A Language Processor and a Sample Language

Ronald Ayres

Computer Science Department
California Institute of Technology

2276:TR:78

A LