue in this manner with all selectors in the first when clause. If none are EQ, then go on to the second when clause. If any are EQ to x, then perform the second terminal, etc. If none of the selectors are EQ to the selectrix, then evaluate the else clause. If the select or selectq is used as an expression, then the value is the value of the evaluated terminal. If the select or selectq is used as a statement and the evaluated terminal is not a goto, a leave, or a return, then control falls through the whole form.

In the following, f1, f2, and f3 are forms, and e is an expression. These two are identical in meaning and interpretation:

```
SELECTQ e                  |   SELECT e
  WHEN A,B THEN f1         |     WHEN "A,"B THEN f1
  WHEN QR  THEN f2         |     WHEN "QR   THEN f2
  WHEN 17  THEN f3         |     WHEN 17    THEN f3
                           |                ELSE NIL
```

If the value of e is the identifier A or B, then f1 is evaluated. Otherwise, if the value of e is the identifier QR, then f2 is evaluated. Otherwise, if the value of e is 17, then f3 is evaluated. In all other cases, NIL is evaluated.

When one of the terminal forms is selected, its evaluation

terminates the evaluation of the select or the selectq form.

## SELECTN

A <selectn> form provides an easy method of conditionally selecting a terminal for evaluation depending upon the value of a number expression. A number expression must have the type integer, float, complex, number, or general. As with select and selectq forms, the embedded expression is called the selectrix, and the clauses in the when clause are called selectors. Let x be the value of the selectrix and n1 and n2 be number expressions. Then the following lists the possible forms of the selectors and their meanings.

```
n1              eval then clause iff x=n1
>n1             eval then iff x>n1
<n1             eval then iff x<n1
=n1             eval then iff x=n1
¬=n1            eval then iff x¬=n1
>=n1            eval then iff x≥n1
<=n1            eval then iff x≤n1
n1 TO n2        eval then iff n1≤x<n2
n1 THRU n2      eval then iff n1≤x≤n2
>n1 TO n2       eval then iff n1<x<n2
>n1 THRU n2     eval then iff n1<x≤n2
>=n1 TO n2      eval then iff n1≤x<n2
>=n1 THRU n2    eval then iff n1≤x≤n2
```

The selectrix is evaluated only once. The selectors are evaluated in the order in which they appear, left to right in the first when clause, then left to right in the second when clause, etc. As soon as one selector is satisfied, the corresponding terminal (then clause) is evaluated. If no selector is satisfied, then the else clause is evaluated.

```
SELECTN X
  WHEN 3, 7 THRU 9 THEN f1
  WHEN <0          THEN f2
                   ELSE f3
```

In this example, if the value of X is 3, 7, 8, or 9, then f1 is evaluated. If the value of X is negative, then f2 is evaluated. Otherwise, f3 is evaluated. Exactly one of the terminals is selected for evaluation. If the selectn is used as an expression, then the value is the value of the selected terminal.

## SELECTT

A <selectt> form picks one of its terminals for evaluation depending upon the type of its selectrix. The selectrix is either a variable or an assignment form with an identifier as its left side. If the selectrix is a variable name, then the selectors (<type-ref>s or NIL) are more specific types. For instance, if G is a general variable, then the following would be typical.

```
SELECTT G
  WHEN ID   THEN f1
  WHEN NODE THEN f2
            ELSE f3
```

When a terminal is evaluated, the type attribute of the variable is known to be the <type-ref> selector. This may be particularly useful when "recovering" an ntuple or array object from the type general.

If the second form of the selectrix is used, then the value of the expression is put in the named local variable, which is automatically bound while in the selectt form. The variable is visible only while the terminal forms are being evaluated. The type attribute of the variable in a

particular terminal is picked up from the corresponding selector. In the else clause, the variable is considered of type general. If the selector is NIL, then the variable is not visible. Thus, in the above example, the type attribute in f1 is id, in f2 the attribute is node, and in f3 it is general. It is not legal to branch into a terminal of a <selectt>.

# FOR LOOP

The CRISP <for-loop> form permits easy, concise representation of iterative computations. The loop functions provided not only can be used for numeric value generation, but also can manipulate elements of lists, ntuples, or other CRISP data types. For loops are organized so that any number of loop variables can be updated during each iteration, and update functions can be executed conditionally, individually, or in groups. Multiple loop bodies containing any valid CRISP statements are permitted, resulting in a powerful computational tool.

For loops can be used as either statements or expressions. A mechanism is provided for computing a value (or values) upon normal termination of the loop. The For loop may be used in both SL and IL and has the following form (a formal description appears at the end of this section):

```
*SL*
     FOR
       {1 or more for-forms}
     ENDF

*IL*
     (FOR {1 or more for-forms})
```

The For loop is implemented as a macro. It is expanded as a <binding-block>, which has the following form:

```
DO set-old-value-1; set-old-value-2; ...; set-old-value-n;
   initial-1; initial-2; ...; initial-n;
   BEGIN
      bind-variable-1,bind-variable-2,...,bind-variable-n;
      t; code-for-first-generator;
      code-for-second-generator;
              .
              .
              .
      code-for-n'th-generator;
      GOTO t;
      r; final-1; final-2; final-3; ...; final-n;
      RETURN returned-value-computation;
      END
   END
```

Where:

'set old value i' indicates presetting of any loop variables whose bindings exist outside the loop.

'initial i' indicates statements used in INITIAL generators.

'bind variable i' indicates binding (and optional presetting) of new loop variables.

'code for i'th generator' indicates code needed for a given generator, conditional, or loop body. These pieces of code appear in the same order as the generators.

'final i' includes statements that appear in FINALLY generators.

'returned value computation' indicates code necessary to generate values for value-producing generators. If more than one value-producing generator is used, all the values are returned as a (node2) list in whatever order the generators producing them appear in the loop.

Any of the above for loop components not necessary for a given loop will not be included.

## Loop Termination

There are two methods of leaving or terminating a for loop; normal and abnormal. Normal termination occurs when a generator that is capable of terminating the loop does so, or when a LEAVE FOR form is executed in a loop body. Normal termination causes the normal ending and value-producing code, if present (at label r above), to be executed. Abnormal termination occurs when, in a loop body, a return or leave is executed or a go leaves the loop. In these cases, computation normally performed at the end of the loop is bypassed, and the user is responsible for generation of any value(s) desired.

## Generator Descriptions

The for loop generators and conditionals are described below. In the syntax and expansion descriptions, the following symbols are used:

| SYMBOL | MEANING |
|---|---|
| v | a variable (identifier or global name) |
| exp1 ... expn | expressions |
| r | a genid label placed before the code that is executed upon normal loop termination (value computation, etc.) |
| t | a genid label placed before the first statement of executable code. |
| g,g1 ... gn | genid variables or labels used in expansion of the loop. |

| | |
|---|---|
| forv | a loop variable carrying optional type information[1] |
| numv | like forv, but can only be a simple numeric type |

If a lower-case 'if' followed by a capitalized word appears in a description, the action(s) immediately following is taken only if the capitalized word appears in the form. This is used, for example, when the code produced by a generator varies depending upon the presence of optional keywords. In addition, the function TESTANDSET is used in some generator code. Although implemented as a single System/370 instruction, it may be viewed as the following transform:

```
TRANSFORM TESTANDSET(FOO)
 IF FOO THEN TRUE
        ELSE (FOO:=TRUE,NIL);
```

This has the effect of bypassing the consequent of an if (using TESTANDSET as its predicate) the first time the if is executed. Subsequent executions of the if will execute the consequent.

In the following descriptions, things following INITIALIZATION are normally bound, optionally with explicit type attributes and presets. Some initialization may not involve binding, but merely setting old variables. If this is the case, it will be indicated. ENDING values are returned upon normal exit from the loop except in the case of FINALLY.

---

[1] in SL, this appears as [OLD|<type-ref>] v
in IL, the form is ([OLD|<type-ref> v) or just v

## Generators Producing Values

**AND**

```
SL -- AND exp
IL -- (AND exp)

INITIALIZATION -- BOOL q := TRUE
GENERATOR -- IF ¬ exp THEN (q:=NIL, GOTO r)
ENDING -- q
```

TRUE is returned if exp is non-NIL for all evaluations. The loop terminates and returns NIL the first time exp evaluates to NIL.

**ALL**

```
SL -- ALL exp
IL -- (ALL exp)

INITIALIZATION -- BOOL q := TRUE
GENERATOR -- IF ¬ exp THEN q:=NIL
ENDING -- q
```

Like AND, but does not terminate the loop if exp evaluates to NIL.

**OR**

```
SL -- OR exp
IL -- (OR exp)

INITIALIZATION -- BOOL q
GENERATOR -- IF exp THEN (q:=TRUE, GOTO r)
ENDING -- q
```

The loop terminates and returns TRUE when exp first evaluates to non-NIL. If all evaluations of exp are NIL, NIL is returned.

**ANY**

```
SL -- ANY exp
IL -- (ANY exp)

INITIALIZATION -- BOOL q
GENERATOR -- IF exp THEN q:=TRUE
ENDING -- q
```

Like OR, but does not terminate the loop if exp evaluates to non-NIL.

FIRST
SL -- FIRST exp
IL -- (FIRST exp)

INITIALIZATION -- GENERAL q
GENERATOR -- if (q:=exp) THEN GOTO r
ENDING -- q

Like OR, but returns the value of exp the first time it evaluates to non-NIL.

VALUE
SL -- VALUE exp
IL -- (VALUE exp)

INITIALIZATION --
GENERATOR --
ENDING -- exp

Upon normal exit of the loop, exp is evaluated, and its value is returned.

SUM
SL -- [INT|FLOAT|NUMBER|COMPLEX] SUM exp
IL -- (SUM [INT|FLOAT|NUMBER|COMPLEX] exp)

INITIALIZATION -- [type] q :=0
GENERATOR -- q:=q+exp
ENDING -- q

Returns the sum of all evaluations of exp. If type is not specified, number is assumed.

PRODUCT
SL -- [INT|FLOAT|NUMBER|COMPLEX] PRODUCT exp
IL -- (PRODUCT [INT|FLOAT|NUMBER|COMPLEX] exp)

INITIALIZATION -- [type] q :=1
GENERATOR -- q:=q*exp
ENDING -- q

Returns the product or all evaluations of exp. If type is not specified, number is assumed.

UNION
```
SL -- UNION exp
IL -- (UNION exp)

INITIALIZATION -- NODE q
GENERATOR -- q:=exp UNION q
ENDING -- q
```

Returns the union of all evaluations of exp.

INTER
```
SL -- INTER exp
IL -- (INTER exp)

INITIALIZATION -- NODE q1,q2
GENERATOR -- IF TESTANDSET(q1) THEN q2:=exp INTER q
                               ELSE q2:=exp
ENDING -- q
```

Returns the intersection of all evaluations of exp.

DAPPEND
```
SL -- DAPPEND exp
IL -- (DAPPEND exp)

INITIALIZATION -- NODE q
GENERATOR -- q:=q @@ exp
ENDING -- q
```

Builds a list using DAPPEND. Each evaluation of exp is DAPPENDed onto the previous list (initially NIL), and the constructed list is returned upon normal exit of the loop. Exp must evaluate to a list.

DAPPENDR
```
SL -- DAPPENDR exp
IL -- (DAPPENDR exp)

INITIALIZATION -- NODE q
GENERATOR -- q:=exp @@ q
ENDING -- q
```

Like DAPPEND, but returned list will have components (evaluations of exp) put together in reverse order. This form is more efficient computationally if order of list elements is not important.

**APPEND**
SL -- APPEND exp
IL -- (APPEND exp)

INITIALIZATION -- NODE q
GENERATOR -- q:=q@@(exp@NIL)
ENDING -- q

Builds a list like DAPPEND but uses APPEND.

**APPENDR**
SL -- APPENDR exp
IL -- (APPENDR exp)

INITIALIZATION -- NODE q
GENERATOR -- q:=exp @ q
ENDING -- q

Builds a list like DAPPENDR but uses APPEND.

**INITIALLY**
SL -- INITIALLY statement
IL -- (INITIALLY $statement)

Causes insertion of specified statement(s) before the binding of variables for the FOR loop. If more than one INITIAL is used, the statements are executed in the order of appearance.

**FINALLY**
SL -- FINALLY statement
IL -- (FINALLY $statement)

INITIALIZATION --
GENERATOR --
ENDING -- (evaluate but do not return) statement

Causes the statement(s) to be evaluated upon normal loop exit. Evaluation is done before any value computation.

COUNT
SL -- COUNT exp
IL -- (COUNT exp)

INITIALIZATION -- INTEGER q :=0
GENERATOR -- IF exp THEN q:=q+1
ENDING -- q

Returns the number of times exp evaluates non-NIL.

LIST
SL -- LIST exp
IL -- (LIST exp)

INITIALIZATION -- NODE q
GENERATOR -- q:=exp # q
ENDING -- DREVERSE(q)

A list of the values of each evaluation of exp is returned such that CAR(list) is 1st value, CADR(list) is 2nd value, etc.

LISTR
SL -- LISTR exp
IL -- (LISTR exp)

INITIALIZATION -- NODE q
GENERATOR -- q:=exp # q
ENDING -- q

Like LIST, but returns the list of values in reversed order. Computationally more efficient if ordering of elements is not required.

## Ordinary Generators

### IS

```
SL -- forv IS exp
IL -- (IS forv exp)

INITIALIZATION -- {[type] v}
GENERATOR -- v:=exp
```

V is set to the value of exp each time the generator is executed. If OLD appears, an existing binding of v is assumed and no new one is created. If type is not specified, implicit typing rules will apply.

### DO

```
SL -- DO block-body {END|ENDF}
IL -- (DO $statement)

INITIALIAZATION --
GENERATOR -- stat1
             stat2
               .
               .
               .
             statn
```

The specified statements are executed each time the DO generator is executed. Mulitiple DOs may appear in a loop. In SL, using ENDF instead of END terminates the DO and the for loop.

### BEGIN

```
SL -- BEGIN block-bind-list; [attribute-list]
          block-body {END|ENDF}
IL -- (BEGIN block-bind-list block-body)

INITIALIZATION --
GENERATOR -- BEGIN block-bind-list;
                   attribute-list;
             stat1
               .
               .
             statn
             END
```

Similar to DO, but permits binding of variables.

FOR
SL -- FOR loop
IL -- (FOR loop)

A for loop may be nested within another and used as a generator. The FOR keyword is used to free the user from coding
    DO FOR ..... ENDF END

IN
SL -- forv IN {exp | OLD var}
IL -- (IN forv {exp | OLD var})

```
INITIALIZATION -- [[type] v]
                    if not OLD in forv, NODE g := exp
GENERATOR -- IF NULL if OLD, var
                    otherwise, g THEN GOTO r
             if OLD with var, v:=CAR(var)
             otherwise, v:=CAR(g)
             if OLD with var, var:=cdr(var)
             otherwise, g:=cdr(g)
```

IN gives v (through successive iterations) the same values that MAPIN would present to its functional argument. Exp must evaluate to a list. When the list is exhausted, the loop is terminated. If the 'OLD var' form appears, a genid variable is not created to hold the successive CDRs. Rather, the existing variable, var, is used. Normally, this would result in var being NIL upon termination of the loop unless another generator causes termination before the list is exhausted. If OLD appears with the forv, an existing binding for v is assumed and a new one is not created. If no type is specified, implicit typing rules will apply.

ON
SL -- [OLD] v ON {exp | *}
IL -- (ON {v | (OLD v)} {exp | *})

```
INITIALIZATION -- NODE g :=exp [,NODE v]
GENERATOR -- IF NULL g THEN GOTO r
             v:=g
             g:=CDR(g)
```

Like IN, but sets v to successive CDRs of exp rather than CARs of successive CDRs. I.e., like MAPON instead of MAPIN. If OLD and * are used together, v itself is CDRed instead of a genid.

**RESET**

```
SL -- forv := {exp1 | *} RESET exp2
IL -- (RESET forv {exp1 | *} exp2)

INITIALIZATION -- if OLD, set v:=exp and bind q
                  otherwise, bind [type] v:=exp,q
GENERATOR -- IF TESTANDSET(q) THEN v:=exp2
```

The first time through the loop, v will have the value of exp1. On subsequent iterations, exp2 will be evaluated and its value stuffed in v. If OLD appears, an existing binding of v is assumed and a new one is not created. If no type is specified, implicit typing rules will apply. If OLD and * are used together, then v will have its original value on the first loop iteration.

**Stepper**

```
SL -- numv := {exp1 | *} [ {TO [ >=|<=|>|<|=|¬= ] | THRU} exp2 ]
                  [ {BY|STEP} exp3]
IL -- ({BY|STEP} numv {exp1 | *}
       [exp3 [ {exp2|({GQ|LQ|GR|LS|EQ|NQ} exp3)} ]])

INITIALIZATION -- if OLD, set v:=exp1
                          otherwise bind [type] v:=exp1
                  if TO, bind q:=exp2
                  if BY, bind q1:=exp3
                  bind q2
GENERATOR -- IF TESTANDSET(q2)
                THEN if BY, v:=v+q1
                     otherwise, if STEP, v:=v+exp3
                     otherwise, v:=v+1
             if TO, IF v specified-relational or GQ q
             if THRU, IF V GR q
                        THEN GOTO r
```

A standard arithmetic stepping loop. BY causes increment to be pre-computed; STEP causes exp3 to be evaluated each time. If neither BY nor STEP appears, 'BY 1' is assumed. TO causes ending test value to be pre-computed. If TO appears without specifying a relational, GQ is used. THRU causes a GR test to be made. To get ending value computed each time, use:

```
          UNTIL v desired-relational exp2
```

If OLD appears, an existing v is assumed and is merely set to exp1 rather than bound with that preset. If no type is specified, implicit typing rules will apply. If * is used with OLD, then v has its original outside value during the first loop iteration.

## Conditionals

### WHEN
SL -- WHEN exp
IL -- (WHEN exp)

GENERATOR -- IF ¬ exp THEN GOTO t

Whenever exp evaluates to NIL, the rest of this iteration of the loop is skipped.

### UNLESS
SL -- UNLESS exp
IL -- (UNLESS exp)

GENERATOR -- IF exp THEN GOTO t

Whenever exp evaluates to non-NIL, the rest of this iteration of the loop is skipped. (like 'WHEN ¬ exp')

### WHILE
SL -- WHILE exp
IL -- (WHILE exp)

GENERATOR -- IF ¬ exp THEN GOTO r

When exp first evaluates to NIL, the loop is terminated.

### UNTIL
SL -- UNTIL exp
IL -- (UNTIL exp)

GENERATOR -- IF exp THEN GOTO r

When exp first evaluates to non-NIL, the loop is terminated. (like 'WHILE ¬ exp')

IF
SL -- IF exp (generator | ($generator))
IL -- (IF exp $generator)

GENERATOR -- IF exp THEN (generator1,
generator2,
.
.
.
generatorn)

The specified generator(s) is executed only if exp evaluates to non-NIL. Note that the generator may be of the control type, as:

```
IF NUMBERP(X) UNLESS X <= 0
```

Since VALUE and FINALLY have no generator part, it is illegal to use them as consequents of IF.

BIND
SL -- BIND <block-bind-list>
IL -- (BIND . <block-bind-list>)

INITIALIZATION -- <block-bind-list>

The specified variables are added to the FOR loop's bind list. There is no facility for using attribute forms as in SL blocks.

FOR LOOP FORMS

| CRISP | SDC-LISP | INTERLISP |
|---|---|---|
| stepper (extended) | stepper | stepper |
| AND | | ALWAYS |
| ALL | | |
| OR | | |
| ANY | | |
| FIRST | | THEREIS |
| VALUE | VALUE | |
| UNION | | |
| INTER | | |
| COUNT | | COUNT |
| FINALLY | | FINALLY |
| LIST | LIST | COLLECT |
| LISTR | | |
| SUM | | SUM |
| PRODUCT | | |
| IS | LOOP/AT | (EACHTIME) |
| IN | IN | IN |
| ON | ON | ON |
| DO/BEGIN | TOP/BOTTOM/DO | DO/EACHTIME |
| RESET | RESET/NOW | |
| WHEN | WHEN | WHEN |
| UNLESS | UNLESS | UNLESS |
| UNTIL | UNTIL | UNTIL |
| WHILE | WHILE | WHILE |
| IF | | |
| DAPPEND | | JOIN |
| DAPPENDR | | |
| APPEND | | |
| APPENDR | | |
| BIND | BIND | |

A COMPARISON OF FOR GENERATORS FOR VARIOUS LISPS

Figure 1

## For Loop Examples

A sample code expansion (in SL) is included with each FOR loop example. Although the code produced in the examples is correct, it does not reflect certain optimizations that would be performed by the actual FOR macro (such as the collapsing of TESTANDSETs, if possible, and the removal of extraneous branching). In the examples, the following IMPLICITs are in effect:

```
     A-H  GENERAL
     I-N  INTEGER
and  O-Z  FLOAT
```

(1) Find the sum of a vector of float numbers:

```
*SL*
     FOR I:=1 THRU ABLN(X)
         FLOAT SUM X[I]
         ENDF

*IL*
     (FOR (BY I 1 1 (ARLN X))(SUM FLOAT (X (I))))
```

Expansion:

```
   BEGIN I:=1,q1:=ABLN(X),FLOAT q2,q3;
t:IF TESTANDSET(q3) THEN I:=I+1;
   IF I > q1 THEN GOTO r;
   q2:=q2+X[I];
   GOTO t;
r:RETURN q2;
   END;
```

(2) Print an endless list of Fibonacci numbers:

```
*SL*
     FOR I:=1 STEP J
         J:=1 RESET I-J
         DO PRINT(J)
         ENDF

*IL*
     (FOR (STEP I 1 J)(RESET J 1 (DIFFERENCE I J))
          (DO (PRINT I)))
```

Expansion:

```
   BEGIN I:=1,J:=1,q1,q2;
 t:IF TESTANDSET(q1) THEN I:=I+J;
   IF TESTANDSET(q2) THEN J:=I-J;
   PRINT(J);
   GOTO t;
   END;
```

(3) Produce a list of the first 10 prime numbers and number them sequentially, as:
((1 . 2)(2 . 3)(3 . 5)(4 . 7)...(10 . 29))
This loop assumes the existence of a function PRIME(X) that returns non-NIL if X is prime.

```
*SL*
     FOR I:=1
         WHEN PRIME(I)
         J:=1
         LIST J#I
         UNTIL J = 10
         ENDF

*IL*
     (FOR (BY I 1)(WHEN (PRIME I))(BY J 1)
          (LIST (CONS J I))(UNTIL (EQ J 10)))
```

Expansion:

```
   BEGIN I:=1,J:=1,NODE q1,q2,q3;
 t:IF TESTANDSET(q2) THEN I:=I+1;
   IF ¬PRIME(I) THEN GOTO t;
   IF TESTANDSET(q3) THEN J:=J+1;
   q1:=(J#I)#q1;
   IF J = 10 THEN GOTO r;
   GOTO t;
 r:RETURN DREVERSE(q1);
   END;
```

(4) Example of the use of LIST, LISTR, APPEND, AND APPENDR:

```
*SL*
     FOR A IN "((A B C) (D E) (F G H I) (J) (K L M)  NIL (N O))
          LIST A
          LISTR A
          APPEND A
          APPENDR A
          ENDF

*IL*
     (FOR (IN A (QUOTE ((A B C) (D E) (F G H I) (J)
                        (K L M)  NIL (N O))))
          (LIST A) (LISTR A) (APPEND A) (APPENDR A))
```

Expansion:

```
   BEGIN MODE A,MODE q1:="((A B C) (D E) (F G H I) (J)
                          (K L M)  NIL (N O)),
         MODE q2,MODE q3,MODE q4,MODE q5;
t:IF ¬q1 THEN GOTO r;
   A:=CAR(q1);
   q1:=CDR(q1);
   q2:=A#q2;
   q3:=A$q3;
   q4:=q4@@(A@NIL);
   q5:=A@q5;
   GOTO t;
r:RETURN LIST(DREVERSE(q2),q3,q4,q5);
   END;
```

The above loop will produce the following list:

```
  (((A B C) (D E) (F G H I) (J) (K L M)  NIL (N O))
   ((N O)  NIL (K L M) (J) (F G H I) (D E) (A B C))
   (A B C D E F G H I J K L M N O)
   (N O K L M J F G H I D E A B C))
```

## FOR Loop Syntax

```
*SL*

<for-loop>::=FOR #[<for-form>[,]] ENDF

<for-form>::=<for-gen>|<for-cond>|<for-control>|
             <for-bind>|<for-body>

<for-gen>::=<stepper>|<value-gen>|<norm-gen>|<reset-gen>

<stepper>::=[OLD|<num-type>]<name>::=[<expression>|*]
            [ [THRU|TO [≥|≤|>=|<=|=| =]] <expression>]
            [ [BY|STEP]<expression>]

<num-type>::=INTEGER|FLOAT|COMPLEX|NUMBER

<value-gen>::=<arith-val>|<norm-val>

<arith-val>::=[<num-type>][SUM|PRODUCT]<expression>

<norm-val>::=[AND|ALL|OR|ANY|FIRST|VALUE|UNION|INTER|COUNT|
              DAPPEND|DAPPENDR|APPEND|APPENDR|LIST|LISTR]
              <expression>

<norm-gen>::=<in-gen>|<on-gen>|<is-gen>

<in-gen>::=[OLD|<type-ref>]<name> IN [<expression|OLD <name>]

<on-gen>::=[OLD] <name> ON [<expression>|*]

<is-gen>::=[OLD|<type-ref>] <name> IS <expression>

<reset-gen>::=[OLD|<type-ref>]<name>:=[<expression>|*]
                                  RESET <expression>

<for-cond>::=<for-if>|<for-term>

<for-if>::=IF <expression> [<fif-form>|(#*,*<fif-form>)]

<fif-form>::=<for-gen>|<for-cond>|<for-body>

<for-term>::=[WHEN|UNLESS|WHILE|UNTIL] <expression>

<for-control>::=<init-gen>|<final-gen>

<init-gen>::=INITIALLY <statement>

<final-gen>::=FINALLY <statement>

<for-bind>::=BIND <block-bind-list>;

<for-body>::=DO <block-body> [END|ENDF]|
             <for-loop>|
             BEGIN<block-bind-list>;
                  [<attribute-list>]
```

```
                    <block-body>
                    {END|ENDF}


*IL*

<for-loop>::=(FOR @<for-form>)

<for-form>::=<for-gen>|<for-cond|<for-control>|
                <for-bind>|<for-body>

<for-gen>::=<stepper>|<value-gen>|<norm-gen>|<reset-gen>

<stepper>::=({BY|STEP} <num-for-var>{<expression>|*}
                <expression>[ ({GQ|LQ|LS|GR|EQ|NQ} <expression>) ])

<num-for-var>::=<name>|({OLD|<num-type>}<name>)

<num-type>::=INTEGER|FLOAT|COMPLEX|NUMBER

<value-gen>::=<arith-val>|<norm-val>

<arith-val>::=({SUM|PRODUCT} [<num-type>]<expression>)

<norm-val>::=({AND|ALL|OR|ANY|FIRST|VALUE|UNION|INTER|COUNT|
                  DAPPEND|DAPPENDR|APPEND|APPENDR|LIST|LISTR}
                  <expression>)

<norm-gen>::=<in-gen>|<on-gen>|<is-gen>

<in-gen>::=(IN <for-var> {<expression>|(OLD <name>)})

<on-gen>::=(ON {<name>|(OLD <name>)}{<expression>|*})

<is-gen>::=(IS <for-var><expression>)

<for-var>::=<name>|({OLD|<type-ref>}<name>)

<reset-gen>::=(RESET <for-var>{<expression>|*}<expression>)

<for-cond>::=<for-if>|<for-term>

<for-if>::=(IF <expression> #{<for-gen>|<for-cond>|<for-body>})

<for-term>::=({WHEN|UNLESS|WHILE|UNTIL} <expression>)

<for-control>::=<init-gen>|<final-gen>

<init-gen>::=(INITIALLY $<statement>)

<final-gen>::=(FINALLY $<statement>)

<for-bind>::=(BIND . <block-bind-list>)

<for-body>::=<binding-block>|<do-block>|<for-loop>
```

# PROCESSORS AND PROCESSES

This section describes processes, processors, and the primitives for manipulating them. Using processors, a "spaghetti" stack[1] implementation may be achieved. However, the cost of using processors instead of ordinary functions is high. Starting or resuming a process may easily exceed a millisecond: linkage to a function costs approximately 40 microseconds. A system design criterion has been that ordinary operation, function calls, and in-line code (as opposed to process switching) should run as rapidly in CRISP as the equivalent code would run in a LISP, FORTRAN, or PL/I system with a good compiler and run time support package. It is believed that this objective has been met in the current system design without severely penalizing programs that use the process switching capabilities. Multiprocessing is not a feature that is automatically invoked. A set of primitives are provided as a parts kit from which the programmer can tailor the system to meet his particular needs.

This section contains subsections on Processes, Processors, Processing Primitives, and Example Programs. The section on scoping and denotation rules (page 25) should be reread along with this section for better comprehension.

---

[1] See "A Model and Stack Implementation of Multiple Environments", Daniel G. Bobrow and Ben Wegbreit, Communications of the ACM, October 1973, Volume 16, Number 10, pages 591-603.

Processes

A process is a program that has been placed into execution. Each process is identified by a handle. A handle is a data object that contains the total state of a process. The state of a process has three parts: activity state, internal state, and external state. There are three possible activity states: active, suspended, and dead. The internal state of a process has two related parts: variable context and control context. The external state comprises three links: context link, abort link, and last activator link.

Several primitive operations are available that modify the state of a process. The operations START, RESUME, and KILL modify a process's activity and external states. The operations CONTEXT, ABORT, and ACTIVATOR can modify a process's external state. The internal state of a process is modified by execution.

The implementation associates with each process a stack that contains the internal state (control and variable contexts) of the process. Function calls and variable bindings are handled on the stacks in a normal, LISP-like manner. The handle associated with the process is a 5-tuple. The elements in a handle are (1) activity state, (2) context link, (3) abort link, (4) last activator link, and (5) the stack associated with the process.

### Activity state

When a process is started, a new handle and stack are created and initialized, and the new process is made active, that is, it is put into execution. Exactly one process in the system is active at any one moment. When a process is resumed, that is, put back into execution, the process that is currently active is suspended. A process is suspended whenever it has been started, has not been killed (or died of various computational ailments), and is not currently active. When a suspended process is resumed, it becomes active and continues operation from the point at which it had been suspended. A process becomes dead when it is killed (by the primitive KILL), when an error or unwrap is initiated in the process and there is no try in effect within the process, or when the internal state of the process becomes null. A dead process may not be resumed.

### Internal state

The internal state of a process is maintained on a stack. The stack contains return addresses, temporaries, variable bindings, and failset points established by the try primitive. If a process is suspended, its stack also contains a program counter save so that the process can be properly resumed. There is no stack associated with a dead process.

As an active process executes, its stack grows and shrinks. When a function is called, the argument values are computed and pushed onto the stack. Also, a return address (location

at which to re-enter the calling functional) is pushed onto the stack. The called function is then entered, and the argument values are paired with the parameters' proper names, thus creating bindings. As blocks are entered, the initial (preset) values of all local and global variables bound by the block are computed and pushed onto the stack. After all presets have been computed and pushed, the values are paired with the block variables' proper names, creating bindings. When a function or a block is exited, its associated bindings are popped off the stack, thus unbinding the variables. A return address is a pair: a function or processor definition and a point in that definition at which execution is to "resume". Thus, when a function is exited, the return address to its caller (which is popped) supplies the information necessary to restart the calling function. When a process is suspended, the program counter is saved in the same format as a return address.

Temporary values and results of computations may also be pushed onto the stack. CRISP is so designed that all such stack usage follows a last in first out (LIFO) discipline. Return addresses, temporaries, and failset points form a process's control state. The variable bindings on a process's stack form its variable context. Thus, the total internal state of a process is reflected by its stack. Failset points are described below in the section on processing primitives (page 179).

External state

The external state of a process is a set of three handles. One handle, the context link, is used to construct the process's total variable context. If a variable is referenced by a process that is not bound in its internal state, then the binding is looked for in the internal state of the process located by the context link. If not bound there, then the context link of that process is followed, etc.

The set of processes in the system form a tree. The processes are the nodes and the context links form the arcs. The root node is the pseudo process that has NIL as its handle. The NIL process is the set of top level variable bindings. The section on dynamic context (page 36) describes this in more detail.

The second part of a process's external state is its abort link. The abort link is a handle. The set of processes in the system also form a tree with themselves as nodes, the abort links as the arcs, and the NIL process as the root node. This tree need not be isomorphic to the tree formed by the context links. The usages of the abort link are described in the section on processing primitives (page 179).

The third component of a process's external state is its activator link. When a process is made active, the activator link is set to the handle of the process that

started or resumed this process. The activator links need not form a tree and may be circular.

## Processors

Syntactically, a processor definition (<processor-def>) resembles a function definition. See the section on definitions (page 97). Both have a name, an argument list, and an expression body. A processor is called much as is a function. (See the description of the START primitive below.) When called, the arguments are evaluated, and control is transferred to the processor definition. At this point, the process making the call is suspended, and a new process is created. On the new process's stack, the argument values are paired with the parameters' proper names. Then the body of the process is evaluated.

The normal way for a processor to exit is by resuming another process. (Unlike function calls, processors need not exit in the order in which they are created.) If a process completes execution of the processor's expression body, then an unwrap is induced in the process pointed at by the abort link. The value passed back by the unwrap is "(h COMPLETE), where h is the handle of the completed process. The process is marked as dead when this happens. The expression body of a processor is a novalue expression, thus, it is not required to produce a value.

## Processing Primitives

This section specifies the syntax of the primitives available for process control and describes their usage and meaning. In the syntax equations, a <proc-expression> is any expression whose value is a subspecified proc type (the name of a processor, for example). A <handle-expression>, or an <hexp> for short, is any expression whose value is of the type handle. The syntax of the primitives is:

```
*SL*
<processing-primitives>::=<failset-primitive>|
                          <process-copy-primitive>|
                          <external-state-primitive>|
                          <activity-changer>

<failset-primitive>::=<try-form>|<exit-form>

<try-form>::=<try>|<tryu>|<trys>|<trya>

<try>::=TRY([USER|SYS|ALL] #[,<form>])

<tryu>::=TRYU(#,*<form>)

<trys>::=TRYS(#,*<form>)

<trya>::=TRYA(#,*<form>)

<exit-form>::=<fail-form>|<error-form>

<fail-form>::=<fail>|<failkill>

<fail>::=FAIL([<expression>[,<hexp>]])

<failkill>::=FAILKILL(<expression>,<hexp>)

<error-form>::=<error>|<errorkill>

<error>::=ERROR([<expression>[,<hexp>]])

<errorkill>::=ERRORKILL(<expression>,<hexp>)

<process-copy-primitive>::=COPYPROC(<hexp>)

<external-state-primitive>::=<myself>|<activator>|
                             <context>|<abort>
```

```
<myself>::=MYSELF

<activator>::=ACTIVATOR(<hexp>[,<hexp>]) |
              MYACTIVATOR([<hexp>])

<context>::=CONTEXT(<hexp>[,<hexp>]) |
            MYCONTEXT([<hexp>])

<abort>::=ABORT(<hexp>[,<hexp>]) |
          MYABORT([<hexp>])

<activity-changer>::=<starter>|<kill>|<resumer>

<starter>::=<implicit-start>|<explicit-start>

<implicit-start>::=<proc-expression>($*,*<expression>)

<explicit-start>::=START(<implicit-start>,{<name>|NIL},
                         <hexp>,<hexp>,<hexp>)

<kill>::=KILLPROC(<hexp>)

<resumer>::=<resume>|<resumec>|<resumek>|<resumeck>

<resume>::=RESUME<resume-body>

<resumec>::=RESUMECOPY<resume-body>

<resumek>::=RESUMEKILL<resume-body>

<resumeck>::=RESUMECOPYKILL<resume-body>

<resume-body>::=(<hexp>[,<expression>[,<hexp>]])

<hexp>::=<handle-expression>


*IL*

<processing-primitive>::=<failset-primitive>|
                         <process-copy-primitive>|
                         <external-state-primitive>|
                         <activity-changver>

<failset-primitive>::=<try-form>|<exit-form>

<try-form>::=<try>|<tryu>|<trys>|<trya>

<try>::=(TRY {USER|SYS|ALL} #<form>)

<tryu>::=(TRYU {<name>|NIL>} #<form>)

<trys>::=(TRYS #<form>)

<trya>::=(TRYA #<form>)

<exit-form>::=<fail-form>|<error-form>
```

```
<fail-form>::=<fail>|<failkill>

<fail>::=(FAIL [<expression> [<hexp>]])

<failkill>::=(FAILKILL <expression> <hexp>)

<error-form>::=<error>|<errorkill>

<error>::=(ERROR [<expression> [<hexp>]])

<errorkill>::=(ERRORKILL <expression> <hexp>)

<process-copy-primitive>::=(COPYPROC <hexp>)

<external-state-primitive>::=<myself>|<activator>|
                              <comntext>|<abort>

<myself>::=(MYSELF)

<activator>::=(ACTIVATOR <hexp> [<hexp>]) |
              (MYACTIVATOR [<hexp>])

<context>::=(CONTEXT <hexp> [<hexp>]) |
            (MYCONTEXT [<hexp>])

<abort>::=(ABORT <hexp> [<hexp>]) |
          (MYABORT [<hexp>])

<activity-changer>::=<starter>|<kill>|<resumer>

<starter>::=<implicit-start>|<explicit-start>

<implicit-start>::=(<proc-expression> $<expression>)

<explicit-start>::=(START <implicit-start> {<name>|NIL}
                          <hexp> <hexp> <hexp>)

<kill>::=(KILLPROC <hexp>)

<resumer>::=<resume>|<resumec>|<resumek>|<resumeck>

<resume>::=(RESUME <resume-body>)

<resumec>::=(RESUMECOPY <resume-body>)

<resumek>::=(RESUMEKILL <resume-body>)

<resumeck>::=(RESUMECOPYKILL <resume-body>)

<resumebody>::=<hexp>[<expression>[<hexp>]]

<hexp>::=<handle-expression>
```

In the following discussion, the symbols h, h1, h2, and h3 are handle expressions, and m is the handle of the currently

active process. The symbol n is a name, and f1 ... fn are forms, either statements or expressions, as indicated by the text.

Failset forms

The failset primitives allow one context of evaluation to be left or aborted because of unusual circumstances and another context to be restored and activated. The <try> forms provide the "protection" points at which control is restored. The exit forms signal the unusual conditions.

<tryu>, <trys>, and <trya> are equivalent to a <try> with the keyword USER, SYS, or ALL, respectively. Thus,

```
TRYU(f1 ... fn)
```

is equivalent to

```
TRY(USER, f1 ... fn)
```

A try may be used as either a statement or an expression. When a try is used as a statement, the embedded forms are statements. A go form may branch out of but not into a try statement. When a try is used as an expression, the embedded forms are expressions. The value type of a try expression is the minimal type (as shown in Figure D, page 49) that contains the value types of all the embedded forms.

In operation, the try evaluates the first embedded form. If it evaluates normally, then the try is finished. If it evaluates abnormally (catches an unwrap caused by an exit form), then the second form is evaluated, etc. The

evaluation of the try is finished with the evaluation of the first embedded form that evaluates normally. If the try is used as an expression, then the value is the value of the first form that completes normally. If the last embedded form is evaluated and does not complete normally, then the unwrap continues through this try to the next "outer" try.

There are two kinds of <exit-form>s: <fail-form>s, which are for users, and <error-form>s, which are used by the system to report illegal situations at run time, such as DIVISION BY 0. Fails return control to a try with either USER or ALL specified, and errors return to a try with either SYS or ALL specified. The following pairs are equivalent:

FAIL() and FAIL(NIL)

ERROR() and ERROR(NIL)

When FAIL or ERROR is called without a second argument, control is returned to the last try (with the proper keyword) executed in the currently active process that has not completed execution. If there is no appropriate try in this process, then the unwrap continues in the process pointed at by the handle in the abort link. The abort links are followed in this manner until an appropriate try is found. As each process stack is searched for an appropriate try, variable bindings and control states are discarded (popped). If an entire process is unwrapped in this manner, it becomes dead. When control is passed back to a try in this manner, the internal state will have been restored to what it was before the trial evaluation was initiated. Variables' bindings but not their values are restored. That

is, if a value in a binding has been set by a failing form, then the value is not reset on abnormal evaluation. The NIL process always has a TRYA in effect to ultimately catch all unwraps that the user does not field himself.

The variable EXITVAL is set to the value of the first argument of an exit form, and the variable EXITKIND is set to either the identifier SYS or USER to reflect the type of the last exit. The settings occur after the unwrap in case either variable has been rebound.

When a two argument exit form is used, the process containing the exit form is suspended, the process located by the second argument is activated, and the unwrap occurs in that process. For example,

FAIL("XYZ,h)

The currently active process, m, is suspended and an unwrap is initiated in the process h with the message XYZ. If the process m is resumed, the value of the resume call will be the value of the fail. (This can occur only in the two argument case.)

The primitives FAILKILL and ERRORKILL work like fail and error with two arguments. The difference is that the process containing the exit form call is killed. These kill primitives should be used whenever it is undesirable to have control return to the process, m.

Process copy primitives

The primitive COPYPROC has one argument, a process handle. A new process with the same internal and external states is made and returned as the value if and only if the process is suspended or dead (in which case there is no internal state.) It is illegal to copy the currently active process, m.

External state primitives

The primitive MYSELF has a value that is the handle of the currently active process. The other three kinds of external state primitives have two forms, either one or two arguments. The following pairs are equivalent:

MYACTIVATOR() and ACTIVATOR(MYSELF)

MYACTIVATOR(h) and ACTIVATOR(MYSELF,h)

Similarly for the CONTEXT - MYCONTEXT pair and the ABORT - MYABORT pair. If ACTIVATOR is called with one argument, h, it returns the handle of the last process to activate h. If ACTIVATOR is called with two arguments, h1 and h2, then it returns the last activator of h1 and sets the last activator link of h1 to h2. Similar actions are taken for CONTEXT and ABORT. In addition, a check is made to ensure that the set of processes in the system will still make a well formed tree. If not, a run time error diagnostic and unwrap is issued.

Activity changers

There are three kinds of activty changing primitives: starters, KILL, and the various resumers. If p is a proc

expression and e1 ... en are expressions, then the following forms are equivalent.

p(e1 ... en) and

START(p(e1 ... en), NIL, MYSELF, MYSELF, MYSELF)

START creates a new process. The arguments e1 ... en are evaluated and passed to the processor that is the value of p. The argument values are paired with the parameter names on the new process's stack. If a name is included in the start form (instead of NIL), then the handle of the new process is placed there. Therefore, the name, if present, must be a variable with a type attribute of either general or handle. The three handle expressions embedded in the start form are, in order, the initial activator link, context link, and abort link for the newly created process. When the new process is started (activated), the currently active process is suspended. When the starting process is resumed, the value of the start form is the value passed by the resume primitive. (See below.)

The primitive KILLPROC deletes the internal state of the process that is its argument, thus rendering it dead. If KILLPROC(MYSELF) is evaluated, the currently active process is made dead and an error unwrap is induced in the process located by MYABORT().

The resumer forms suspend (or kill) the currently active process and activate another process. If e is an expression, then the following pairs are equivalent:

RESUME(h) and RESUME(h, NIL, MYSELF)

RESUME(h, e) and RESUME(h, e, MYSELF)

The operation of a resume follows these steps: (1) evaluate the three arguments, (2) set the last activator link of the first argument process to the value of the third argument, (3) suspend the currently active process and (4) activate the first argument process and feed the value of the second argument forward to the resume point. As an example:

```
PROCESSOR P()
  BEGIN;
    PRINT(RESUME(MYACTIVATOR(), "P.STARTED));
    ...
    END;

BEGIN HANDLE H;
  PRINT(START(P(), H, MYSELF, MYSELF, MYSELF));
  ...
  RESUME(H, "RESUMED);
  ...
  END;
```

When the processor P is started by the first line of the block, its external state links are all initialized to the process containing the block (call it M), and the handle of the new process is stuffed in the variable, H. The first thing done by the new process is to restart M and feed back the value P.STARTED, which becomes the value of the start call in M. Therefore, the first thing printed is P.STARTED. Eventually, the new process (with the handle in H) is resumed with the value RESUMED, which becomes the value of the resume call in the (new) process. The second message printed is therefore RESUMED. All message passing is done with data objects of type general even if this involves conversion.

The primitive RESUMECOPY is identical to RESUME except that the first argument process is copied by COPYPROC. The

equivalences are:

RESUMECOPY(h) and RESUME(COPYPROC(h))

RESUMECOPY(h,e) and RESUME(COPYPROC(h),e))

RESUMECOPY(h1,e,h2) and RESUME(COPYPROC(h1),e,h2)

RESUMECOPY is not normally used with co-routine implementations but becomes necessary when generalized backtracking schemes are being implemented. See the first example program in the next section.

The primitive RESUMEKILL is identical to RESUME except that the currently active process is killed rather than suspended. RESUMECOPYKILL, like RESUMECOPY, copies the first argument and, like RESUMEKILL, kills rather than suspends the currently active process.

If an attempt is made to resume an active or a dead process, an error occurs in the process containing the resume call. Neither RESUMECOPY nor RESUMECOPYKILL may attempt to copy the currently active process.

## Example Programs

The following program is an implementation of a full backtracking parser that is built from a combination of functions and processors. The processors are used to provide the state saves necessary to correctly perform backup. The equations are an internal representation of

BNF. However, the algorithm does not handle left recursion.

```
DEC PAT.LIST<PATTERN<NAME PAT.NAME, PAT PAT.PART>,
             LINK PAT.LIST>,
    GEN PAT.PART, ID PAT.NAME, ID TERMINAL,
    REP<MIN INT, MAX INT, PAT PAT.PART>,
    ELM<KIND ID, ITEM PAT.PART ARRAY(*)>,
    PAT.NAME START;

DEC NODE INPUT,
    EXITLIST<PRO HANDLE, RESTORE INPUT, LINK EXITLIST>;

PAT.LIST FUNCTION FINDPAT(PAT.NAME NAME)
  FOR PAT.LIST L:=PAT.LIST RESET L_LINK WHILE L
      IF NAME=L_PATTERN_NAME DO RETURN L END
      VALUE NIL
      ENDF;

BOOL FUNCTION PARSER(GLOBAL INPUT)
  TRY(USER, BEGIN GLOBAL EXITLIST;
                  RPARSER(START);
                  FOR EXITLIST F:=EXITLIST
                      RESET F_LINK WHILE F
                      UNLESS F_PRO=MYSELF
                      DO KILL(F_PRO) ENDF;
                  RETURN TRUE
                  END,
            NIL);

NOVALUE FUNCTION RPARSER(PAT.PART E)
  SELECTT E
    WHEN NIL THEN IF INPUT THEN EXIT()
    WHEN ID THEN BEGIN PAT.LIST L:=FINDPAT(E);
                       WHEN L THEN RPARSER(L_PAT)
                       WHEN INPUT&INPUT_FIRST=E
                            THEN INPUT:=INPUT_SECOND
                            ELSE EXIT()
                       END
    WHEN REP THEN REPEAT(E)
    WHEN ELM THEN SELECTQ E_KIND
                    WHEN ALT THEN ALTERNATIVE(E_ITEM)
                    WHEN CAT THEN CONCATENATE(E_ITEM)
                         ELSE ERROR("SYNTX)
         ELSE ERROR("SYNTX);

NOVALUE FUNCTION EXIT()
  IF EXITLIST
     THEN BEGIN EXITLIST F:=EXITLIST;
                EXITLIST:=EXITLIST_LINK;
                INPUT:=F_RESTORE;
                FAIL(NIL, F_PRO)
                END
     ELSE FAIL(NIL);

NOVALUE FUNCTION EXITSET()
  BEGIN EXITLIST X:=*;
```

```
        X_PRO:=MYSELF;
        X_RESTORE:=INPUT;
        X_LINK:=EXITLIST;
        EXITLIST:=X
        END;

PROCESSOR REPEAT(REP R)
  BEGIN L:=R_MIN, H:=R_MAX, PAT.PART E:=R_PAT,
        C:=MYACTIVATOR();
        ATTRIBUTE INT(L,H);
        FOR INT I:=1 THRU L
            DO RPARSER(E)
               ENDF;
        IF L>H THEN EXIT();
        FOR I:=L+1 TO H FINALLY RESUMEKILL(C)
            DO TRY(USER, (EXITSET(), RESUMECOPY(C)),
                         RPARSER(E))
               ENDF;
        END;

PROCESSOR ALTERNATIVE(PAT.PART ARRAY(*) A)
  BEGIN HANDE H:=MYACTIVATOR(), INT C:=ARLN(A);
        IF C=0 THEN EXIT();
        FOR INT I:=1 TO C
            DO TRY(USER, (EXITSET(),
                          RPARSER(A[I]),
                          RESUMECOPY(H)),
                         NIL)
               END
            FINALLY (RPARSER(A[C])), RESUMEKILL(H))
            ENDF;
        END;

NOVALUE FUNCTION CONCATENATE(PAT.PART ARRAY(*) C)
  FOR INT I:=1 THRU ARLN(C)
      DO RPARSER(C[I])
         ENDF;
```

The following program contains two co-routines, INPUT and OUTPUT. OUTPUT prints characters in four groups of three per line with each group separated by a space. INPUT returns the next character in the string S. If the character is a digit, then the next character is returned that many times instead of the digit. The character following the digit is then returned. Thus the string,

'A2B5E3426FGOZYW3210PQ89R'

is like

```
'ABBBEEEEEE4444666FGZYW222200PQ999999999R'
```

and is output as

```
ABB BEE EEE E44
446 66F GZY W22
220 0PQ 999 999
999 R
```

This contrived co-routine example has been borrowed from Knuth.[2]

```
NOVALUE FUNCTION OUTPUT(GLOBAL STRING S)
  BEGIN HANDLE H;
        START(INPUT(), H, MYSELF, MYSELF, MYSELF);
        FOR I:=1
            CHAR C IS RESUME(H) WHILE C,
            FINALLY IF REMAINDER(I,12) =1 THEN TERPRI(),
            DO PRINTCH(C);
               WHEN REMAINDER(I,12)=0 THEN TERPRI()
               WHEN REMAINDER(I,3)=0 THEN PRINTCH(SPACE)
               ENDF;
        END;

PROCESSOR INPUT()
  BEGIN I:=1,L:=ARLN(S);
        ATTRIBUTE INT(I,L);
        RESUME(MYACTIVATOR());
      L:IF I>=L THEN RESUMEKILL(MYACTIVATOR());
        IF DIGITP(S[I])
           THEN (FOR J:=0 THRU CHAR2INT(S[I])-CHAR2INT("$'0')
                    DO RESUME(MYACTIVATOR(),S[I+1]) ENDF,
                 I:=I+2)
           ELSE (RESUME(MYACTIVATOR(),S[I]),I:=I+1);
        GO L
        END;
```

---

[2] See "Fundamental Algorithms, the Art of Computer Programing", Knuth, Vol 1, page 191-194.

# THE CAP ASSEMBLER

The CAP assembler is used as the last pass of the CRISP compiler and is also available for those users who want to program in machine language. There are two versions of the CAP language: CAP SL, which closely resembles the standard IBM assembly language format, and CAP IL, which closely resembles LAP languages. Besides the normal capabilities of an assembler, CAP provides many pseudo instructions and operands that are used to maintain the stack, bind and unbind variables, and perform linkages. Also, CAP code is block structured in a manner that is similar to IL and SL.

CAP code sequences are always introduced into a program as either a CAP operand in SL or as a CAP expression in IL; either form may be used as a statement. In SL, a CAP operand is the word CAP followed by a value type, a sequence of CAP instructions (in SL format separated by semicolons), and the word END. In IL, a CAP expression is a form with the word CAP as its operator followed by a value type and a sequence of CAP instructions in IL format. The value of a CAP form is assumed to be in register F0 if floating and in register R5 if anything else. An example of a CAP sequence that adds the value of the two integer variables I and J in CAP SL format is

```
CAP INTEGER
    L    R5,I;
    A    R5,J;
    END
```

The same example in CAP IL format is

```
(CAP INTEGER
     (L R5 I)
     (A R5 J))
```

The system accepts input definitions for functions, processors, macros, and generators in one of four formats: SL, SL CAP, IL, and IL CAP. (See the section on tree structured files and the disk compiler on page 259.) An example of the same function written in each of these four formats is given next. The function performs the calculation I+2*J, producing an integer value from its two integer arguments, I and J. (The value of the last argument, in this case J, is passed in R5.)

FORMAT SL:

```
INT FUNCTION FOO(INT I, INT J)
             CAP INT
                 AR   R5,R5;
                 A    R5,I;
                 END;
```

FORMAT SL CAP:

```
INT FUNCTION FOO(INT I, INT J)
             AR   R5,R5;
             A    R5,I;
             END;
```

FORMAT IL:

```
(FUNCTION (FOO INT)((I INT)(J INT))
          (CAP INT
               (AR R5 R5)
               (A R5 I)))
```

FORMAT IL CAP:

```
(FUNCTION (FOO INT)((I INT)(J INT))
          (AR R5 R5)
          (A R5 I))
```

Thus the SL CAP and the IL CAP formats remove the necessity of redundantly entering the value type and the word CAP.

The scoping and denotation rules in CAP are the same as those in CRISP. The block structuring defined below

interacts with the stack allocation primitives to automatically assign the proper field values to instructions. The following subsections describe instruction formats, operand formats, pseudo instructions, and the macro facility. The section on register allocation and linkage (page 297) should be read along with this section for better comprehension.

## Instruction Formats

Appendix I, Summary of IBM 370 Instruction Formats (page 310), summarizes the available machine operations and the format of their operand fields. In SL, an instruction is written with its op code followed by the operands (separated by commas) and terminated with a semicolon. For example,

```
LA    R3,X;
A     R6,14(R3,6);
MVC   8(FOO,R2),X;
```

In IL, an instruction is a list with the op code appearing as the form operator and the operands appearing as subsequent members of the list. The above SL examples would appear in IL as

```
(LA R3 X)
(A R6 (14 R3 6))
(MVC (8 FOO R2) X)
```

If an identifier appears in place of a whole instruction, then it is assumed to be a label: "X;" in SL or "X" in IL (without parentheses). Integers in the range $-2^{20}$ through $2^{20}-1$ may also be used as labels. However, this can be dangerous because the compiler and assembler use negative

integers as generated labels.

Because of the semicolon delimiter in SL and the list structuring in IL, it is not necessary to start label definitions in column 1. You may have zero or more instructions on any line, and a single instruction may be split over many lines. Thus, the input format is truly free form and contains no column rules.

## Operand Formats

Appendix II, CAP Operand Formats (page 312), summarizes the operand formats of CAP instructions and should constantly be referred to when this section is being read.

There are six basic kinds of operand fields: registers; (bit) masks; numerics; full addresses that include a 12 bit displacement and two register fields; half addresses that include a 12 bit displacement and a base register field; and length addresses that include a 12 bit displacement, an operand length, and a base register field. The following paragraphs describe the various abbreviations used in Appendix II.

### Register and rid operand

When a register field is expected as an operand, only an expression or a register mnemonic can be used. (See the section on register allocation, page 297, for listing of

register mnemonics.) If an identifier, say R5, is used for a register field, the assembler looks for the value of the integer variable, R5$REGISTER. If no such variable exists, then the identifier is assumed to be a simple expression. The value determined for the register is converted to an integer and truncated to four bits.

Mask operand

A mask is used either as the branch condition for a BC or BCR command or as the byte selector for ICM, CLM, and STCM commands. The expression used for the mask is converted to an integer and truncated to four bits.

Numeric operand

A numeric operand is a constant of four, eight, or twelve bits, depending on its usage. The value of the expression used as a numeric operand is converted to an integer and truncated.

Address operands

A variety of operand formats are usable as full, half, or length addresses. If the selected form produces both index and base registers for a half or length operand, an error diagnostic will be issued. The length field in a length address is either a four bit or eight bit integer, depending upon the instruction. The length value is implicit for certain operand formats or is given explicitly by a ladr. Because length values should be one less than actual operand lengths, the assembler subtracts one from the specified

length (after truncation) before placing it in the binary image. (If the length is zero, then no subtraction is performed.)

Name

A name may be either local or global. The same scoping rules that are used in CRISP are used in CAP to determine the proper name of an identifier. There are exceptions. If the name is PUSHP., PUSHN., POPP., POPN., or RET., then the name is assumed to be a stackop. (These identifiers may not be used as local variable names in SL or IL.) Also, if an identifier is not located by the local and global name searches, then the name tailed with "CAPSYN" is looked for. If found, then the value of that synonym is used in place of the identifier. Also, a global name explicitly tailed into CAPSYN will be used as a global synonym. A local name is transformed into a (stack) displacement and a base register and thus may be used as a full, half, or length address. A global name is transformed into a displacement, base, and index register form and may therefore only be used as a full address. The implicit length of a name operand is four bytes.

Labelop

Since a label and a variable may have the same name, a syntax mechanism is necessary to avoid confusion. Note that a label may not be used as part of an expression. However, an indexed branch may be specified using a labelop. If an index is used, then a label operand may appear only as a

full address. Otherwise, a label operand may be used as a half or length address. The implicit length of a label operand is four bytes.

Sysop

A sysop is used to address an entry in sys1, sys2, num1, or num2 space. The identifier is the name of the entry. Sys entries are defined by the function MAKESYS or by the assembler pseudo-op SYS. The value of the expr, if present, is added to the displacement of the entry (from the proper base register) to form the complete displacement. The register operand, if present, is used as an index register. If the index field is not specified, then the sysop may also be used as a half or length address with an implicit length of 4 bytes.

Literal

Literal forms are used to introduce constant data in a program. There are no pseudo-ops in CAP to generate inline data. The reason for this is that the system must be able to disassemble a program sufficiently (when and if it is garbage collected) to knock down counts for name space references. Data in the program image would present random combinations against the environment.

There are several types of literals. Quote, hquote, and type allocate pointer constants that are stored in name space and may therefore be used only as full addresses. Int, float, half, byte, and multint literals allocate numeric

constants that are stored in num1 or num2 space and may therefore be used as any kind of address.

A quote literal is translated into a full address of a namea cell that contains a pointer at its value. If another quote is EQUAL to this one, it may share the same namea cell. Sharing is not guaranteed but may occur. Therefore, do not damage quote cells or change the fields of their values. The actual pointer put in the NAMEA cell is to a copy of the body of the quote operand. An hquote literal is like a quote except that the body is not copied, and EQ rather than EQUAL is used for the sharing criterion. Hquote is normally used when CAP code is generated by another program, not for constant fields in programs that are input from an external source such as disk.

The body of a type literal (a <type-ref>) is forced through the scoping rules and simplified to a simple type name (identifier), a global name of an ntuple, or to a function, processor, or variable or array subtype. If the result is an identifier or global name, then type responds like quote. Otherwise, the result is hashed and singularized so that EQ comparisons between <type-ref>s are possible. In any event, a type literal may be used only as a full address. Also, a command of the form

```
L R5,TYPE(A$B)
```

may be replaced by the faster command

```
LA R5,A$B
```

assuming that A$B is the name of a variable with ntuple type

A$B. Since quote, hquote, and type operands can be used only as full addresses, they do not have an implicit length.

Int and float literals, after proper conversion, allocate 32 bit constants in num1 or num2 space. They may be used as any kind of address, and their implicit length is four bytes.

Half and byte literals allocate 16 and 8 bit integers in num1 or num2 space. They may be used as any kind of address, and their implicit lengths are two and one, respecitvely.

A multint literal is used to allocate an integer in num1 or num2 space. The first operand, an integer, is the byte length of the operand. The following expressions are evaluated and converted to integers. The proper number of bytes are extracted, four per expr, and the extra high order bytes are truncated. The maximum length is 256 bytes.

Stackop

Stackops are used to reference unnamed quantities on the stacks. All have an implicit length of four bytes and all except a TOPP. or TOPN. operand, with an additional register specified, may be used as any kind of address. The latter may be used only as full addresses.

The CAP assembler maintains virtual stack maps at compile time and uses them to assign addresses relative to the stack

registers PDP and PDM. Thus; the stackops do not change the value of these registers, they only change the configuration of the virtual stack maps. Therefore, if the instruction

ST R5,PUSHP.

is executed in a loop, it references the same cell on each iteration (for a given function invocation). CRISP and CAP have been designed to allow this static stack mapping at compile time. With neither hardware stack operations nor hardware display registers on the 370, this scheme buys back execution time. Further, for a few code sequences, it is actually faster.

The address field PUSHP. advances the virtual stack pointer to the pointer stack by eight bytes and uses the updated address. The address field POPP. subtracts eight bytes from that virtual stack pointer and uses the updated address. The address field TOPP. uses the address of the top (last push not matched by a pop) of the pointer stack. The address field TOPP.(expr) adds the value of expr to the address of the virtual top of the pointer stack to derive the address. Recall that pointer stack entries are eight bytes long and that the value of expr is interpreted as a byte offset. Also, the real stack pointer is 400x bytes behind the return address on the stack for a given function invocation, so negative values of expr are perfectly appropriate. The address field

TOPP.(expr,register)

uses expr in the same manner. The register field is used as an index register. This latter form may be used only as a

full address.

The address fields PUSHN. and POPN. and the forms of TOPN. are equivalent in function to the same names ending in P. The difference is that they refer to the number stack instead of the pointer stack, and that the increments and decrements performed by PUSHN. and POPN. are four bytes.

The stackop SECOND provides a convenient method of addressing the high order four bytes of a local stack name on the pointer stack. Thus, SECOND(L) addresses the part of the stack entry for L that is used for global binding save information, not the value component that would normally be addressed by using the name, L. The stackop RET. acts like a local variable whose assigned stack location is the return address for the current function invocation. (This is stored on the pointer stack.)

Adr

An adr produces a 12 bit displacement and, optionally, a base and index register. If both registers are specified, the first is the index. If an index register is given, then the adr may be used only as a full address. An adr may not be used for a length address because of the syntax conflict. Therefore, it is meaningless to speak of the implicit length of an adr.

Ladr and implength

A ladr is used whenever an instruction is expecting a length

address field. In the form

expr(expr,register)

the first expr is the displacement and the second is the operand length. The register is the base. If the implicit length is not the one you desire, then any legitimate half address (h) can be used with

IMPLENGTH(expr,h).

The value of expr will override the implicit length of h.

Expr

An expr is a rudimentary expression that can be used to compute a value at assembly time. The expression is computed in full mixed mode, converted to the proper type, and, if necessary, truncated to the proper number of bits. The only non-obvious expr components are offsets and lengths. The argument for both is a full item name. That is, no containing group names may be omitted. Also, the specified item must be in the ntuple; links to other structures are not automatically followed, as they are by the compiler. Further, subscripts are not used.

The value of an offset form is the distance in bytes from the beginning of the ntuple to the first byte of the specified item. (This offset includes eight bytes for the ntuple header.) If the item would normally be subscripted, the value of the offset is the distance to the item with the lowest legal subscript values.

The value of a length form is the length in bytes of the

specified item including any interior or trailing slack bytes. Length of an ntuple name is the length of the structure, not including the header.

## Pseudo Instructions

Appendix III, CAP Pseudo Instructions (page 315), summarizes the formats of the pseudo instructions available in CAP and should be read along with this section. Each pseudo instruction is described below.

### CAP blocks

A CAP block is the assembly language equivalent of a CRISP <binding-block> (See the section on blocks, page 109.) The word BEGIN is followed by the sequence of instructions that compute the preset values for the variables, a BIND pseudo instruction that binds the block variables, the block body (another sequence of instructions), and the word END. The preset computation leaves the initial variable values on the stacks in the order in which the variables appear in the bind pseudo instruction. For example, the CRISP block prologue:

```
BEGIN I:=17,J:=-12,FLOAT F,NODE2 N:="(A B);
  ATTRIBUTE INT(I,J) GLOBAL(F,N);
```

would appear in CAP as:

```
BEGIN LA    R5,17;
      ST    R5,PUSHN.
      MVC   PUSHN.,INT(-12);
      ST    ZERO,PUSHN.;
      L     R5,"(A B);
      ST    R5,PUSHP.
      BIND INT I,INT J,GLOBAL FLOAT F,GLOBAL NODE2 N;
```

The format of the body of the bind instruction is the same as the format of a CRISP block bind list, except that: (1) own variables are not allowed, (2) <local-syn-dec>s are not allowed, (3) no preset forms are permitted, and (4) no attribute forms may be used.

If any of the variables in the bind list are global names (or the scope attribute is GLOBAL) then the bind pseudo instruction outputs the code necessary to globally bind those variables. If none of the variables are global, then the bind instruction does not output any code; it just alters the virtual stack maps. Also, in this case it is permissible to branch out of the block using any of the branch instructions. When global variables are bound, it is necessary to use the GO pseudo instruction or to "fall through" the block to make sure that globals are properly unbound. That is, the unbinding sequence for blocks that bind global variables is automatically generated at the end of the block. If the block binds no variables, then it is permissible to branch into the block at any label. Note, you are not considered to be in the block until the bind has been executed and the block body is entered. The preset sequence acts like part of an outer block. This means the bound variables are visible only in the block body.

Branching pseudo instructions

The branching pseudo instructions are the standard IBM extended op code set defined in the "little yellow card"[1], plus branch on true (BONT), branch on false (BONF), and GO. The extended op codes all expand to BC or BCR instructions with masks that select the proper values of the condition code. For instance, BZ means branch on a zero result. The codes ending with an R produce BCR instructions, and the others produce BCs. The label required by BONT, BONF, GO, or the extended ops that produce a BC may simply be a label name. It is not necessary to write out a labelop operand unless you wish to use an index as well as a base register. If the operand is an identifier or an integer, it is assumed to be a label name. The instruction

```
BONT r,L
```

is equivalent to

```
BXH r,ZERO,LABEL(L)
```

and

```
BONF r,L
```

is equivalent to

```
BXLE r,ZERO,LABEL(L)
```

The pseudo instruction, GO, is an unconditional branch to the specified label. Its operand must be a label (an integer or identifier) or the identifier "RET.". RET. used as a GO operand means return from the program that is currently in operation. GO may branch out of blocks that

---

[1] IBM System/370 Reference Summary, GX20-1850-n.

bind global variables. If it does, then code is generated to unbind the variables automatically. Therefore, the amount of code generated by a GO pseudo instruction is variable. If no global bindings are crossed, then "GO RET." generates the two byte instruction "BR FNRT", and "GO L" generates the four byte instruction "B L". Four additional bytes are generated for each block that is crossed that contains global bindings.

Stack pseudo instructions

The stack pseudo instructions cause the assembler to increment or decrement a virtual stack pointer by the number of bytes specified by the value of their operand, expr. The value of expr is rounded upwards to a multiple of eight for the instructions PUSHP. and POPP. (which refer to the pointer stack), and to a multiple of four bytes for PUSHN. and POPN. (which refer to the number stack). These operations do not produce any code and therefore do not initialize the stack entries. They are normally used to reserve an area of stack to use as a scratch area for local computation or to inform the assembler that certain stack temporaries are no longer in use.

Callers

The caller pseudo instructions provide an easy method of coding linkages to other programs. CALL is used to link to an ordinary function, FUNCALL is used to invoke a func valued expression, START is used to start a processor, STARTPROC is used to start a proc valued expression, and

SYSCALL is used to invoke a function in sys1, sys2, num1, or num2 space. Because arguments may be transmitted on the stacks, the assembler automatically returns the virtual stack configurations to the ones that existed before the caller was executed. Thus, the stack entries used as arguments are automatically popped. Some examples of the use of callers are:

```
APPEND(A,B) CALL(APPEND
                 L  R5,A;
                 ST R5,PUSHP.;
                 L  R5,B);

READ()      CALL(READ);

F(A,B) where DEC FUNC F(FLOAT,INT)
             FUNCALL(L  F0,A;
                     ST F0,PUSHN.;
                     L  R5,B;
                     L  R7,F);

LENGTH(L) without error checking
             SYSCALL(LENGTH L R5,L);
```

CALL, FUNCALL, START, and STARTPROC output the usual linkage sequence which includes

```
BALR LINK,FNLK;
```

CALL and START also output the load address necessary to locate the name cell of the called program. SYSCALL only outputs the BAL necessary to reach the proper SYS space entry. Normally, code reached by a SYSCALL expects all of its arguments in registers; the stacks are not actually incremented by the function linkage sequence, as they are in the other cases.

Synonyms

The CAP local synonym capability is very limited. The expr body must contain no forward references and cannot include

any labels or other non-synonym names. The scope of the synonym is lexically forward for the duration of the block (or program definition) in which it is defined. (Recall that preset calculations are part of an outer block.)

SYS pseudo instruction

The SYS instruction requests the assembler to locate the binary of the program in which it appears in the next set of available locations in the specified system space. It should appear somewhere near the beginning of the definition (how near has not yet been determined). The SYS command inhibits the assembler from automatically outputting the instructions that place the last argument on a stack, the instructions that automatically split the stack for processors, and the return via "BR RNRT" at the end of the definition. Code placed in num1 or num2 space must not modify itself because parts of it may double as numeric literals.

TRY pseudo instruction

The TRY pseudo instruction is used to establish protection points against stack unwraps. The first operand specifies the kind of unwraps to stop. The remaining operands are an ordered set of instruction sequences. If an unwrap occurs in the first sequence, then the second sequence is automatically attempted, etc. If unwrap occurs in the last sequence, it continues through the current try to the next outer try that is in effect. In all but the last sequence, there must be at lease one ENDTRY pseudo instruction.

ENDTRY is the only legitimate method of branching out of a non-terminal try sequence without exiting or failing. The operand for ENDTRY is a label. The try entry on the stack is zapped, and a GO to the label is performed that unbinds any necessary globals.

Space test

A space test determines what kind of space a pointer addresses. The specified register contains the pointer. The pointer is clobbered by the operation of the space test. The following phrases specify space kind name and transfer point pairs. NIL means "fall through" the space test. A label may be specified, in which case the proper BC to that label is generated; or "R register" may be used, in which case the appropriate BCR to that register will be generated. Do not use R0 as a branch address register; it may lead to a surprising memory protect violation. If the word RET. is used, it will be treated identically to "R FNRT". A space test may not branch out of blocks or programs that bind global variables.

Assume that you wish to test the pointer in R2; if it is an array, then you want to branch to label L. If it is an integer, you want to fall through the test. If anything else, then you want to return from the program. To accomplish this, code the space test,

```
SPACE R2,ARRAY L,INT NIL,RET.;
```

As can be seen from this example, the last phrase can specify a "nothing works" branch point. After execution of

a space test, the specified register contains the byte address of the QCM (Quantized Core Map) entry corresponding to the addressed page (see section on core maps, page 267).

## CAP Macros

The macro facility for CAP programs has not yet been designed. The most probable implementation is through the transform facility with the addition of some conditional pseudo ops.

# SYSTEM DESCRIPTION

The second half of this document describes the CRISP system. It is necessarily incomplete because the system has not yet been written. The sections included are: The I/O Facility, Primitives, Tree Structured Files and the Disk Compiler, Memory Management Facility, and Register Allocation and Linkage. The sections that are not included are: How to Login and Get Started, Interactive Supervisor, and The Program Check Handler. For the typical user, the most important sections are The I/O Facility, Primitives, and Tree Structured Files and the Disk Compiler.

# THE I/O FACILITY

The I/O facility available in CRISP provides a flexible capability for communications through the CMS[1] system. There are primitives for both binary and symbolic data transfer. The devices that may be used are disk, tape, card reader, card punch, printer, and terminal. The card devices and the printer are of course, the spools provided by CP (the VM Control Program, which is responsible for management of the user's "machine"). In the future, a facility to use VM's channel-to-channel adapter simulation (CTCA) will be implemented. Using a CTCA, CRISP programs may initiate connections via the ARPA network to other computers. Also, virtual machines may communicate with each other through a CTCA connection. Another capability anticipated for the future is a formatted output package similar to that provided by FORTRAN or PL/I systems. The original system will have automatic formatting for structuring output called "pretty" printing, but no provisions for user specified formats.

Besides the I/O facility, this section describes the other kinds of CP and CMS services provided to CRISP programs. These include access to the timer, program saves, and

---

[1] CMS is the VM Conversational Monitor System. It is the operating system that usually runs in a user's virtual machine and provides a file system, utilities, etc. See the IBM Virtual Machine Facility/370: Command Language Guide for General Users (GC20-1804-n) for a complete description.

construction of command tables.

## File Descriptor List

In order that a CRISP program be able to properly access a file, two types of information must be specified: (1) the information necessary to locate and correctly identify the file and (2) the features that describe the intended use and the format of the file. Both types of information are grouped together in a file descriptor list (FDL). In an FDL, the file identification information appears first, followed by the usage information. The file identification information is highly order dependent, while the usage information may be permuted without effect.

### File identification

The format of the file identification varies with the type of device. The following description explains, by using examples of FDLs that contain only the identification, the legal formats. A terminal file identification is (c-name TERMINAL) where c-name is an identifier that will be used as the internal name of the file in CRISP. Several of the I/O primitives require only the internal name as an argument. Similarly, the formats of card and printer file identifications are (c-name READER), (c-name PUNCH), and (c-name PRINTER). The format of the identification of a file on tape is ([c-name] TAPn). If the c-name is omitted, then TAPn will be used as the internal name. TAPn, where n

is a single digit, identifies the tape unit address as 18n hex in the normal CMS tradition. Since you are not allowed to have two files simultaneously opened on the same tape unit, there is no loss in generality by using TAPn as the internal name.

The identification format for a file on disk is slightly more complicated. The possibilities are

```
(c-name DISK f-name {f-type|*} {f-mode|*})
(f-name {ftype|*} {fmode|*})
```

In the second format, the internal name is not given explicitly and is assumed to be the same as the f-name. F-name, f-type, and f-mode are the file name, type, and mode as known to CMS. That is, f-name and f-type are identifiers, and f-mode is either a single letter (A to G, Y, Z, or S) or an alphanumeric combination such as A1, C3, etc. A star (*) may only be used as the f-type or f-mode when an existing file is opened for input or for output extention. The * implies that the system should search for a file that meets the rest of the specification. The identification format (f-name f-type *) means the first file encountered with the specified name and type, by going through the modes in the same order as would CMS. See the IBM VM/370 Command Language Guide for General Users for explanation of file naming conventions and search ordering.

The identification format (f-name * f-mode) means the first file encountered with the specified name and mode, by going through the types named by the variable FILETYPES. The initial value of FILETYPES is (CRISP INDEX DATA). You may

change the type search order by re-ordering, adding, or deleting elements from this list. The identification format (f-name * *) means the first file with this name, encounterd by searching FILETYPES and the CMS modes. For example, the total search order for (XYZ * *), assuming that the CMS search order is (A B) is:

```
(XYZ CRISP A)
(XYZ INDEX A)
(XYZ DATA  A)
(XYZ CRISP B)
(XYZ INDEX B)
(XYZ DATA  B)
```

As can be seen in the above, the type is varied faster than the mode.

File usage information

The following describes the possible members of the usage specification portion of an FDL. The members follow the file identification and may appear in any order. Not all usage specifications can be used with every kind of file. The restrictions, such as input vs. output, binary vs. symbolic, disk vs. tape, etc., and the use of the FDL, open vs. close, are also noted. If a set of usage parameters are described together and one of them is the default, the default is underscored.

SYMBOLIC, BINARY

Binary files are used to hold byte strings of data that are directly transferred to and from the user's buffer with no interpretation by CRISP. Symbolic files are used for both formatted and unformatted data that is represented in either EBCDIC or ASCII characters. All program text and most data

files are symbolic. Note, this distinction is apparent only to CRISP; CMS has no such categorization of files.

R, W, WE

The respective meanings are input, output, and output extension to an existing file. Output extension, WE, makes sense only to a disk or tape file. If specified for a tape, the unit is spaced ahead to the next file mark, and the mark is erased. The default value is W for PUNCH and PRINTER files and R for all other files.

DL|DM|DH

The respective meanings are for density: low, medium, and high. This makes sense only for an output tape file because, on reading, a tape unit automatically determines the density.

V, F, (SIZE i)

These mean, respectively, variable and fixed length records, and SIZE specifies that the record size is i for F files and a maximum of i for V files. All size, length, and margin information for lines is given in bytes. The default values depend upon whether the file is R, W, or WE and whether the file is binary or symbolic. The table summarizes the default actions:

| | R | R-B | WE | WE-B | W | W-B |
|---|---|---|---|---|---|---|
| DISK | 0 | 0 | 0 | 0 | V80 | V |
| TAPE | V80 | V | F80 | V | F80 | V |
| CARD | F80 | V | - | - | F80 | V |
| PRINTER | - | - | - | - | V130 | V |
| TERMINAL | V130 | V | - | - | V72 | V |

B means that the file is binary, and V without a length means simply input or output the number of bytes specified for a binary file. For all symbolic I/O, an upper bound length must be specified because CRISP associates a buffer with each symbolic file. O means that whether the file is V or F is determined by opening the file and looking at the CMS control block. If F, then the size information is deduced by looking at the control block. If V and binary, the length information is not needed. However, if the file is V and symbolic, then the default length will be the maximum of the longest record written to date and 80 bytes. If you do not like the defaults, then override them with an explicit usage parameter.

ERASE, REWIND

These option words are meaningful only when a file is being opened. When opening an output disk file (W), ERASE specifies that if a file already exists with the same name, mode, and type, it should be erased. If ERASE had not been included in the FDL and the file had already existed, then OPEN would signal an error. REWIND is useful only in open calls for tape files; it ensures that the tape is positioned at its load point.

PURGE, CONT, REW, REU

These option words are meaningful only when a file is being closed. PURGE aborts the spool for card and printer files, erases a disk file, and is a nop for tape and terminal files. For card and printer files, CONT closes the file as far as CRISP is concerned but does not close the spool. CONT closes a tape file without writing a tape mark. For disk and terminal files, CONT is a nop. REW and REU are meaningful only for tape files. Their respective meanings are rewind and rewind and unload.

OK, TD, TN, TTY, TI, PDP10

These option words are meaningful only for symbolic output files. They specify the available character set assuming that a human will try to read the file. That is, they specify which characters must be printed using the %% mechanism to ensure visibility. The meanings are:

| | |
|---|---|
| OK | do not use %% for any char |
| TD | using TD print train |
| TN | using TN print train |
| TTY | using a Model 33 TTY |
| TI | using a TI 700 or Model 38 TTY |
| PDP10 | using the SBI PDP-10 printer |

The default setting for each device type is:

| | |
|---|---|
| DISK | TI |
| TAPE | OK |
| PUNCH | OK |
| PRINTER | TD |
| TERMINAL | TI |

EBCDIC, ASCII

Specifies whether characters are to be in EBCDIC or ASCII format in the file. These specifications are meaningful only for symbolic files. Note, a terminal is handled as an

EBCDIC device by VM. This option is intended primarily for symbolic tape exchange between the 370 and the PDP 11 or the Raytheon 704. Therefore, if these two character sets turn out to be sufficiently different from each other, two brands of the ASCII option will be available.

CCA

This option word specifies that a symbolic output file is to have carriage control information added to the data. This is useful primarily for printer files and disk files of file type listing that will ultimately be output on the printer.

PAGEN, (HEAD s), (TPMG t), (BTMG b), (PAGEL l)

If any of these options is used, then the symbolic output will appear in page format. PAGEN specifies page numbering on the top line of each page. The phrase "PAGE i", where i is the page number, will be right justified. (HEAD s) specifies that the value, s, should be left justified in the top line of each page; s must be an identifier or a string. (TPMG t) specifies that t lines should be left before the beginning of text at the top of each page. t includes the heading and numbering line. (BTMG b) specifies that b blank lines should be left at the bottom of each page. (PAGEL l) specifies that the total number of lines on a page is l. When page formatting is selected by the user and any or all of t, b, or l are not specified, then their default values are 3, 3, and 66, respectively. The following conditions must be satisfied:

$t+b<l$
$t \geq 1$ if PAGEN or HEAD

If CCA has been specified, then most of the formatting will occur with carriage control characters. Otherwise, blank lines will be used. CCA is useful without page format to force an occasional skip to top of form.

(LFMG l), (RTMG r), LINEN

These option words are meaningful only for symbolic files. (LFMG l) means ignore the first l columns of input or automatically print l blanks in an output line. (RTMG r) means ignore the last r columns of an input line or put blanks in the last r columns of an output line. LINEN is equivalent to (RTMG 8). CRISP does not use or generate line numbers. LINEN is merely a convenient method of skipping over line numbers on input or to leave room for them on output.

PRETTY, UGLY

These option words apply only to symbolic output files. PRETTY means do an automatic formatting of printed structures. UGLY means just print the structure as a token string with the necessary separating blanks supplied.

ASYM, SYM, NSYM, EYE

These options are meaningful only for symbolic output files. SYM means "symmetric" printing is required. That is, the structure must be output in a form that allows it to be reread by a CRISP program. If anything is printed in a SYM file that is deemed to be not rereadable, then an error is signalled. Examples of such unprintable structures are

handles, pointers into heaps, and circular structures. When necessary, identifiers will be printed using the $' mechanism, and strings will be printed with primes. ASYM, the default, is like SYM except that no error is signalled when a non-rereadable structure is printed. NSYM is like ASYM except that the $' mechanism is never used. However, strings are still printed with primes. With EYE, neither is the $' mechanism used nor are primes used to print strings.

CAPS, NCAPS

These options are meaningful only for symbolic input files. CAPS specifies that input characters should be raised to upper case, if necessary. This is the default. NCAPS specifies that no capitalization should be performed.

## File Handling Primitives

This section describes the set of primitives that may be used to change the status of a file. Following sections describe the primitives that can actually result in data transfer.

OPEN(FDL)

OPEN defines and makes a file usable to CRISP programs. The FDL is error checked, and if found faulty, an error condition is signalled. Examples of errors besides the FDL format are:

- specified file does not exist
- file is already open

- tape unit, printer, reader, or punch already in use
- disk full and other CMS detected errors

Once a file has been opened, it is always referred to by its internal name. The system is initialized with the following OPEN calls:

```
OPEN("(ITERM TERMINAL R));
OPEN("(OTERM TERMINAL W));
```

The value of OPEN is a copy of the file identification portion of the FDL. If * had been used for either the type or the mode, then the value of OPEN would include the actual values used. Only one file each is permitted for the card reader, printer, card punch, or any single tape drive at any given time. Also, an open disk file may not be opened again without first closing it.

CLOSE(c-name | FDL)

CLOSE removes the specified file from the set of active files usable by CRISP programs. If the internal name is specified (by use of a c-name argument) then the default action for each device type is:

| | |
|---|---|
| DISK | make an FSCLOSE call |
| CARD | close the spool, NOCONT |
| PRINTER | close the spool, NOCONT |
| TAPE | write a tape mark |
| TERMINAL | no operation |

If an FDL is specified, the option words PURGE, CONT, REW, and REU are obeyed as specified in the previous section. In any event, if the specified file is not open (to CRISP), then CLOSE is a nop and the value is NIL. Otherwise, the value is TRUE.

TURNAROUND(c-name)

The argument is the internal name of a file currently opened for output. It is closed and then re-opened as an input file. If the file is on tape, then a tape mark is written and the tape is backspaced one file. Note that the tape is not rewound.

EXTEND(c-name)

The argument is the internal name of a file currently opened for input. It is closed and then re-opened as a WE (write extension) file.

CHANGE(c-name,$*,*attributes)

The specified usage attributes of the specified symbolic output file are changed. The possible arguments are:

```
(HEAD s)
PRETTY, UGLY
ASYM, SYM, NSYM, EYE
```

The heading may be changed only if page formatting was selected by the open call.

POSITION({c-name|TAPn},command)

A tape unit is positioned. Whether or not a file is opened on this unit, TAPn may be used to identify the drive. The possible commands and their meanings are:

```
REW  rewind
REU  rewind and unload
ERG  erase a gap
BSR  backspace 1 record
BSF  backspace 1 file
FSR  forward space 1 record
FSF  forward space 1 file
WTM  write a tape mark
```

Note, everything mentioned deals with physical records,

files, and drives. If you start positioning around symbolic files you either need to know exactly the format of the records or be the recipient of divine guidance.

SEEK(c-name,i)

The pointer to the specified disk file is set to the ith record. If i is zero, then the pointer is set at the file's end. For restrictions on the use of this command, see the CMS documentation.

ERASE(f-name,f-type,f-mode)

The specified disk file is erased. If the file is opened, then it will immediately be closed.

RENAME(f-name1,f-type1,f-mode1,f-name2,f-type2,f-mode2)

The file, (f-name1 f-type1 f-mode1) is renamed as (f-name2 f-type2 f-mode2) by CMS. The file must not be open in CRISP.

## Binary I/O Primitives

This section describes the primitives that are used to transfer data to and from binary files.

WRITE(c-name,structure)

The content of the structure (not including its header) is output to the specified file. The structure must be an array or ntuple that contains no elements that are

represented by pointers. The length of the structure is determined from its header. If the data structure is longer than can be contained in a single record, then multiple records are output.

WRITEX(c-name,loc,offset,length)

loc is the integer byte location of the buffer, offset is a byte offset away from loc, and length is the number of bytes that are to be output. WRITEX promises to do nothing that will cause a garbage collect; therefore, loc will remain valid. WRITEX is pure hacking and should be used only by the knowledgeable hacker and only in emergencies.

BREAD(c-name,structure)

The content of the structure (not its header) is filled from the specified binary input file. If necessary, multiple records are input. Unused bytes in the last input record are discarded. The structure must contain no fields that are represented by pointers. If the end of file condition is encountered, the value is NIL; otherwise the value is TRUE.

BREADX(c-name,loc,offset,length)

BREADX is the input equivalent of WRITEX. It is even more of a hacker's delight. No error checking is done other than to ensure that the file is opened and that the input occurs without channel or device error. If you exceed memory bounds or clobber the system, you are on your own; the program check handler will probabley be turned into a basket

case. The value of BREADX is the same as that of BREAD. If an end of file is encountered by either function in the middle of data input (can occur only in multi-record reads), an error will be signalled.

It should be noted that binary I/O primitives do not move data to and from system buffers. The transfer is done "in place". To do otherwise would create a severe page thrashing and execution time penalty for large transfers.

## Symbolic I/O Primitives

This section describes the primitives used to transfer data to and from symbolic files. Unlike the primitives that transfer binary data, the file is not specified as an argument. At any moment, one input and one output file are selected for symbolic data transfer. The primitives always work with a file that is currently selected. The value of the variable READFILE is the internal name (identifier) of the currently selected symbolic input file, and the value of the variable PRINTFILE is used in a like manner for the selected symbolic output file. Both variables have the last name CRISP.

File selection may be accomplished easily by assigning a new value to one of the variables. Also, file selection may be protected against error or fail unwraps by binding the variables. Another method, borrowed from LISP, is also

available for selecting files. The function RDS has one argument, the internal name of a symbolic input file. RDS sets READFILE to the argument value and returns the old value of READFILE. Thus, RDS(r) is equivalent to:

```
BEGIN ID I:=READFILE;
  READFILE:=r;
  RETURN I;
  END;
```

The function PRS is available to select an output file. It works in a manner identical to RDS but with the variable PRINTFILE. The following paragraphs describe the symbolic I/O primitives.

READCH()

The next charcter in the input line is input and converted to a character identifier. If the current input line is exhausted, the the next line is read. If the end of file condition is detected, the value of READCH is NIL. Characters represented by %% and two hex digits are converted to a single character. A blank is inserted as the last character of each input line. If that implicit blank is the character input, then the value of the boolean variable ENDOFLINE$CRISP is set TRUE.

READCHX()

Just like READCH except that no %% conversion is performed.

BACKCH()

BACKCH causes the last character input by READCH or READCHX to be re-input when one of the character readers is next called. Provisions are made only for one character backup.

Therefore, repeated BACKCH calls are equivalent to a single call.

READTOK()

READTOK skips over leading spaces and comments in the input, parses the next token, converts it to internal form, and returns it as its value. If the implicit blank, supplied at line end by READCH, occurs in the middle of an identifier input using the $' mechanism or a string surrounded by primes, then it is discarded and not used in building the token. If the input token is an identifier input with the $' mechanism, then the boolean variable USEDOLLAR in section CRISP is set TRUE. READTOK includes + and - as part of a succeeding number. Thus, -15 is input as one token, not two. If an end of file is encountered, the value is the identifier EOF and the boolean variable EOF$CRISP is set TRUE.

READTOKU()

READTOKU is identical to READTOK except that + and - are considered as separate tokens from succeeding numbers. Thus, -15 is input as two tokens.

BACKTOK()

BACKTOK functions for tokens in a manner similar to the way BACKCH works for characters. That is, after a call on BACKTOK, the next call on READTOK or READTOKU will return the token a second time. Provisions are made for backing up only one token. Therefore, multiple calls on BACKTOK are

equivalent to a single call. If a signed token has been input by READTOK and backed up, then a call on READTOKU will retrieve the signed number. Token backup does not imply a corresponding character backup. The next character pointer and character backup flag are not affected in any manner by BACKTOK. Therefore, in general, an input stream should be considered to be either a token sequence or a character sequence and caution should be used when mixing modes.

READINT()

READINT inputs the next token. If it is an integer, that is the value. If it is a float, it is converted to an integer and returned. Otherwise, an error is signalled. If the end of file condition is encountered, the value is zero, and the boolean variable EOF is set TRUE. Using READINT instead of READTOK, when possible, is more efficient because the integer does not need to be converted to a general datum.

READFLT()

Like READINT except that it inputs a floating value. It is willing to convert from an integer input.

READ()

READ inputs the next external datum in the input stream. READ can input lists, nodes, strings, arrays, complex numbers, and ntuples as well as one token data such as integers and floating point numbers. When the tokens $'(', $')', $'[', and $']' are encountered, they are treated as identifiers and not as structural delimiters. If the end of

file condition appears in the middle of an incomplete structure, an error is signalled. If the end of file condition is met after skipping leading spaces and comments and before any structural data are encountered, the value is the identifier EOF, and the boolean variable EOF is set TRUE.

CRUNCH(l)

The argument is a list of tokens. The tokens are treated as an input stream to READ. If a token would have been input using the $' mechanism because it would otherwise have been a structural delimiter, then it should be in a sublist. Signs may be included. Also, numbers may be negative. The value of

```
CRUNCH("($'(' GEN ARRAY $'(' 4 $')' - 4
          -17 ($')') ABC))
```

is

```
(GEN ARRAY(4) -4 -17 $')' ABC)
```

If the input list is not exactly exhausted, an error is signalled. Notice that type-refs are given in token form just as everything else is.

READSL()

An SL top level expression is input from the file. The value is the IL equivalent. The input must be delimited by a semicolon.

TABINTO(i)

The input character pointer is set to the ith character position of the current line. If i is less than the left

margin, the pointer is set to the first character following the margin. If i would cause the pointer to be set into or beyond the right margin, then it is placed in a position such that the next character input will come from the next input line. In any event, the variables EOF, ENDOFLINE, and USEDOLLAR are set to NIL, and any token or character backup is cleared.

TABINBY(i)

Identical to TABINTO except that i represents a delta instead of a character position. Of course, negative values of i are meaningful.

ENDLINEIN()

Sets the next character input pointer so that the next call for a character input will cause another line to be read. USEDOLLAR is set to NIL, and any token and character backup is cleared. ENDOFLINE is set TRUE.

NEXTLINEIN()

Inputs the next line. Returns NIL and sets EOF TRUE if the end of file condition is encountered. Otherwise, TRUE is returned. USEDOLLAR and ENDOFLINE are set to NIL, and any character and token backup is cleared.

Note: An attempt by any input operation to pass through the end of file a second time will cause an error to be signalled.

PRINTCH(c)

The argument, a character identifier, is entered into the output buffer. If appropriate, it is printed as a %% and two hex digits. If this character fills the line, then the line is output to the file. The value is the argument.

PRINTCHX(c)

Like PRINTCH except that the %% mechanism is not used.

PRINT(x)

The argument is printed. If the currently selected output file has the usage attribute PRETTY (as opposed to UGLY), the output is formatted. The last line of the output is padded with blanks, if not completed, and written out. The value is the argument.

PRIN(x)

Like PRINT except that the last line is not padded nor is it forced out. Thus, several PRINs may be used to enter data on the same output line. However, blanks are not automatically inserted.

BLANK()

Does a PRINTCH of a space.

BLANKS(i)

Outputs i spaces using PRINTCH.

BLANKTO(i)

Moves the next character output pointer to column i. Blanks are printed until column i is reached, even if this means moving to the next output line.

TABOUTTO(i)

Moves the next character output pointer to column i. Blanks are not output and you stay in the current line unless i is in or beyond the right margin, in which case the current line is simply output.

TABOUTBY(i)

Like TABOUTTO(i) except i is used as a delta rather than a column number.

ENDLINEOUT()

Blanks the remainder of the current output line and transfers it to the file. If this is a V file, the line is output at its current length with no padding except for the right margin (if any). This is the function used by PRINT and others to pad and transfer incomplete lines.

TOPPAGE()

Meaningful only for output with page formatting or CCA. Causes the rest of the current page to be blank unless output is exactly at the top of the page. In the latter case, no operation is performed.

BLANKPAGE()

Like TOPPAGE but is perfectly willing to output an entirely blank page.

NOADVANCE()

Will not advance paper before printing the next line and will inhibit incrementing of the line count for the page when that print is done. Useful for underscoring, but does not work for TERMINAL files.

FORMCONTROL(c)

Specifies a form control character to be used with the next output line.

PRINTLIST(l)

The argument is a node2 list. Each element in the list is output with PRIN, and blanks are used to separate the items. The last line of the output is padded with blanks and forced out to the file.

PRINLIST(l)

Like PRINTLIST except that the last line is not padded or forced out.

PRINTINDEF($*,*x)

Like PRINTLIST except that there are an indefinite number of arguments instead of a list. The output items are separated by blanks, and the last line is padded and forced out.

PRINTINDEF($e,*x)

Like PRINTINDEF except that the last line is neither padded nor forced out.

PRINTINT(c,i)

The integer i is printed, right justified to column c if c>0. Otherwise, i is printed left justified to column -c. The line is padded with blanks and output. Interior parts of the line skipped over are blanked. If the next character pointer is already too far to the right, then printing occurs on the next line.

PRININT(c,i)

Like PRINTINT except that the remainder of the line is neither padded nor forced out.

PRINTFLT(c,f)

Like PRINTINT except the printed number is floating.

PRINFLT(c,f)

Like PRININT except the printed number is floating.

PRINTHEX(c,x)

Like PRINTINT except number is output in hex.

PRINHEX(c,x)

Like PRININT except number is output in hex.

PRINTGEN(c,q)

Like PRINTINT except the printed value may be anything. If an attempt is made to print a node, array, complex number, or ntuple with right justification, then a space will be printed and the structure will be output without justification.

PRINGEN(c,q)

Like PRINTGEN except that the last line is neither padded nor forced out.

Note: At present, we believe that the output primitives will be able to handle circular structure when the file is not SYM. This assumes that none of the symbolic output primitives will use any dynamically allocated space other than stacks. When a circular pointer is detected, "*CIRCULAR*" will be output instead of reprinting the structure.

## System Primitives

This section describes the primitives that use CP or CMS facilities other than I/O.

TIMER()

The value is a floating number that gives the value of the virtual CPU timer in microseconds.

NOW()

The value is a list in the format:

(month day year hour minute second)

RCMS(i)

Permanently returns control to CMS with the value i. Normal exit from the CRISP system.

CALLCMS(l)

CMS is called to perform a single subset command. The argument, l, is a list. Each top level element of l is turned into an 8 byte request table entry.

CALLCP(l)

This is equivalent to CALLCMS("CP#l). This causes CMS to pass the request through to CP for action.

SUBSET()

CRISP will turn terminal control over to the CMS subset handler. To re-enter CRISP, simply type RETURN.

MYNAME()

Returns the identifier name of the presently operating module.

SAVE(id | FDL)

Outputs the presently operating copy of CRISP. If the argument is an id, then the file will have that name, and will be of type MODULE on the A disk. If the argument is an

FDL, then only the identification portion will be used. It must specify a disk file, and the usage attribute ERASE is always assumed. When a module save generated by SAVE is reloaded, all I/O files will automatically be shut, and operation will continue in the top level supervisor process after reopening the terminal files ITERM and OTERM.

SUSPEND

The primitives necessary to suspend and resume a module and maintain its entire state have not yet been designed. Also, the primitives necessary to stack pseudo terminal input lines have not yet been designed.

# GENERAL PRIMITIVES

This section describes a variety of primitives that are available in the system. The descriptions are grouped by the kind of task performed and the type of arguments. Many of the primitives are not functions; they are forms that are handled specially by the compiler or are the names of macros or transforms. For primitives that are functions or pseudo functions, a declaration is given. If there is a special SL operator, it is mentioned.

## Bit Logical

All primitives described in this section are pseudo functions. They produce in-line code. Each treats its arguments as 32 bit strings and performs a logical operation on the bits.

INT FUNCTION INV(INT) SL=INV
INV returns the one's complement of its integer argument. MINUS (- in SL) may be used to compute the two's complement of an integer.

INT FUNCTION BAND(INT INDEF) SL=&&
BAND ands together all of its integer arguments to produce an integer value. The value of BAND with 0 arguments is 0FFFFFFFFX.

INT FUNCTION BOR(INT INDEF) SL=||

BOR computes the inclusive or of its arguments. The value of BOR with 0 arguments is 0.

INT FUNCTION BXOR(INT INDEF) SL=BXOR

BXOR computes the exclusive or of its arguments. The value of BXOR with 0 arguments is 0.

```
INV(0CX) is 0FFFFFFF3X
INV(0AX) is 0FFFFFFF5X
BAND(0AX,0CX) is 8
BOR(0AX,0CX) is 0EX
BXOR(0AX,0CX) is 6
```

## Arithmetic

All primitives described in this section are pseudo functions and may produce in-line code.

PLUS SL=+

PLUS has an indefinite number of numeric arguments. The value is the sum of the arguments' values. The value of PLUS with 0 arguments is 0. The type of value produced by PLUS is complex if any argument is complex, or float if any argument is float; otherwise, the value is of type integer.

MINUS SL=-

The value of MINUS is the negative of its argument. The value type is the same as the argument type.

DIFFER SL=-

DIFFER subtracts its second argument from its first argument. DIFFER(A,B) is equivalent to PLUS(A,MINUS(B)).

TIMES SL=*

TIMES has an indefinite number of numeric arguments. The value is the product of the arguments' values. The value of TIMES with 0 arguments is 1. The type of value produced by TIMES is complex if any argument is complex, or float if any argument is float; otherwise, the value is of type integer.

RECIP

RECIP computes the reciprocal of its numeric argument. If the argument is of type integer or float, the value is of type float. If the argument is complex, the result is complex.

QUO SL=/

QUO(A,B) is equivalent to TIMES(A,RECIP(B)).

INT FUNCTION IQUO(INT,INT) SL=//

The value of IQUO is the integer quotient of its two integer arguments. If the division is not exact, then the result is rounded towards zero.

INT FUNCTION REMAINDER(INT,INT)

The result is the remainder that results from integer division of the first argument by the second, assuming the quotient has been rounded towards zero. The sign of the

result is the sign of the dividend (first argument).

INT FUNCTION ENTIER(FLOAT)

ENTIER converts its float argument to an integer by truncating towards zero. ENTIER(F) is equivalent to DRIVE(INT,F).

INT FUNCTION ROUND(FLOAT)

ROUND(F) is equivalent to ENTIER(F+0.5) or ENTIER(F-0.5) as F is positive or negative.

MAX

MAX has an indefinite number of numeric arguments. The value is the largest. The value of MAX with 0 arguments is 80000000X, the smallest integer. If any argument is complex, the value is complex; if any argument is float, the value is float; otherwise, the value is integer. The norms of complex numbers are used for the comparison.

MIN

MIN has an indefinite number of numeric arguments. The value is the smallest. The value of MIN with 0 arguments is 7FFFFFFFX, the largest integer. If any argument is complex, the result is complex; if any argument is float, the value is float; otherwise, the value is integer. The norms of complex numbers are used for the comparison.

SIGN

The value of SIGN is -1 if the argument is less than 0, +1 if the argument is greater than 0, and 0 if the argument is 0. SIGN of a complex is 1 unless the argument is {COMPLEX 0.0 0.0}.

Note, none of the above (including the bit logical primitives) guarantee the order of evaluation or combination of their arguments. That is, A+(B+C) may evaluate and add A, B, and C in any of the possible permutations. If you desire to control the order of combination and/or the order of evaluation, write the forms as a sequence of statements and use variables to hold temporary results.

## Trigonometric

The functions SIN, COS, TAN, ARCTAN, LOG, LOG10, SQRT, EXP, and E.EXP all have float argument and produce a float value. All except EXP have one argument; EXP has two. E.EXP raises e to the power of its argument. Errors are detected in the standard way, e.g., SQRT of a negative is an error. The functions NSIN, NCOS, NTAN, NARCTAN, NLOG, NLOG10, NSQRT, NEXP, and NE.EXP are also available. Their arguments are of type number, and the value is of type number. This means that the argument and/or the the value can be complex.

SQRT(-1) is an error

NSQRT(-1) is {COMPLEX 0.0 1.0}

The rules that determine which value is returned when there

is a choice will be detailed later.

In SL, EXP is represented by **.

## Boolean Logicals

The primitives described in this section have boolean arguments. Interpreted as a boolean, any object other than NIL is equivalent to TRUE. All the primitives are pseudo functions that generate in-line code.

BOOL FUNCTION NOT(BOOL) SL=¬
The value of NOT is TRUE if the argument is NIL. Otherwise the value is NIL.

BOOL FUNCTION AND(BOOL INDEF) SL=&
The value of AND is TRUE if none of its arguments is NIL; otherwise, the value is NIL. The value of AND with 0 arguments is TRUE. The arguments are evaluated left to right. If one of the arguments is NIL, the remaining arguments are not evaluated.

BOOL FUNCTION NAND(BOOL INDEF) SL=¬&
NAND(b1 ... bn) is equivalent to NOT(AND(b1 ... bn)).

BOOL FUNCTION OR(BOOL INDEF) SL=|
The value of OR is NIL if none of its arguments is TRUE; otherwise, the value is TRUE. The value of OR with 0

arguments is NIL. The arguments are evaluated left to right. If one of the arguments is TRUE, the remaining arguments are not evaluated.

BOOL FUNCTION NOR(BOOL INDEF) SL=¬|
NOR(b1 ... bn) is equivalent to NOT(OR(b1 ... bn)).

BOOL FUNCTION IMPLY(BOOL,BOOL) SL=>>
IMPLY(A,B) is equivalent to OR(NOT(A),B).

BOOL FUNCTION IMPLIED(BOOL,BOOL) SL=<<
IMPLIED(A,B) is equivalent to OR(A,NOT(B)).

## Relationals

This section describes the relational operators. All are pseudo functions and may generate in-line code.

GR,GQ,LS,LQ SL= >,>=,<,<= (among others)
If either argument is non-numeric, the value is NIL. Otherwise, the value of the first argument is compared to the value of the second argument. If the first is greater than, greater than or equal, less than, or less than or equal to the second argument for the operators GR, GQ, LS, and LQ, respectively, then the result is TRUE. Otherwise, the result is NIL. If an argument is complex, its norm is used for the comparison.

EQ SL='='

EQ takes two arguments of any type and compares them for equality. For them to be EQ, one of the following must hold: (1) both are pointers at the same structure, (2) both are integers with the same value, or (3) both are floating with the same value. Thus, EQ is an exact equality test. The types of objects represented by pointers for which EQ is always TRUE for arguments with the same print name are character, identifier, name, subspecified name, boolean, and number or general objects with integer values in the range $-2^{20}$ to $2^{20}-1$. EQ should not be used with complex objects or other types of numbers kept under the cover types number and general.

NQ SL= ¬=

NQ(A,B) is equivalent to NOT(EQ(A,B))

BOOL FUNCTION EQUAL(GEN,GEN) SL='=='

EQUAL returns TRUE if its arguments are EQ, if both arguments are the identical type of node, array, or ntuple and the corresponding fields are EQUAL, or both arguments are numeric and their values are the same after type conversion. Otherwise, EQUAL returns NIL. EQUAL can enter a nonterminating loop if its arguments are circular structures.

BOOL FUNCTION NEQUAL(GEN,GEN) SL='¬=='

NEQUAL(A,B) is equivalent to NOT(EQUAL(A,B)).

## Dimensions

The following are pseudo functions for which the compiler may generate in-line code.

INT FUNCTION NUMDIM(ARRAY)
The value is the number of dimensions the argument has.

INT FUNCTION SIZEDIM(ARRAY,INT)
The value is the extent of the dimension selected by the second argument.

INT FUNCTION ARLN(ARRAY)
ARLN(A) is equivalent to SIZEDIM(A,1).

## LISP Primitives

The following primitives are borrowed from LISP. All work with binary node arguments. In all cases, the node2 arguments are assumed to be lists. A list is either NIL or a node2 object whose CDR is a list. Many generate in-line code and are pseudo functions.

NODE2 FUNCTION INTER(NODE2,NODE2) SL=INTER
The value is a list of the common elements in the two argument lists. EQUAL is used to determine equality of elements in the lists.

NODE2 FUNCTION UNION(NODE2,NODE2) SL=UNION

The value is the second argument augmented by members of the first argument list that are not already in the second argument list. The value of UNION(1,NIL) is a copy of 1 with duplicate members deleted.

NODE2 FUNCTION IN(GEN,NODE2) SL=IN

The list is searched for a CAR (or a CAR of some CDR) that is EQ to the first argument. If found, the value is the remaining list, starting with the item that was EQ. If no match is found, then the value is NIL.

IN("B,"(A B C)) is (B C)

NODE2 FUNCTION MEMBER(GEN,NODE2)

MEMBER is equivalent to IN except that EQUAL instead of EQ is used for the search.

NODE2 FUNCTION ON(NODE2,NODE2) SL=ON

ON searches the list argument for a CDR that is EQUAL to the first argument. If it is found, it is returned. Otherwise, the value is NIL.

NODE2 FUNCTION APPEND(NODE2,NODE2) SL=@

APPEND appends the first argument list to the second by copying the first and CONSing the elements to the second.

APPEND("(1 2 3),"(A B C)) is (1 2 3 A B C)

NODE2 FUNCTION DAPPEND(NODE2,NODE2) SL=@@

DAPPEND is the LISP NCONC. It is like APPEND except that

the first argument list is not copied. The eventual CDR that is NIL is changed to point at the second argument list. If the value of N is (1 2 3), execution of DAPPEND(N,"(A B C)) would produce the value (1 2 3 A B C), and the value of N would also be (1 2 3 A B C).

NODE2 FUNCTION FIND(GEN,NODE2)

The argument list is presumed to be a list of node2 structures. The list is searched for the first member whose CAR is EQUAL to the first argument. If no match is found, then NIL is the value.

FIND(3,"((1 A)(2 B)(3 C)(4 D))) is (3 C)

NODE2 FUNCTION FINDN(GEN,NODE2)

FINDN is like FIND except that EQ instead of EQUAL is used for the search.

GEN FUNCTION DGET(GEN,NODE2)

DGET is like FIND except that the CDR of the first matching structure is returned as the value.

DGET(3,"((1 A)(2 B)(3 C)(4 D))) is (C)

GEN FUNCTION DGETN(GEN,NODE2)

DGETN is like FINDN except the CDR of the first matching structure is returned as the value.

NODE2 FUNCTION REVERSE(NODE2)

The value is a copy of the input list with the top level items reversed.

REVERSE("(1 (2 3) 4)) is (4 (2 3) 1)

NODE2 FUNCTION DREVERSE(NODE2)

DREVERSE is like reverse except that the input list is not copied. The top level nodes (succesive CDRs) are reused. This destroys the original list.

NODE2 FUNCTION LIST(GEN INDEF)

LIST returns a list made of node2 objects of all of its arguments. For example, the value of LIST(1,2,3) is (1 2 3). This is equivalent to

CONS(1,CONS(2,CONS(3,NIL))).

INT FUNCTION LENGTH(NODE2)

The value returned is the length of the argument list.

GEN FUNCTION NTH(INT,NODE2)

The value returned is the nth (as specified by the first argument) item in the list. If the list is not long enough, then NIL is returned.

NTH(2,"(A B C)) is B

NODE2 FUNCTION NON(INT,NODE2)

The value is the argument list with N (=first argument value) CDRs removed. If the list is not long enough, the value is NIL. If N is zero or negative, the value is the original list.

NON(2,"(A B C D)) is (C D)

NODE2 FUNCTION NOFF(INT,NODE2)

The value is a list that is made up of the first n (= first argument value) top level items of the argument list. If the list is not long enough, then the result will have the same length as the list argument.

NOFF(2,"(A B C D E)) is (A B)

GEN FUNCTION LAST(NODE2)

The value returned is the last element in the argument list. If the list is empty, e.g. NIL, then NIL is the value.

LAST("((A B)(C D)(E F))) is (E F)

Delete Functions NODE2 FUNCTION deletefn(GEN,NODE2)

There are eight delete functions: DELE, DELEN, DDELE, DDELEN, DELETE, DELETEN, DDELETE, and DDELETEN. All have two arguments; the first is a general object and the second is a list. The DELE functions work with a list of node2 structures like FIND and FINDN, and the DELETE functions work with simple lists like IN and MEMBER. Names ending in N use EQ for the search, and others use EQUAL. Names beginning with DD reuse the nodes in the input list, hence they destroy some of it; the others copy where necessary. They remove all items in the list that match the search test.

GEN FUNCTION SUBST(GEN,GEN,GEN)

The first argument is substituted for the second argument in the third argument. EQUAL is used to detect the presence of the second argument. SUBST works recursively through the

third argument as long as its type is node2 structure.

SUBST(1,"(A),"((X A)(A)[NODE3 (A) B C](A))) is

((X . 1) 1 [NODE3 (A) B C] 1)

GEN FUNCTION SUBSTN(GEN,GEN,GEN)

SUBSTN is like SUBST except that EQ instead of EQUAL is used for the testing.

## MAPPERS

The mapping functions take two arguments: a list made of node2 objects and a func with the type GEN FUNC(GEN). The mappers are MAPIN, MAPINL, MAPON and MAPONL. MAPIN and MAPINL apply the func to the CAR of the list, then to the CAR of the CDR, then to the CAR of the CDDR, etc. MAPON and MAPONL apply the func to the list, the CDR of the list, then the CDDR, etc. MAPIN and MAPON return NIL. MAPINL and MAPONL return a node2 list of the values of the function calls. Given the definition,

GEN FUNCTION FOO$ABC(GEN X) PRINT(X+1);

MAPIN("(1 2.0 3),FOO$ABC) and

MAPINL("(1 2.0 3),FOO$ABC)

both print 2, 3.0, and 4. MAPIN returns NIL. MAPINL returns (2 3.0 4). (Recall that the value of PRINT is its argument.)

MAPON("(1 2.0 3),PRINT) and

MAPONL("(1 2.0 3),PRINT)

both print (1 2.0 3), (2.0 3), and (3). MAPON returns NIL.

MAPONL returns ((1 2.0 3) (2.0 3) (3)).

## Copiers

The following primitives are used to copy structures. COPY and COPYNODE should not be used on circular structures because they may enter a nonterminating loop. All are pseudo functions. MOVE and MOVENEW may produce some in-line code.

COPY

If the argument is not a node, array, or ntuple, the value of COPY is its argument. If the argument is a node, ntuple, or array, then a new structure with the same type is allocated, and each field is initialized to a COPY of the corresponding field in the argument object.

COPYNODE

If the argument is not a node, the value of COPYNODE is its value. If the argument is a node, then a new node with the same number of fields is allocated, and the fields are initialized to a COPYNODE of the corresponding fields in the argument object.

NOVALUE FUNCTION MOVE(GEN,GEN)

Both arguments to MOVE must be of exactly the same type of node, ntuple, or array (with precisely the same outer dimensions). Either or both arguments may be flattened

fields. The fields in the second argument are copied bit for bit into the corresponding fields of the first argument. Given the following declaration,

```
DEC X<A INT, B FLOAT>,
    X Y,
    Z<Q FLAT X, R X>,
    X ARRAY(*) N,
    FLAT X ARRAY(*) M;
```

then any of the following may appear as the first and/or second argument of MOVE: X, Y, Z_Q, Z_R, M[I] and N[J].

MOVENEW

If the argument to MOVENEW is not a node, ntuple, or array, the value is the argument. Otherwise, a new structure is allocated and the fields of the argument are moved, bit for bit, into the corresponding fields of the new object. The argument may be a flattened field.

## Evals

The eval forms are normally used by programs that compute IL expressions and then wish to evaluate them. The eval process is very expensive. Therefore, if a particular piece of IL is to be used several times, compile it. The compiler may be called from a program as described below.

All the eval forms have an optional argument of type handle. If present, the IL argument is evaluated with the handle used as the variable context. When this option is selected, a new process may be created, so beware of mucking about

with the external context of MYSELF.

All free references from the IL argument are to objects with global names. That is, local variables in the program containing the use of the eval form are not visible. If identifier names are used freely, then the default tailing conditions in force at the time of the evaluation are used.

GEN FUNCTION EVAL(GEN[,HANDLE])
The argument is evaluated, and its value is the value of EVAL. The value of EVAL(READ()) is 3 if (PLUS 1 2) is input.

GEN FUNCTION EVALQ(GEN[,HANDLE])
The argument is evaluated only "one deep" instead of two deep as with EVAL. Thus, the value of EVALQ(READ()) is (PLUS 1 2) with the above example input. The real use of EVALQ is evaluation in a different context. For example, the value of EVALQ(A$X+B$Y,H) is the sum of the values of the variables A$X and B$Y in the process selected by the handle, H.

GEN FUNCTION APPLY(GEN,NODE2[,HANDLE])
APPLY applies its first argument to the list of argument values in its second argument. Thus, APPLY(A,B) is equivalent to EVAL(A$B). The value of APPLY(LIST,"(A B)) is a two element list of the values of A and B when the apply form is evaluated.

GEN FUNCTION APPLYQ(GEN,NODE2[,HANDLE])

APPLYQ applies its first argument to the quoted items in the list. The value of APPLYQ(LIST,"(A B)) is (A B).

NAME FUNCTION COMPILE(GEN,ID,NODE2)

The first argument is an IL definition to be compiled; the second and third arguments are the default tail and ordered default tail list, respectively, that are to be used for the compilation. The value is the name object that is the name of the compiled definition.

NAME FUNCTION COMPILEX(GEN,ID,NODE2)

COMPILEX is like COMPILE except that the compiled definition is not protected from garbage collection after all references to the name of the compiled definition disappear.

## Traces

The set of tracing functions has not yet been determined.

# TREE STRUCTURED FILES AND THE DISK COMPILER

To aid in the construction of large programs, CRISP provides an extensive capability that allows development of symbolic program text within the CMS file system. The key features of this capability are:

- tree structured program files,
- library files,
- full declaration pass, and
- access to an editor during compilation.

Two kinds of files are used to construct CRISP programs: (1) files of file type CRISP that contain a sequence of program text, format commands, and library commands, and (2) files of file type INDEX that provide the branching nodes of the file tree.

An index file must be of file type INDEX. It contains a list. The elements of the list are either file names or file descriptor lists. The files named in this list are in turn other files of file type CRISP or INDEX. When the disk processor functions described in this section visit an index file, they automatically continue onward to visit each of the files mentioned in the index. Thus, starting from an index file, an entire file tree may be visited and processed.

A file of file type CRISP contains a sequence of top level

forms.

```
<file>::=$[<SL-top-level-form>;|<IL-top-level-form>]

<top-level-form>::=<default>|<implicit-form>|<declare>|
                   <definition>|<expression>|
                   <format-command>|<library-command>

<format-command>::=FORMAT [IL|SL][CAP];

<library-command>::=LIBRARY([$[file name|file descriptor]])
```

Format commands specify whether the top level forms following in this file are in SL or IL format, e.g., whether the forms obey SL or IL syntax. If the CAP option is used, then it is assumed that the bodies of all the following function, processor, macro, and generator definitions are written in CAP. This option relieves the necessity of using CAP-forms for the bodies and redundantly entering the definitions' value types. All non-index files have an initial implicit "FORMAT SL;" assumed at the beginning.

The library command specifies the names of a set of "library" files that are necessary to compile and/or execute the forms in this file. These files are themselves of file type CRISP or INDEX. In a way, a library file resembles an embedded index file. However, there are differences: (1) when files in a file tree are listed, the library files are not and (2) library files are compiled without editor support. Therefore, programs used as library files should be reasonably debugged.

In the above, provisions have been mentioned only for files

of file types CRISP and INDEX. Other file types can be used. However, if this is done, then the file can not be found from the file name alone; a file descriptor list must be used that specifically mentions the file type. The functions provided to operate with tree structured disk programs are:

```
RUN(f-d,mode)
BATCH(f-d,mode,{NIL|TERMINAL|PRINTER|F-d})
LISTING(f-d,mode)
COMBINE(f-d,mode,f-d)
VISITTREE(f-d,mode,novalue func(id,id,id))
MAKETREE(f-tree,mode)
GETTREE(f-d,mode)
```

where f-d is a file name or a file descriptor, mode is either a file mode or *, and f-tree is a file tree that is described below. The mode parameter is used as the default file mode in locating a file specified only by a file name. If it is *, then the file is looked for in the CMS-determined search order. When the system is trying to locate a file that is specified only by its name, the default file types INDEX and CRISP are tried in that order. Thus, if only the file name F is specified along with a mode parameter of * and the default search order is A, B, C, then the sequence of search is

```
F INDEX A
F CRISP A
F INDEX C
F CRISP C
F INDEX B
F CRISP B
```

Each of the tree traverse functions is described below.

RUN(f-d,mode)

The file located by f-d (and mode) is visited by RUN. If it is an index file, the files in the index list are visited in their order of occurrence, left to right. Thus, the entire tree that has f-d as its root is traversed in preorder. If the visited file is not an index file, then it is evaluated. The evaluation operation on a file follows four ordered steps: (1) turn all SL into IL, (2) for all l such that l is a library file do BATCH(l,mode,NIL), (3) operate the declaration pass, and (4) complete the compilation of forms in this file. During step 4, the top level forms that are definitions are compiled, and the top level forms that are expressions are evaluated, in the order of their occurrences. The name of compiled definitions and the values of evaluated expressions are output on the user's terminal. If any errors occur during steps 1, 3, or 4, the user is placed in edit mode with the entire text of this file available for correction. The text should be corrected and rewritten to the disk before compilation of this file is restarted. (The editor will be a subset of the CMS editor that has been made part of CRISP. The exact design of the subset has not yet been completed.)

If an attempt is made to visit a library, index, or other file that has already been visited by this call to RUN (or BATCH, etc.), then that file will be skipped. Therefore, no harm or lost time results from a file's being used as a library file more than once.

Each non-index file is assumed to start with the implicit sequence,

```
FORMAT SL;
DEFAULT USER,(USER,CRISP);
IMPLICIT;
```

Operation of format, default, and implicit forms has no effect on conditions outside of the file.

BATCH(f-d,mode,{NIL|TERMINAL|PRINTER|f-d})

BATCH visits files and handles libraries in exactly the same manner as RUN. The essential difference is that BATCH does not provide editor support for errors. The third argument specifies the output device for the names of compiled definitions, expression values, and error and warning messages. TERMINAL, PRINTER, and f-d do the obvious things; NIL squelches output except for error and warning messages, which are printed on the user's terminal. If the third argument is a file name (trival case of a file descriptor), then the output is to a file so named with file type LISTING and file mode A. All formating, default, and implicit form assumptions are the same as for RUN.

LISTING(f-d,mode)

LISTING visits files in the same order as RUN except that library files are not visited. Each file is copied, line for line, to the virtual line printer. Pages are numbered, and the output for each file starts on a new page. A table of contents is provided at the back of the listing and gives the files and their page numbers. Neither RUN nor BATCH provides its own listing facility; LISTING does.

COMBINE(f-d,mode,f-d)

COMBINE visits files in the same order as RUN except that library files are not visited. The file specified by the third argument (if it exists) is erased and then opened for output. If only a file name is specified, then the file type is assumed to be CRISP and the file mode is assumed to be A. Each non-index file that is visited is copied into the file specified by the third argument. In front of each copied file, three lines are inserted:

```
FORMAT SL;
IMPLICIT;
DEFAULT USER,(USER,CRISP);
```

This ensures that the interpretation of the program text is uniform whether or not the file has been combined.

VISITTREE(f-d,mode,novalue func(id,id,id))

VISITTREE visits files in the same order as RUN except that library files are not visited. As each file, index or other, is visited, the third argument is called with the file name, file type, and file mode as arguments. The file is opened and selected for reading before the func is called and closed after the func returns. VISITTREE is a tool in the parts kit for users who wish to build their own file tree processors.

MAKETREE(f-tree,mode)

MAKETREE provides a method of generating a file tree; that is, a method of generating the index files that make the file tree. An f-tree is either an f-d or an (INDEX f-d $f-tree). The latter specifies an index file whose name is

the f-d following the word INDEX. The $f-tree forms are the sub trees to be reached from this index. The second argument, mode, is the disk on which to write index files when only their name is specified. For a simple example,

```
MAKETREE("(INDEX I
           A
           (INDEX J B C)
           D), "A)
```

The index file, (I INDEX A), would be created with the contents (A J D), and the index file (J INDEX A) would be created with the contents (B C). If an index file already exists before output by MAKETREE, then it is first erased and then rewritten.

GETTREE(f-d,mode)

GETTREE starts from the specified file and returns the f-tree with that file as the root node. Assuming that all files are on the A disk and that all non index files are of file type CRISP, then the value of GETTREE("I,"A) given the above use of MAKETREE is

```
(INDEX (I INDEX A)
       (A CRISP A)
       (INDEX (J INDEX A)
              (B CRISP A)
              (C CRISP A))
       (D CRISP A))
```

# MEMORY MANAGEMENT FACILITY

The CRISP memory management facility comprises the allocation mechanism and the garbage collector. The following paragraphs describe the allocation strategy and mechanism and the organization of the garbage collector.

A principal goal of the system is to allow user and system components to make efficient use of a virtual memory resource. Because the IBM 370 has a large address space ($2^{24}$ bytes), the problems of address management and memory management may be partially decoupled. For instance, a space with fixed maximum size may be fully allocated even though only a small fraction is generally in use. This is reasonable whenever the amount of "wasted" address space does not grow too large. In the CRISP system, many of the data spaces, such as pushdown stacks and small integers, will be statically allocated to a maximum size at core image generation time.

Another major feature of allocation is the concept of selectable spaces. For instance, there may be several node2 spaces, say NODE2$X, NODE2$Y, and NODE2$Z. Then, the user (or a system component) could say USESPACE(NODE2$Y) and further CONSing of binary nodes would take place in NODE2$Y space. Thus, program segments building structures that will be used together frequently may create and select spaces for that particular task. In many situations, this will help

alleviate page thrashing when something is known about the program's dynamics. The value of USESPACE is the space previously selected for the same structure kind.

New spaces may be defined dynamically by using the function NEWSPACE (see page 293). The creation of a new space requires (among other things) the specification of functionals that provide the allocation policy, planning policy, update policy, and the moving policy for the space. Through the use of these functionals, NODE2$X could be folded, NODE2$Y could use a simple avail list, and NODE2$Z could use a smart CONS. Thus, different spaces for the same kinds of structure may use quite different management techniques. The description of the garbage collector (below) will elaborate on the use of these policy guides.

## Core Maps

Each space is made up of a set of regions and each region is made up of a set of contiguous quanta of memory. A quantum is a 4096 byte page of memory. (Maybe 2048 bytes .. not yet decided.) The system's memory map is also built as a three level hierarchy corresponding to quanta, regions, and spaces. The quantum core map (QCM) contains one byte of information for each quantum. The byte specifies the space kind. The possible space kinds are:

- NODE1 ... NODE8,
- IDENTIFIER,

- •CHARACTER - identifiers with one character names,
- •INTEGER - integer values of general data,
- •FLOAT - real values of general data,
- •COMPLEX,
- •ARRAY,
- •NTUPLE,
- •NAMEA - program reference space ... binding cells,
- •NAMEB - program reference space ... declarative info,
- •PDP - pointer stack,
- •PDN - numeric stack,
- •BPS - binary program space,
- •HANDLE - handles for retained contexts,
- •HEAP - heaps,
- •SMINT - small integer values of general pointers.

The region core map (RCM) is made up of region control blocks (RCBs). An RCB includes:

- •Beginning quantum of the region,
- •Ending quantum of the region,
- •Current ending address in the region,
- •Link to other RCBs for the space.

The space core map (SCM) is built up from a space control block (SCB) for each space. An SCB includes:

- •Space name,
- •Space kind,
- •Space property (communication cell),
- •Region link - link to RCBs for this space,

- Max regions - the number of regions to be allocated before automatic garbage collection,
- Region size - number of quanta per region,
- Space link - link to other SCBs,
- Allocation policy function,
- Prune policy function,
- Planning policy function,
- Update policy function,
- Moving policy function.

## Garbage Collection

The garbage collector is customarily invoked by an allocation policy function. When a structure creator such as CREATE or CONS cannot find enough room to build a structure in the currently active region of the selected space, the allocator for that space can either attach a new region or initiate garbage collection.

The garbage collector is organized into six logical phases: marking, pruning, planning, updating, moving, and fixing.

The marking phase locates each structure that is active and marks it for retention. For some space kinds, there is a mark-loop driver. For instance, for the pointer stack, the loop driver would initiate marking of each structure addressed by a stack item. The loop driver is the space's marking policy function. Also, for each space kind, there

is a marking function. The marking function is called with an active structure as an argument. For each pointer field in that structure, the addressed structure is also marked. A universal mark function is provided. It will mark a structure of any kind by determining the structure's kind and passing the structure to the marker associated with that space kind.

The pruning phase of garbage collection "restrings" certain system link structures. Among these are the identifier hash links and the system property lists. The pointer items forming these link structures are not chased during the mark phase.

The planning phase determines where each structure will be and the amount of structure remaining in each region at the conclusion of garbage collection. The planning policy for each space is provided by its SCB. For instance, for arrays (which are normally compacted), each header will be set to the forwarding address of the array; for nodes (which are normally folded), folding takes place and the forwarding addresses of the nodes that are moved are left in the vacated sites.

The updating phase changes each pointer field to the address that the addressed structure will occupy after garbage collection. Essentially, each field chased during the marking phase must be updated. For each space, an update-loop driver visits each active structure in the space

and updates the appropriate fields. The loop driver is the update policy function. For each space kind, there is an update function that takes as an argument a structure and returns as its value the new address. The universal update function takes as an argument any structure. It determines the space kind and applies the corresponding specific update function.

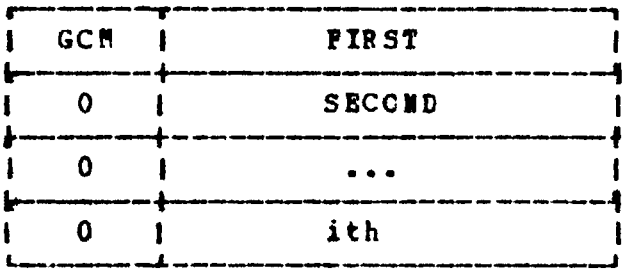
The moving phase is controlled by the moving policy function specified for each space. Structures that are compacted, such as arrays and binary programs, are moved to their final resting places.

The fixing phase goes through the RCM and deletes all RCBs that correspond to regions holding no active structure. The associated quanta are also released. Also, each space for which a delete request has been received and for which there are no RCBs is released.

## Space Formats

The following pages describe each space kind in more detail. A graphic representation of the structures in each space is included. The breakdown of elements is into bytes, half words, or full words as appropriate. Ellipses represent indefinite numbers of occurrences, and GCM and MOVE stand for garbage collector mark and forwarding address fields, respectively.

NODE1 ... NODE8 SPACE




4*i byte entries (for NODEi spaces)

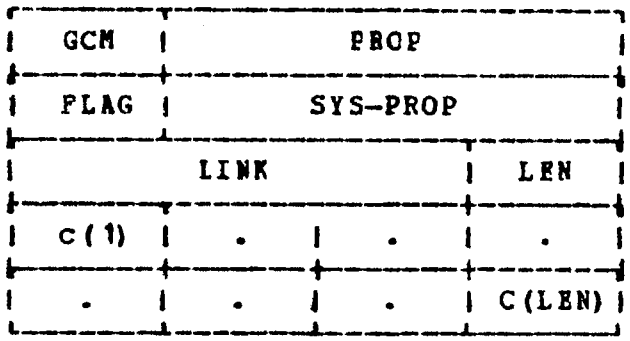
allocation - multiple selectable spaces
region=quantum
CONS

accessors - CAR ... CDDDDR, FIRST ... EIGHTH

gc algorithms
loop -
mark - FIRST ... ith
prune -
plan - fold
udate = mark
move -

The default garbage collection algorithm for node spaces is folding. However, other management strategies are provided for. The first two words of each quantum are reserved for use by the various allocation and management tasks. These words are used for such things as avail lists and move / no move breaks after folding.

IDENTIFIER SPACE




FLAG in {regular,special,genid}*keyword*type-name

12+4*(LEN+3)//4 byte entries

allocation - single space
single region
GENID, COMPRESS, STRING2ID

accessors - PROP, EXPLODE, ID2STRING

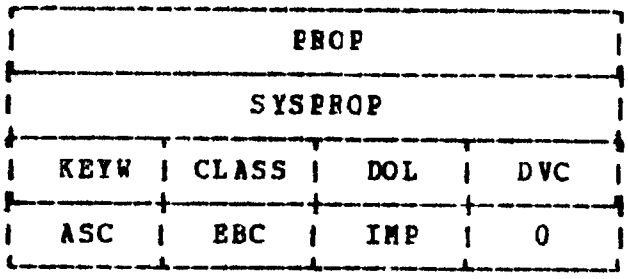
gc algorithms
loop - if PROP then mark self & PROP
mark -
prune - SYS-PROP chain
plan - copy thru LINK if appropriate
update - PROP
move -

There are four types of identifiers. Identifiers with one character names are stored in character space. In id space the three kinds are: regular; special ids, which must be printed using the "$'" mechanism to avoid ambiguity when reread; and genids, which have been created without specific names. The SYS-PROP field is the chain to all global names that have this identifier as their first name. LINK chains together all ids with the same hash address to facilitate searching and all genids. LEN specifies the number of characters in the identifier's print name. C(1) ... C(LEN) are the EBCDIC characters that are the name of the identifier. Enough pad characters are used to make the structure an integral number of words. LEN for a genid is 8 to allow for later name generation.

During the plan phase of the garbage collection, a decision is made whether to reallocate id space. The decision is based upon the number of new ids created since the last garbage collection, how much space is left, and how long it has been since the space has been reallocated. If it is decided to reallocate identifiers, a new id space is created and the marked ids are moved there. The copying is through

the hash links. This assures that hashed searching for an id will not page thrash.

## CHARACTER (IDENTIFIER) SPACE




CLASS in BLANK*DELIM*DIGIT*HEX*SUPER*ALPHA*LCASE

DOL in NEEDDOLLAR*CAUSEDOLLAR

DVC in TTY*TI*PDP10*TD*TN

IMP in {GENERAL|INTEGER|FLOAT}

16 byte entries

allocation - single static space
single quantum=region
COMPRESS, STRING2ID

accessors - PROP, EXPLODE, ID2STRING
qc algorithms
loop - prop
mark -
prune - SYS-PROP chain
plan -
update - prop
move -

Character identifier space contains the structure of identifiers with one character print names. The space is statically allocated so that the id's print name may be computed by its relative position in the space. KEYW is true if the identifier is an SL keyword. CLASS is used by the token parsing routines to classify characters. DOL tells whether this identifier needs to be printed with the $' mechanism to be correctly reread and whether the inclusion of this character in another identifier's name causes that identifier to be printed with the $' mechanism. DVC tells whether this character has a graphic representation on the named output device. ASC is the equivalent ASCII code for this EBCDIC character. EBC is the EBCDIC code for the ASCII character. IMP is the implicit type associated with this character.

## INTEGER SPACE


4 byte entries

allocation - multiple selectable spaces
region=quantum
INTEGER2GENERAL

accessors - GENERAL2INTEGER

gc algorithms
loop -
mark -
prune -
plan - fold
update -
move -

Integer space contains fixed-point values of number or general data. The first two words of each quantum are used to aid the garbage collector and allocator mechanism. See description of node space (page 272) for more information. An additional 32 word (1024 bits) bit vector is also left at the beginning of each quantum to be used as a mark table.

## FLOAT_SPACE


4 byte entries

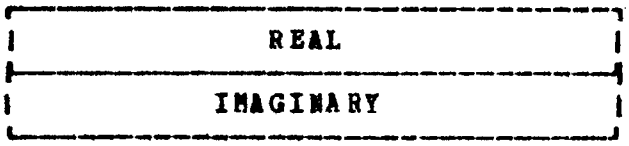
allocation - multiple selectable spaces
region=quantum
FLOAT2GENERAL

accessors - GENERAL2FLOAT

gc algorithms
loop -
mark -
prune -
plan - fold
update -
move -

Float space contains floating-point values of number or general data. The first two words of each quantum are used to aid the garbage collector and allocator mechanism. See description of node space (page 272) for more information. An additional 32 words (1024 bits) bit vector is also left at the beginning of each quantum for use as a mark table.

## COMPLEX SPACE




8 byte entries

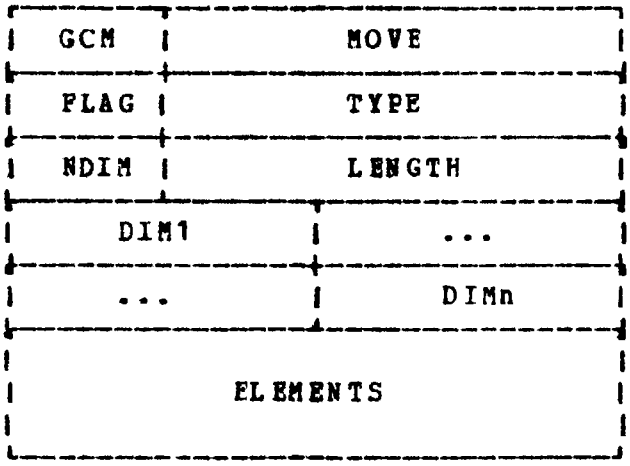
allocation - multiple selectable spaces
region=quantum
COMPLEX

accessors - REAL, IMAGINARY

qc algorithms
loop -
mark -
prune -
plan - fold
update -
move -

Complex space contains pairs of floating point numbers. The first two words of each quantum are used to aid the garbage collector and allocator mechanism. See description of node space (page 272) for more information. An additional 32 words (1024 bits) bit vector is also left at the beginning of each quantum for use as a mark table.

ARRAY SPACE




FLAG=MARKP*FLAT

12+4*(NDIM+1)//2+4*(LENGTH+3)//4 byte entries

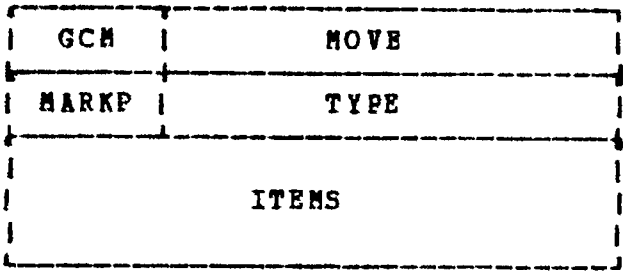
allocation - multiple selectable spaces
variable length regions
CREATE

accessors - TYPEP, SIZEDIM, NUMDIM, subscripting

gc algorithms
loop -
mark - pointer elements and TYPE
prune -
plan - set move to forwarding address
update = mark
move - compact

An array is constructed so that the first element begins on a word boundary and padder is added so that the array ends on a word boundary. The flag field specifies whether any of the elements are pointers and whether they are flat ntuples. In any event, if the MARKP bit is not set, then the array does not need to be marked from or updated by the garbage collector. The length field specifies the exact number of bytes occupied by the array elements.

## NTUPLE SPACE




8+4*(length+3)//4 byte entries

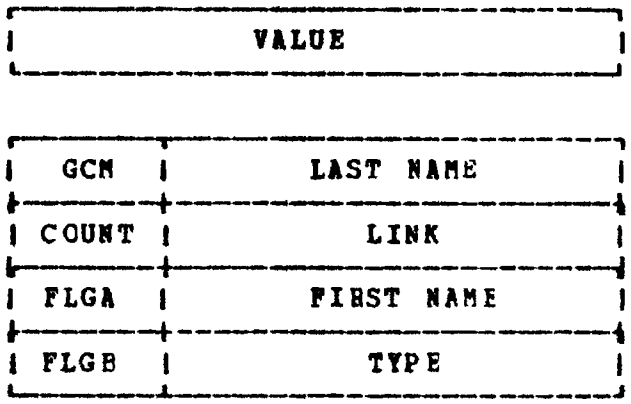
alloation - multiple seletable spaces
variable length regions
CREATE

accessors - TYPEP, item-names

gc algorithms
loop -
mark - pointer items and TYPE
prune -
plan - set MOVE to forwarding address
update = mark
move - compact

An ntuple is always padded to occupy an integral number of words. Ntuple items are marked and updated only if MARKP is set. In this case, the TYPE information is consulted as to which items are pointers and thus must be chased. The ntuple length is also derived from the type field.

## NAMEA and NAMEB SPACES




```
FLGA in {SYNONYM|MACRO|GENERATOR|TRANSFORM|SPACE|
         VARIABLE|FUNCTION|PROCESSOR|NAME|FAST|
         QUOTE|FREE}

FLGB in {BPS|SYS|NUMBER|POINTER}*PROTECT*HIDDEN*
        FREEABLE



4 byte entries in NAMEA
16 byte entries in NAMEB

allocation - single space
             single region
             MAKENAME, NEWSPACE

accessors - FIRSTNAME, LASTNAME

gc algorithms
  loop - if ¬FREE|COUNT¬=0 then mark
  mark - TYPE, FIRSTNAME, LASTNAME,
         if FLGB in POINTER then VALUE
  prune - through LINK with id prune
  plan -
  update = mark
  move -
```

The name spaces contain the objects referenced by global names. Namea contains the (shallow bound) value cells for global variables, code pointers to functions, etc. Associated with each one word object in namea is a four word object in nameb that is the compiler's and assembler's symbol table. All objects in namea space are covered by base registers. Therefore, the value cells may be directly referenced from code. (True only for RX format instructions; see section on General Purpose Registers, page 297.) FLGA denotes the object's type without subspecification, e.g., function, space, variable. For types that require subspecification (variable, processor,

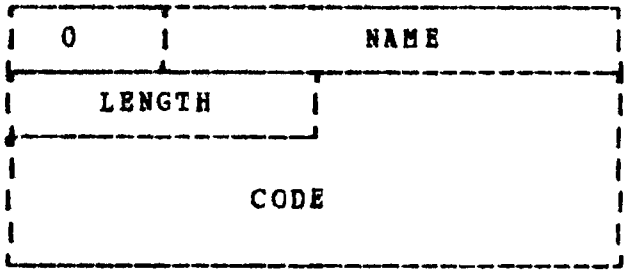
and function), the additional information is provided by the field, TYPE. The FLGA value, NAME, denotes a global name that has been generated but not declared or defined. Besides the subtypes of name, name space contains objects for non-numeric constants and special purpose code chunks (called FAST). Name objects that have not been allocated are marked FREE.

The system maintains an avail list of free objects in this space, linked through the LINK field. All global objects with the same first name are strung together (starting from the identifier) through the LINK field. In a similar maner, all quotes are also strung. In all cases, the LINK field is a nameb pointer.

FLGB contains information for the garbage collector and the declaration mechanism. A FREEABLE indication says that this structure may be reclaimed as FREE as soon as COUNT is 0 and no pointer references it. HIDDEN indicates that this structure has been redefined with different attributes and, therefore, another structure may exist with the same first and last name. The PROTECT indicator signals a vital system object that may not be redefined. See section on Name Primitives (page 123) for a description of the unlocking mechanism. The other subfield in FLGB describes VALUE, the corresponding object in NAMEA.

Objects in name spaces are never moved. The COUNT field specifies the number of references to the object from assembled code. It is, therefore, necessary neither to mark from nor update address computation in compiled code.

## BINARY PROGRAM SPACE




LENGTH byte entries <= 4096 bytes

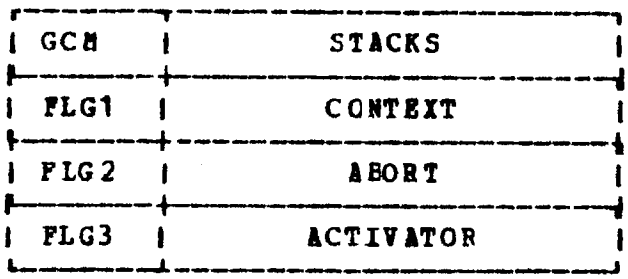
allocation - multiple selectable spaces
variable length regions
GETBPS

accessors - EXCISE, invocation

gc algorithms
loop -
mark -
prune -
plan - update loc(NAME) to forwarding address
and move
update -
move -

Binary programs may be a maximum of one core page in length. Thus, the compiler and assembler do not have to face problems of multiple code bases. The NAME field locates the name structure that defines the program's argument and value types. When code is moved, the namea word associated with the program is updated to the new location. The only pointer references from code are to the stack that is automatically chased during loop phase and to name structures that have count fields, so the references do not need to be chased from the code.

HANDLE SPACE




FLG1 in {STACK|HEAP}*VARACT*CODEACT*KILLED

16 byte entries

allocation - single static space
single region
STARTPROC, COPYPROC

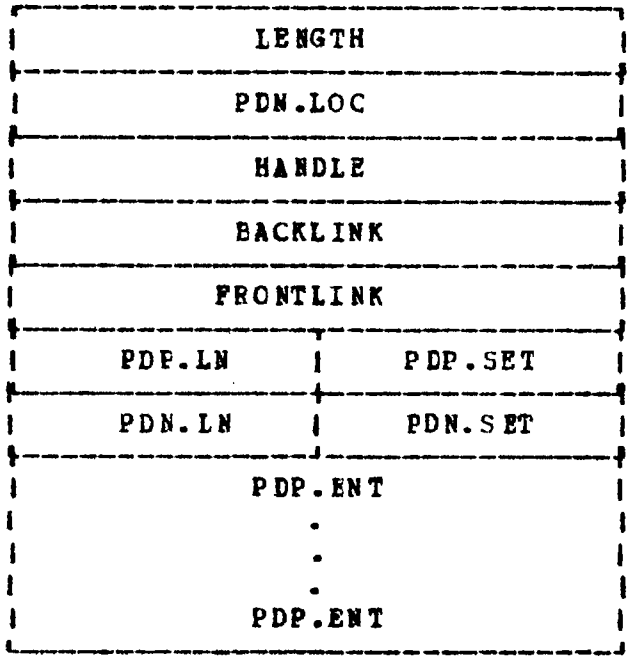
accessors - CONTEXT, ABORT, ACTIVATOR

gc algorithm
loop - if CODEACT
mark - CONTEXT, ABORT, ACTIVATOR, STACKS
prune -
plan - link free list
update = mark
move -

An object in this space is a control block for a process. If this entry is in use, then STACKS points to a block on the pointer stack (PDP) if FLG1 is STACK, and points to the process heap if FLG1 is HEAP. If FLG1 is CODEACT, this object is associated with the process that is currently active. VARACT indicates that this process is part of the total context of the process that is currently active. KILLED indicates that this process may no longer receive a program counter. However, it is not necessary for a process to be KILLed in order to be garbage collected; it is necessary only that this process not be referenced. CONTEXT, ABORT, and ACTIVATOR are pointers to other objects in handle space or NIL. FLG2 and FLG3 are markers used by the process switching algorithms.

Objects in handle space are not moved by the garbage collector. Free entries are collected on an avail list that is linked through the CONTEXT field.

POINTER STACK SPACE (PDP)




LENGTH byte entries

allocation - single static space
fixed size region
EVAL, EVALPROC

accessors - assembled code

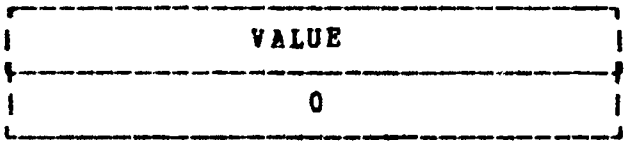
gc algorithm
loop -
mark - pointers in PDP.ENT
prune -
plan -
update = mark
move -

A block on the pointer stack (PDP) is, together with an object in handle space and a block on the number stack, a complete process variable and control state. The blocks on the two stacks may be packaged together and stored in the process heap. Only the top-most block on PDP is associated with the process in execution. This block grows and shrinks as the process computes, and its size may be determined by examining the value of the pushdown pointer register. All other blocks in PDP are stationary with a length of LENGTH bytes (always a multiple of 4). PDN.LOC is the location of the number stack block associated with the same process, and HANDLE is a pointer at the handle object for the process. BACKLINK and FRONTLINK are two-way threads that allow some of the process control primitives and the garbage collector

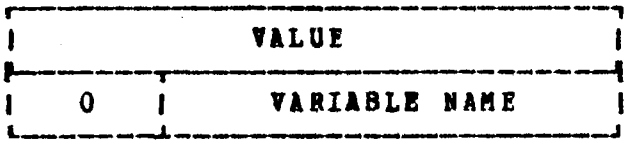
to work their way around the stack blocks. PDP.LN equals LENGTH and is the length of the block. PDP.SET is the placement for the pushdown pointer register, relative to the beginning of the block, when the process resumes execution. PDP.LN is the length of the associated block on the number stack, and PDN.SET is the placement of the numeric pushdown pointer register, relative to the beginning of that block, when the process resumes execution.

The PDP.ENT fields are either eight or sixteen byte entries. They are used for local variable bindings, temporaries, global binding saves, return addresses, and failsets.

The format for local bindings and temporaries is:



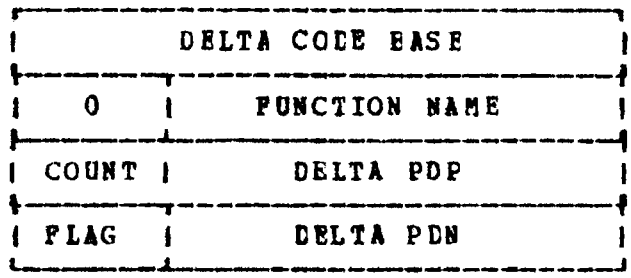

Global binding saves are used as part of the shallow binding mechanism of global variables. In general, the latest active binding is in named space, and old, saved values are on the stack. The format is:




The format of a return address is:


DELTA CODE BASE is the distance past the beginning of the function (in bytes) at which the function call was made. The high order 8 bits may contain the garbage generated by the EALR operation. Function returns are distinguished from global binding saves by examination of the name pointed at.

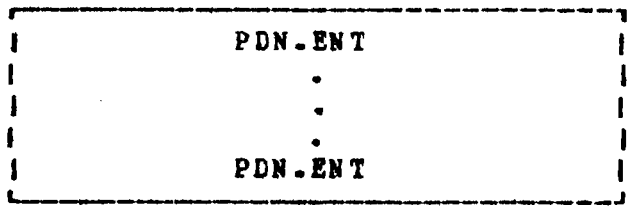
The format of a failset is:




The first two words have the same format as a return address and tell the unwrap mechanism where to resume execution. The DELTA PDP and DELTA PDN fields specify where to realign the pointer and number pushdown registers when execution is resumed at the indicated spot. The COUNT and FLAG fields are used by the TRY form to mark the number of trial expressions already evaluated and the kind of unwraps to catch, respectively.

To allow proper operation of the program check handler, it is conventional that even-numbered (second and fourth) words of PDP.ENTs be zeroed when they are popped.

## NUMBER STACK SPACE(PDN)




variable length entries

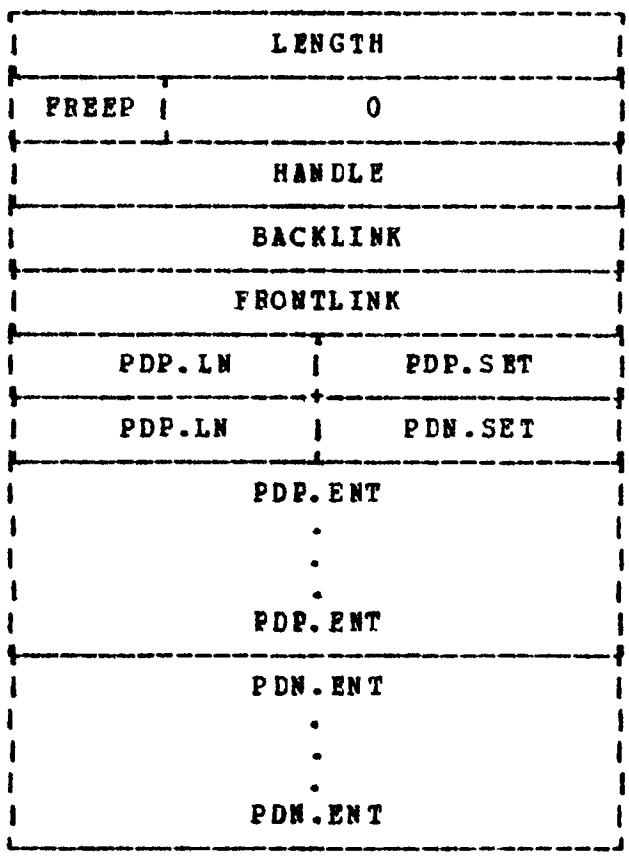
allocation - single static space
fixed size region
EVAL, EVALPROC

accessors - assembled code
gc algorithm
loop -
mark -
prune -
plan -
update -
move -

The length on a block on the number stack is determined by an associated block on the pointer stack (PDP). The topmost block is associated with the process currently in execution, and its length is determined by the number push down register. See description of PDP (page 285) for more information.

The entries in a PDN block (PDN.ENT) are 32 bit numbers, either integer or floating; only the code that put them there knows which.

PROCESS_HEAP




LENGTH byte entries

allocation - single static space
single region
MAKEPROC, COPYPROC

accessors - process switching primitives

gc algorithms
loop -
mark - from pointers in PDP.ENTs
prune -
plan -
update = mark
move - compact

The Process heap holds objects comprising a pointer and number stack pair. The formats of objects in this space and PDP and PDN are chosen so that stacks may be quickly saved and restored. The formats of PDP.ENT and PDN.ENT are described on the pages detailing stack layouts. LENGTH is the length of the heap object in bytes (always a multiple of 4). FREEP specifies whether the block is allocated. BACKLINK and FRONTLINK are two-way threads to other blocks.

The heap is maintained by the buddy system between garbage collects and allocated with a first fit strategy. At garbage collection time, the entire space is compacted, and only those blocks whose HANDLEs are marked are kept.

PDP.LN and PDN.LN are the lengths of the saved pointer and number stacks, respectively. PDP.SET and PDN.SET are the relative placement points for the pushdown pointer registers when this process resumes execution. HANDLE points at the handle object for this saved process.

# SPACE PRIMITIVES

This section describes the facility that manages data space utilization, the primitive that creates new spaces, and some of the primitives used by the system to allocate structures within a space. The structure allocation primitives for users are described in the section on data primitives (page 119). The section on memory management facility (page 266) describes the individual data spaces, their formats, and the garbage collector.

## Space Names

Global names, e.g., NODE2$X, are used to name spaces. The first name of the global name is a space kind, and the last name may be any identifier. The names of the space kinds are NODE1, NODE2, NODE3, NODE4, NODE5, NODE6, NODE7, NODE8, IDENTIFIER, CHARACTER, INTEGER, FLOAT, COMPLEX, ARRAY, NTUPLE, NAME, PDP, PDN, BPS, HANDLE, HEAP, and SMINT. Of these space kinds, CHARACTER, NAME, PDP, PDN, HANDLE, and SMINT are statically allocated. That is, there is exactly one space of each kind, and that space is allocated to its maximum size at system generation time. Although there is only one identifier space, its size may vary in time. Heap spaces may be created and destroyed. Therefore, neither identifier nor heap spaces are statically allocated.

When the system is generated, there is one of each kind of space except that there are two heap spaces. The names of these spaces are the space kind tailed with SYSTEM. For example, ARRAY$SYSTEM and BPS$SYSTEM are space names. The second heap holds process states and is named HEAP$PROCESS.

## Selectable Spaces

Spaces of the kinds node1, node2, node3, node4, node5, node6, node7, node8, integer, float, complex, array, ntuple, and bps are selectable. There may be more than one space of each of these kinds, and, for each kind of selectable space, one of them is said to be selected at any given moment. By convention, all structures are allocated in a selected space. Variables (of type name) whose first names are selectable space kinds and whose tails are SELECTED determine the currently selected spaces. Thus, all CONSing of binary nodes is done in the space whose name is the value of NODE2$SELECTED.

If a program has created several spaces of the same kind and selects from one to another, then a problem can arise when a fail or exit primitive is executed; when control is resumed at a try, the wrong space may be selected. To handle this problem, some sequence such as

```
TRY(...,BEGIN NODE2$SELECTED:=a NODE2 space;
              ....
              END,...)
```

may be used. The user can write a macro or transform to

generate the protection blocks around the try terminals. (The current I/O file selection may be protected by a similar mechanism.)

Warning: there is very little protection or error checking on the SELECTED variables; they must be used very carefully or unrecoverable program checks may result.

## NEWSPACE

NEWSPACE$CRISP creates new spaces or modifies the space control block of an existing space. The declaration is:

```
NAME FUNCTION NEWSPACE(NAME,INT,INT,INT,
                       GEN FUNC (NAME),
                       NOVALUE FUNC(NAME,INT),
                       NOVALUE FUNC(NAME),
                       NOVALUE FUNC(NAME),
                       NOVALUE FUNC(NAME))
```

The arguments, in order, are the space name, number of quanta per region, maximum number of regions for the space, space property value, allocation policy function, prune policy function, plan policy function, update policy function, and the moving policy function. Only selectable spaces and heaps can be created with NEWSPACE. The argument to each of the funcs (when called by the garbage collector) is the space name. If any func is NIL, the garbage collector does not call the function associated with the space during the corresponding phase. If the space does not already exist, a space control block is built for the new space. If it does exist, the call to NEWSPACE updates the SCB. Only the non-zero integers and the non-NIL funcs are placed into the existing SCB.

A general purpose allocation function is available:

```
GEN FUNCTION ALLOCATE$CRISP(NAME,INT)
```

The second argument is ignored. A region of appropriate size is added to the space, and a pointer at the first byte of the region is returned. In general, the second argument to an allocation policy function is the size (in bytes) of the structure that is being allocated. (This allows for regions of variable size.) Usually, the allocation policy function is called only when a structure will not fit in the space without an additional region or garbage collection. ALLOCATE adds a region only if the maximum number of regions allowed for this space has not been reached. If it has reached maximum size, the garbage collector is invoked. If no regions are reclaimed for this space, an error is induced.

The mark, prune, plan, update, and move policy functions are as described in the section on garbage collection (page 269). A space may be entirely reclaimed if it contains no structure and if its name has been hidden by the function HIDENAME (see section on name primitives, page 123).

## Allocation Primitives

Most of the primitives for allocating structures in data spaces are described in the section on data primitives (page 119). The rest are described below.

```
NODE1 FUNCTION CONS1(GEN)
NODE2 FUNCTION CONS2(GEN,GEN)
    .....
NODE8 FUNCTION CONS8(GEN,GEN,GEN,GEN,GEN,GEN,GEN,GEN)
```

For each kind of node space, there is a CONS function. At compile time, the pseudo function, CONS, determines which actual CONSn to use. All CONSn functions have the last name CRISP.

```
ID FUNCTION MAKEID$SYSTEM(INT,BOOL,GEN,INT)
```

The first argument is the number of characters in the print name. The second argument is TRUE if this id needs to be printed with the $' mechanism in order for it to be re-readable. The third argument points to the structure containing the name (a string, for instance), and the fourth argument is the offset from the pointer to locate the first byte of the name. If an identifier by this name exists, it is returned as the value. If not, a new identifier with the specified name is created.

```
NUMBER FUNCTION INTEGER2NUMBER$SYSTEM(INT)
NUMBER FUNCTION FLOAT2NUMBER$SYSTEM(FLOAT)
```

Each of these functions places its argument in an integer or float space and returns a pointer at it.

```
ARRAY FUNCTION MAKEARRAY$SYSTEM(GEN,BOOL,INT INDEF)
```

The first argument is the array type, the second is TRUE if the array is flat, and the indef ints are the extents of the dimensions. The array type must have been hashed by one of the assembler pseudo functions. The new structure is uninitialized.

NTUPLE FUNCTION MAKENTUPLE$SYSTEM(NAME)

The argument is the ntuple type. The created ntuple is not initialized.

NAME FUNCTION GETNAME$SYSTEM()

The value is a pointer at a name structure that has not been initialized.

GEN FUNCTION GETBPS$SYSTEM(INT)

The argument is the length of the bps area needed to hold a program image, including its header. No initialization is done. The value is a pointer at the first byte of a home for the image.

HANDLE FUNCTION GETHANDLE$SYSTEM()

The value is an uninitialized handle.

All of the above functions can call the garbage collector through the allocation function associated with the space.

# REGISTER ALLOCATION AND LINKAGE

This section is a "must" for anyone who wishes to write assembler language in the CRISP system. The topics covered are register allocation and usage conventions, the linkage mechanism, and the mechanism that binds and unbinds global variables.

## Register Allocation

This section describes the usage of the floating point and general purpose registers in the CRISP system. The contents of registers are not guaranteed over function calls.

### Floating point registers

The four floating registers, 0, 2, 4, and 6, are known by the mnemonics F0, F2, F4, and F6, respectively. By convention, if the last argument to a function or processor is a floating point number, then the value is passed in F0. If the value of a function is a floating point number, then it is returned in F0. Within a function, processor, etc., the floating point registers may be used for any purpose the programmer desires.

### General purpose registers

Table M gives the mnemonics and the contents (if constant) of the 16 general purpose registers. By convention, if the

Table M -- Register Contents and Mnemonics

| Register | Contents | Mnemonics |
|---|---|---|
| 0 | - | R0 |
| 1 | - | R1 |
| 2 | - | R2 |
| 3 | - | R3 |
| 4 | - | R4 |
| 5 | - | R5 |
| 6 | - | R6, LINK |
| 7 | - | R7, CB2 |
| 8 | - | R8, CB |
| 9 | 0 | R9, ZERO |
| 10 | - | R10, PDP |
| 11 | - | R11, PDN |
| 12 | 20000X | R12, SYS1, FNLK |
| 13 | 21000X | R13, SYS2, FNRT, QCM |
| 14 | 23000X | R14, NUM1, BIND |
| 15 | 24000X | R15, NUM2, UNBIND |

last argument to a function, processor, etc., is a pointer or an integer (including an indef count), its value is passed in R5. If the value is an integer or a pointer, it is returned in R5. Within a program, the programmer may use registers R0 through R6 for any purpose. R7 is also available in programs whose length does not exceed 4096 bytes. R8 is the code base. R7 is the second code base register used in programs longer than 4096 bytes. R6, also named LINK, is used as the "return address" register by the BALR or BCR command that transfers to the function call, function return, global variable binding, or global variable unbinding sequences described below.

R9 is also called ZERO, and its contents are always 0. Besides easy access to the constant 0, use of R9 makes possible a one instruction NIL test. Thus, "BXH r,ZERO,1" branches to location 1 if the contents of general register r

are strictly positive. "BXLE r,ZERO,1" branches to location 1 if the contents of general register r are less than or equal to 0. Recall, NIL is represented by a pointer at address 0, and all other pointers are strictly positive addresses. Therefore, if r contains a pointer, then BXH is a transfer on non-NIL, and BXLE is a transfer on NIL in register r. For this trick to work, the register ZERO must be an odd numbered general register.

The registers R10 and R11 are also named PDP and PDN and are, respectively, the pointers to the pointer stack and the numeric stack. By convention, both registers are operated 400X bytes behind their virtual top of stack locations at function entry. Only the function call, function return, and processing primitives ever increment or decrement the stack registers. All other stack allocation is done by the assembler, which assigns virtual locations at assemble time. Promiscuous mucking with the stack registers is a sure way to develop an unrecoverable program check.

The registers R12, R13, R14, and R15 are used by the system for a variety of purposes, and each has several names to reflect these usages. CRISP is loaded at byte address 20000X. Therefore, the four registers contain the addresses of the first, second, fourth, and fifth pages of the system. (These five pages are a system heap.) The first and second pages are used to hold such things as garbage collector tables and code sequences that cannot conveniently be operated in binary program space. These pages (and

registers R12 and R13) are named SYS1 and SYS2. The first several words of SYS1 contain the function call sequence. A function call is initiated by the command "BALR LINK,FNLK", where FNLK is another name for the register SYS1. The first several words of SYS2 contain the function return sequence. This sequence is initiated by any branch to the beginning of SYS2; for instance, the command, "BR FNRT", where FNRT is another name for the register R13. Another usage of R13 is to access the quantized core map, QCM. QCM is a 4096 byte table with each byte corresponding to a page of memory. The byte identifies the kind of space to which the page belongs. The first four bytes of QCM occupy the last four bytes of SYS2. Thus, to test whether pointer p addresses a page of space kind k, use the following.

```
L       R6,p;
SRL     R6,12;
LA      R6,4092(R6,QCM);
CLI     0(R6),t;
```

The condition code will be set to indicate the value of the test. Such code sequences as the above are normally generated by compiler and assembler macros. Since the bytes of QCM are always relatively addressed, having SYS2 and QCM occupy the same register provides sufficient coverage.

R14 and R15 each cover a page of floating point, integer, half, and byte constants, and are also named NUM1 and NUM2. The first several words of NUM1 are the code sequence used to bind global variables. The command, "BALR LINK,BIND", where BIND is another name for R14, is used to invoke this sequence. The first several words of NUM2 are the code sequence that unbinds global variables. The command, "BALR

LINK,UNBIND", where UNBIND is another name for R15, is used to invoke this sequence. Both binding and unbinding sequences are normally invoked through the use of assembly pseudo instructions.

The registers R12, R13, R14, and R15 have yet another major use in the system. That is, they are used as base - index pairs so that namea space can be directly accessed by RX format instructions. (Only RX format instructions can use both a base and an index register in computing the effective address.) Advantage is taken of the symmetry in usage of the base - index pair. Using the four registers, addresses 40000X through 48FFFX (the 9 full pages of namea) can be addressed. The page - pair correspondences are:

```
40000X-40FFFX  delta(R12,R12)
41000X-41FFFX  delta(R12,R13)
42000X-41FFFX  delta(R13,R13)
43000X-43FFFX  delta(R12,R14)
44000X-44FFFX  delta(R13,R14)
45000X-45FFFX  delta(R13,R15)
46000X-46FFFX  delta(R14,R14)
47000X-47FFFX  delta(R14,R15)
48000X-48FFFX  delta(R15,R15)
```

where 0≤delta<4096. The particular placement or R12 - R15 has been selected to maximize the number of contiguous pages that can be spanned using four registers, two at a time. This selection is a solution to an equivalent "postage stamp" problem.[1] The assembler converts global names used as

---

[1] Lunnon, W.F., "A Postage Stamp Problem", 1969.
The postage stamp problem consists of choosing, for a given n and m, a set of positive integers (stamp prices) such that
(a) sums of m or fewer of these integers can realize the numbers 1,2,3...N-1 (postage due)
(b) the value of N in (a) above is as large as possible.
The IMB 360/370 register allocation problem has m=2 and n

addresses into the proper delta and register pair combination.

## Function Linkage

Functions, processors, macros, and generators are normally called by the CALL, START, etc., pseudo instructions. This section describes the actual code sequences used to call and return from functions. There is no essential difference in the calling and return sequences from functions, processors, macros, and instructions.

When a function is called, the address of its named word is loaded into R7. Control is then transferred to the linkage sequence at the beginning of SYS1. For example, to call the function FCN$USER, the CALL pseudo instruction provides

```
LA      R7,FCN$USER;
BALR    LINK,FNLK;
-pdp bump-
-pdn bump-
```

where -pdp bump- is the distance in bytes between the return address from the calling function (on the pointer stack) and the new return address to the calling function and -pdn bump- is the distance in bytes between the top of the numeric stack upon entry to the calling function and the present top of the numeric stack. Both -pdp bump- and -pdn bump- are half word fields. (Fortunately, the pseudo instructions generate them.) Assume that FOO is a variable

---

the number of available general registers. For m=2, it is easy to show that $n^2/4+o(n)<n^2/2+o(n)$.

with a FUNC value; then the invocation sequence to the value of FOO is:

```
L       R7,FOO;
BALR    LINK,FNLK;
-pdp bump-
-pdn bump-
```

The following code is the linkage sequence at the beginning of SYS1 that is activated by the above calls.

```
AH      PDP,0(LINK);      %'Increment stack pointers'
AH      PDP,2(LINK);
SR      LINK,CB;          %'Compute return address'
ST      LINK,400X(PDP);   %' relative to CB'
L       CB,SYS(IFD);      %'System word IFD always'
ST      CB,404X(PDP);     %' contains function that'
ST      R7,SYS(IFD);      %' is currently active'
L       CB,0(R7);         %'The new code base'
B       6(CB);            %'Enter just past header'
```

Note, only registers CB, LINK, R7, PDP, and PDN are modified by the calling sequence. This is a guaranteed feature. The return sequence from a function is any branch to the beginning of SYS2. For instance, the command "BR FNRT". The code sequence at the beginning of SYS2 is:

```
L       CB,404X(PDP);     %'Restore system word IFD'
ST      CB,SYS(IFD);
ST      ZERO,404X(PDP);   %'Double word pop'
L       CB,0(CB);         %'Restore CBs from namea'
L       R7,404X(PDP);     %'Get rest of return address'
SH      PDP,0(CB,R7);     %'Unbump stacks'
SH      PDN,2(CB,R7);
B       4(CB,R7);         %'Re-enter function'
```

Beside the process control primitives, the only things that increment or decrement the stack pointers are the above AH and SH commands. The total overhead for a call and return including the "LA R7,fcn" to the instruction following the -pdn bump-, e.g., the time to call and return from a function that does nothing, is 37.928 microseconds. If and when the system allows binary programs that are longer than

4096 bytes, the command "LA CB2,4092(CB)" will be inserted as the next to last instruction in both the call and return sequences. This increases overhead time to 40.832 microseconds. (All timings quoted in this section are for the IBM 370, Model 145.)

The following describes the argument passing conventions. All except the last argument (indef count if present) are passed on the appropriate stack. The last argument is passed in F0 if a float and in R5 if anything else. First, the indefs are placed on the appropriate stack in their order of occurrence; then the other arguments (except the last or indef count) are placed on their appropriate stack in their order of occurrence. Given the declarations,

```
FUNCTION A(INT,FLOAT,NODE)
FUNCTION B(GEN,GEN,FLOAT)
FUNCTION C(NODE,INT,GEN INDEF)
```

then some examples of calling sequences are:

```
for A(I,F,N)
     L        R5,I;
     ST       R5,PUSHN.;
     L        F0,F;
     ST       F0,PUSHN.;
     L        R5,N;
     CALL     A;

for B(G1,G1,F)
     L        R5,G1;
     ST       R5,PUSHP.;
     L        R5,G2;
     ST       R5,PUSHP.;
     L        F0,F;
     CALL     B;

for C(N,I,G1,G2)
     L        R5,G1;
     ST       R5,PUSHP.;
     L        R5,G2;
     ST       R5,PUSHP.;
     L        R5,N;
     ST       R5,PUSHP.;
     L        R5,I;
```

```
ST      R5,PUSHN.;
LA      R5,2;           %'Indef count'
CALL C;
```

The only guarantee of register values when calling A, B, and C is R5 in A and C and F0 in B. That is, the compiler or programmer may use other, almost equivalent sequences that leave values on the stacks in the right order and the last argument in the proper register.

When processors are called (started), the registers R1, R2, R3, and R4 are loaded with the address to stuff the new handle, the abort link, the context link, and the activator link, respectively. Macros and generators are called in the same way as a FUNC with one general argument.

## Global Binding Mechanism

The global variable binding and unbinding mechanism is invoked automatically or semiautomatically by assembler pseudo instructions. The binding mechanism is activated by transferring to the code sequence at the beginning of NUM1 using

```
BALR    LINK,BIND;
-# pointer variables-
-# numeric variables-
-pointer binding locations-
-numeric binding locations-
```

-# pointer variables- is the number of global variables with pointer values that are to be bound, and -# numeric variables- is the number of global variables with numeric values that are to be bound. Each field is a half word.

For each pointer variable to be bound, there is a corresponding -pointer binding locations- that consists of two half word fields: the first half word is the location relative to PDP to make the value save (also contains the new value), and the second half word is the location of the variable's shallow binding (namea) cell relative to the beginning of namea space. For each variable with a numeric value, there is a corresponding -numeric binding locations- that consists of three half word fields: the first half word is the stack location, relative to PDP, where the old value is to be saved. The second half word is the location of the variable's shallow binding cell relative to the beginning of namea space. The third half word is the stack location of the new value relative to PDN. The binding sequence at the beginning of NUM1 is

```
   LH      R1,0(LINK);
   LH      R0,2(LINK);
   LTR     R1,R1
   BZ      X;
L;LH       R2,44(LINK);
   LH      R3,6(LINK);
   A       R3,SYS(NAME);
   ST      R3,4(PDP,R2);
   LE      F4,0(PDP,R2);
   LE      F6,0(R3);
   STE     F6,0(PDP,R2);
   STE     F4(0,R3);
   LA      LINK,4(LINK);
   BCT     R1,LABEL(L);
X;LTR      R0,R0;
   BZ      4(LINK);
Y;LH       R1,8(LINK);
   LH      R2,4(LINK);
   LH      R3,6(LINK);
   A       R3,SYS(NAME);
   ST      R3,4(PDP,R2);
   LE      F4,0(PDN,R1);
   LE      F6,0(R3);
   STE     F6,0(PDP,R2);
   STE     F4,0(R3);
   LA      LINK,6(LINK);
```

```
     BCT     R0,LABEL(Y);
   Z;B       4(LINK);
```

Note that of the programmer usable registers, only R4, R5, F0, and F2 are not clobbered.

The in-line sequence to unbind a set of global variables is

```
     BALR    LINK,UNBIND;
     -bind location--
```

where -bind location- is the location of the in-line bind sequence (shown above) relative to CB. It is a half word field. The code sequence at the beginning of NUM2 is

```
     LH      R3,0(LINK);
     AR      R3,CB;
     LH      R1,0(R3);
     LH      R0,2(R3);
     LTR     R1,R1;
     BZ      X;
   L;LH      R2,4(R3);
     ST      ZERO,4(PDP,R2);
     LE      F6,0(PDP,R2);
     LH      R2,6(R3);
     A       R2,SYS(NAME);
     STE     F6,0(R2);
     LA      R3,4(R3);
     BCT     R1,LABEL(L);
   X;LTR     R0,R0;
     BZ      2(LINK);
   Y;LH      R2,4(R3);
     ST      ZERO,4(PDP,R2);
     LE      F6,0(PDP,R2);
     LH      R2,6(R3);
     A       AR,SYS(NAME);
     STE     F6,0(R3);
     LA      R3,6(R3);
     BCT     R0,LABEL(Y);
     B       2(LINK);
```

Of the programmer usable registers, only R4, R5, F0, F2, and F4 are not clobbered. The total run time to bind a set of p pointer variables and n numeric variables (all global) in

microseconds is

26.396+p*35.131+n*37.426+3.5*[p=0]-.084*[n=0]]

If the programmer wants to improve the execution time at the expense of storage, then the following technique can be used to bind and unbind global variables. Assume that l is the name of the stack location on PDP that is to hold the save value, V$VAR is the name of the global variable to be bound, and the new value is in R5. The binding sequence is

```
LA      R1,V$VAR;
L       R4,0(R1);
ST      R5,0(R1);
ST      R4,l;
ST      R1,l+4;
```

The unbinding sequence is

```
ST      ZERO,l+4;
L       R5,l;
ST      R5,V$VAR;
```

The timing of this sequence is 14.247*v, where v is the total number of variables. The time to load the new value into R5 is included. Also, the operand "l+4" would be generated using SECOND(l).

# APPENDICES

# APPENDIX I

## Summary of IBM 370 Instruction Formats

The following table summarizes the IBM 370 instruction repertoire sorted by format. For more information, see the "little yellow card"[1] or the manual, IBM System/370 Principles of Operation.[2] In the table, the following abbreviations are used:

| | |
|---|---|
| r1, r2, r3 | -- register number (4 bits) |
| b1, b2 | -- base register (4 bits) |
| x2 | -- index register (4 bits) |
| m1, m3 | -- mask (4 bits) |
| d1, d2 | -- displacement value (12 bits) |
| l1, l2 | -- field length (4 or 8 bits) |
| i, i1, i2, i3 | -- number (4, 8, or 12 bits) |
| op | -- instruction mnemonic |

In the abbreviations, the number specifies to which operand the expression applies.

---

[1] IBM System/370 Reference Summary, GX20-1850-n.
[2] GA22-7000-n

| CLASS | FORMAT | INSTRUCTION MNEMONICS |
|---|---|---|
| RR | op r1,r2 | AR ALR NR BALR BCTR CR CLR CLCL DR XR ISK LR LTR LCR LNR LPR MVCL MR OR SSK SR SLR AXR ADR AER AWR AUR CDR CER DDR DER HDR HER LTDR LTER LCDR LCER LDR LNDR LNER LPDR LPER LRDR LRER LER MXR MDR MXDR MER SXR SDR SER SWR SUR |
| RR | op r1 | SPM |
| RR | op i | SVC |
| RR | op m1,r1 | BCR |
| RX | op r1,d2(x2,b2) | A AH AL N BAL BCT C CH CL CVB CVD D X EX IC L LA LH LRA M MH O ST STC STH S SH SL AD AE AW AU CD CE DD DE LD LE MD MXD ME STD STE SD SE SW SU |
| RX | op m1,d2(x2,b2) | BC |
| RS | op r1,r3,d2(b2) | BXH BXLE LCTL LM STCTL STM |
| RS | op r1,d2(b2) | SLDA SLDL SLA SLL SRDA SRDL SRA SRL |
| RS | op r1,m3,d2(b2) | ICM CLM STCM |
| SI | op d1(b1),i2 | NI CLI XI MC MVI OI RDD STNSM STOSM TM WRD |
| SS | op d1(l1,b1),d2(l2,b2) | AP CP DP MVO MP PACK SP UNPK ZAP |
| SS | op d1(l1,b1),d2(b2) | NC CLC ED EDMK XC MVC MVN MVZ OC TR TRT |
| SS | op d1(l1,b1),d2(b2),i3 | SRP |
| S | op d2(b2) | HIO HDV LPSW RRB SCK SCKC SPT SSM SIO SIOF STIDC STCK STCKC STIDP STPT TS TCH TIO |
| S | op r1,r2,i3 | DIAG |
| S | op | PTLB |

# APPENDIX II

## Summary of CAP Operand Formats

There are six kinds of operand fields that may be used in CAP instructions. For each kind, there are several operand formats that may be used. The basic form and alternative formats for each operand kind are:

| Kind | Basic Form | Operand Formats |
|---|---|---|
| register | ri | rid \| expr |
| mask | mi | expr |
| numeric | ii | expr |
| full address | di(xi,bi) | name \| label \| literal \| stackop \| sysop \| adr |
| half address | di(bi) | name \| label \| literal \| stackop \| sysop \| adr |
| length address | di(li,bi) | name \| label \| literal \| stackop \| sysop \| ladr |

The operand formats are syntactically different in SL and IL. Their definitions follow:

*SL*

| Operand Format | Definition |
|---|---|
| rid | identifier \| synonym-name |
| name | identifier \| global-name \| synonym-name |
| labelop | LABEL({identifier\|integer}[,register]) |
| sysop | SYS(identifier[,expr[,register]]) |
| literal | "external-data \| HQUOTE(external-data) \| TYPE(type-ref) \| {INT\|FLOAT\|HALF\|BYTE}(expr) \| MULTINT(integer,${, expr}) |
| stackop | PUSHP. \| PUSHN. \| POPP. \| POPN. \| TOPP.[(expr[,register])] \| TOPN.[(expr[,register])] \| SECOND(identifier) \| RET. |
| adr | expr[(register[,register])] |
| ladr | expr(expr,register) \| IMPLEN(expr,half-address) |
| expr | CAPop ${CAPoperator CAPop} |
| CAPoperator | + \| - \| * \| / \| // \| && \| \|\| \| BXOR |
| CAPop | number \| synonym-name \| offset \| length \| {+\|-\|INV}CAPop \| (CAPop) |
| offset | OFFSET item |
| length | LENGTH item |
| item | {identifier\|global-name}${_ identifier} |

*IL*

| Operand Format | Definition |
|---|---|
| rid | identifier \| synonym-name |
| name | identifier \| global-name \| synonym-name |
| label | (LABEL {identifier\|integer} [register]) |
| sysop | (SYS identifier [expr [register]]) |
| literal | ({QUOTE\|HQUOTE} external-data) \| (TYPE type-ref) ({INT\|FLOAT\|HALF\|BYTE} expr) \| (MULTINT integer $expr) |
| stackop | PUSHP. \| PUSHN. \| POPP. \| POPN. \| (TOPP. [expr [register]]) \| (TOPN. [expr [register]]) \| (SECOND identifier) \| RET. |
| adr | expr \| (expr [register [register]]) |
| ladr | (expr expr register) \| (IMPLEN exp half-address) |
| expr | number \| synonym-name \| offset \| length \| ({MINUS\|INV} expr) \| ({PLUS\|TIMES\|BAND\|BOR\|BXOR} $expr) \| ({DIFFER\|QUO\|IQUO} expr expr) |
| offset | (OFFSET item) |
| length | (LENGTH item) |
| item | {identifier\|global-name} $identifier |

# APPENDIX III

## CAP Pseudo Instructions

The following summarizes the format of the CAP pseudo instructions in SL:

| PSEUDO INSTRUCTION | FORMAT |
|---|---|
| CAP block | BEGIN ${instruction;}<br>BIND $*,*<block-var-dec>;<br>$*;*instruction<br>END |
| branches | {BONT\|BONF} register,label \|<br>special-branch label \|<br>special-branch-r register \|<br>GO {label\|RET.} |
| special-branch | B \| BH \| BL \| BE \| BO \| BP \| BM \| BNE \|<br>BNO \| BNH \| BNL \| BNP \| BNM \| BNZ \| BZ |
| special-branch-r | BR \| BHR \| BLR \| BER \| BNHR \| BNLR \|<br>BNER \| BOR \| BPR \| BMR \| BNPR \| BNMR \|<br>BNZR \| BNOR |
| label | identifier \| full-address |
| stackint | {PUSHP.\|PUSHN.\|POPP.\|POPN.} expr; |
| callers | CALL(f-name $*;*inst) \|<br>FUNCALL ($*;*inst) \|<br>START(p-name $*;*inst) \|<br>STARTPROC ($*;*inst) \|<br>SYSCALL(s-name $*,*inst) |
| synonym | identifier := expr |
| sysint | SYS {SYS1\|SYS2\|NUM1\|NUM2} |
| try | TRY{USER\|SYS\|ALL} *($*;*{inst\|endtry}) |
| endtry | ENDTRY label |
| space test | SPACE register ${,s-name slabel}<br>[,slabel] |
| slabel | label \| NIL \| R register \| RET. |

The formats in IL are:

| PSEUDO INSTRUCTION | FORMAT |
|---|---|
| CAP block | (BEGIN $instruction<br>(BIND $<block-var-dec>)<br>$instruction) |
| branches | ({BONT\|BONF} register labelop) \|<br>(special-branch label) \|<br>(special-branch-r register) \| |
| (GO {label\|RET.}) | |
| special-branch | B \| BH \| BL \| BE \| BO \| BP \| BM \| BNE \|<br>BNO \| BNH \| BNL \| BNP \| BNM \| BNZ \| BZ |
| special-branch-r | BR \| BHR \| BLR \| BER \| BNHR \| BNLR \|<br>BNER \| BOR \| BPR \| BMR \| BNPR \| BNMR \|<br>BNZR \| BNOR |
| label | identifier \| full-address |
| stackint | ({PUSHP.\|PUSHN.\|POPP.\|POPN.} expr) |
| callers | (CALL f-name $inst) \|<br>(FUNCALL $inst) \|<br>(START p-name $inst) \|<br>(STARTPROC $inst) \|<br>(SYSCALL s-name $inst) |
| synonym | (SET identifier expr) |
| sysint | (SYS {SYS1\|SYS2\|NUM1\|NUM2}) |
| try | (TRY {USER\|SYS\|ALL} #($${inst\|endtry})) |
| endtry | (ENDTRY label) |
| space test | (SPACE register $(s-name slabel)<br>[slabel]) |
| slabel | label \| NIL \| B register \| RET. |

# APPENDIX IV

## Key Words and their Alternatives

The following are keywords and characters having syntactic significance in CRISP. Though it may work correctly to use them as variable or function names, extreme caution should be observed. When more than one word is listed on one line, the words are equivalent in meaning.

```
(
)
{
}
[
]
<
>
≤
≥
=
%
"
$
*
+
-
/
_
:
:
.
&
|
¬
#
@
ABS
ALL
AND
ANY
APPEND
APPENDR
ARRAY
ADDRIBUTE / ATTR
BEGIN
```

```
BIND
BOOLEAN / BOCL
BXOR
BY
BYTE
CAP
CHARACTER / CHAR
CHEAT
COMPLEX
COUNT
DAPPEND
DAPPENDR
DECLARE / DEC / DCL
DEFAULT
DO
DRIVE
EIGHTH
ELSE
END
ENDF
FIFTH
FINALLY
FIRST
FLAT
FLOAT / FLT
FOR
FOURTH
FUNC
FUNCTION
GENERAL / GEN
GENERATOR
GLOBAL
GOTO / GO
HALF
HANDLE
IDENTIFIER / ID
IF
IL
IMPLICIT
IN
INDEF
INITIALLY
INTEGER / INT
INTER
IS
LEAVE
LIKE
LIST
LISTR
LOCAL
MACRO
MYSELF
NABS
NAME
NODE1
NODE2
NODE3
NODE4
```

NODE5
NODE6
NODE7
NODE8
NOVALUE
NTUPLE
NUMBER
OLD
ON
OR
ORIF
OWN
POPN.
POPP.
PROC
PROCESSOR
PRODUCT
PROP
PUSHN.
PUSHP.
RESET
RETURN / RET
RET.
SECOND
SELECT
SELECTN
SELECTQ
SELECTT
SEVENTH
SIXTH
SL
STRING
SUM
SYNONYM / SYN
SYNX
THEN
THIRD
THRU
TO
TOPN.
TOPP.
TRANSFORM
TYPEP
UNION
UNLESS
UNTIL
VALUE
VARB
VARIABLE
WHEN
WHILE

# APPENDIX V

## Initial conditions

When CRISP is loaded, the follcwing initial conditions exist.

```
DEFAULT USER(USER,CRISP);
FORMAT SL;
IMPLICIT;
LOWSUB:=1;
FILETYPES:="(CRISP INDEX DATA);
OPEN("(ITERM TERMINAL R));
OPEN("(OTERM TERMINAL W));
RDS("ITERM);
PRS("OTERM);
PRECISIONP:=TRUE;
CHKFIELD:=TRUE;
ERRCHK:=TRUE;
SPACE:="$' '
LPAR:="$'('
RPAR:="$')'
LBRC:="$'{'
RBRC:="$'}'
DOLLAR:="$'$'
PERCENT:="$'%'
```

Standard data spaces are selected.

# APPENDIX VI

## System Limitations

The following are some limitations imposed by the CRISP system and the IBM 370 as opposed to the language.

id length < 256 chars

length of string representable with primes < 256 chars

$-2^{32} \leq$ integer value $< 2^{32}$

$-2^{20} \leq$ small int value $< 2^{20}$

$16^{-65} \leq$ magnitude of float $\leq (1-16^{-6})*16^{63}$

number of array dimensions < 256

extent of a single dimension $< 2^{15}-1$

I/O record length $< 2^{16}$ bytes

number of records in a file $< 2^{15}$

number of pointer args to function or processor < 128

number of numeric args to function or processor < 256

number of NAME space entries < 9216

maximum size of binary program < 4096 (maybe 8188) bytes

number of handles < 1024

pointer stack entries < 8192

number stack entries < 8192

process heap $< 2^{16}$ bytes

number of forms in a try < 256

# APPENDIX VII

## Static Page Allocation

The following summarizes the locations of statically allocated pages; that is, which pages belong to spaces that are statically allocated. Pages are numbered in hex.

| PAGE | USAGE | PAGE | USAGE | PAGE | USAGE |
|---|---|---|---|---|---|
| 20 | SYS1 | 40 | NAMEA | 60 | NAMEB |
| 21 | SYS2 | 41 | NAMEA | 61 | NAMEB |
| 22 | QCM | 42 | NAMEA | 62 | NAMEB |
| 23 | NUM1 | 43 | NAMEA | 63 | NAMEB |
| 24 | NUM2 | 44 | NAMEA | 64 | NAMEB |
| 25 | | 45 | NAMEA | 65 | NAMEB |
| 26 | | 46 | NAMEA | 66 | NAMEB |
| 27 | CHAR | 47 | NAMEA | 67 | NAMEB |
| 28 | PDP | 48 | NAMEA | 68 | NAMEB |
| 29 | PDP | 49 | NAMEB | 69 | NAMEB |
| 2A | PDP | 4A | NAMEB | 6A | NAMEB |
| 2B | PDP | 4B | NAMEB | 6B | NAMEB |
| 2C | PDP | 4C | NAMEB | 6C | NAMEB |
| 2D | PDP | 4D | NAMEB | 6D | HNDLE |
| 2E | PDP | 4E | NAMEB | 6E | HNDLE |
| 2F | PDP | 4F | NAMEB | 6F | HNDLE |
| 30 | PDP | 50 | NAMEB | 70 | HNDLE |
| 31 | PDP | 51 | NAMEB | 71 | PHEAP |
| 32 | PDP | 52 | NAMEB | 72 | PHEAP |
| 33 | PDP | 53 | NAMEB | 73 | PHEAP |
| 34 | PDP | 54 | NAMEB | 74 | PHEAP |
| 35 | PDP | 55 | NAMEB | 75 | PHEAP |
| 36 | PDP | 56 | NAMEB | 76 | PHEAP |
| 37 | PDP | 57 | NAMEB | 77 | PHEAP |
| 38 | PDN | 58 | NAMEB | 78 | PHEAP |
| 39 | PDN | 59 | NAMEB | 79 | PHEAP |
| 3A | PDN | 5A | NAMEB | 7A | PHEAP |
| 3B | PDN | 5B | NAMEB | 7B | PHEAP |
| 3C | PDN | 5C | NAMEB | 7C | PHEAP |
| 3D | PDN | 5D | NAMEB | 7D | PHEAP |
| 3E | PDN | 5E | NAMEB | 7E | PHEAP |
| 3F | PDN | 5F | NAMEB | 7F | PHEAP |
| | | | | 80 | PHEAP |

Where PHEAP is the process heap and HNDLE is handle space.

# APPENDIX VIII

## Spaces Summary

The following table summarizes the default management scheme for each space kind. The included information is the possible number of spaces of that kind, whether the space is static or selectable, the region size, the reclamation technique, whether there is a local mark table, and the number of regions in the space.

| SPACE | NUMS | STATIC | SELECT | RSIZE | MOVE | LMARK | NREG |
|---|---|---|---|---|---|---|---|
| NODE1 | M | N | Y | Q | F | N | M |
| NODE2 | M | N | Y | Q | F | N | M |
| NODE3 | M | N | Y | Q | F | N | M |
| NODE4 | M | N | Y | Q | F | N | M |
| NODE5 | M | N | Y | Q | F | N | M |
| NODE6 | M | N | Y | Q | F | N | M |
| NODE7 | M | N | Y | Q | F | N | M |
| NODE8 | M | N | Y | Q | F | N | M |
| ID | 1 | N | N | B | CC | N | 1 |
| CHAR | 1 | Y | N | 1 | - | N | 1 |
| INTEGER | M | N | Y | Q | F | Y | M |
| FLOAT | M | N | Y | Q | F | Y | M |
| COMPLEX | M | N | Y | Q | F | Y | M |
| ARRAY | M | N | Y | B | C | N | M |
| NTUPLE | M | N | Y | B | C | N | M |
| NAMEA | 1 | Y | N | 9 | EL | N | 1 |
| NAMEB | 1 | Y | N | 36 | EL | N | 1 |
| PDP | 1 | Y | N | 16 | - | N | 1 |
| PDN | 1 | Y | N | 8 | - | N | 1 |
| BPS | M | N | Y | E | C | N | M |
| HANDLE | 1 | Y | N | .4 | EL | N | 1 |
| HEAP | M | N | N | B | - | N | M |
| SMINT | 1 | Y | N | 512 | - | N | 1 |

M - multiple B - multi quantum regions Q - region=quantum
F - fold C - compact EL - erasure list CC - copy collect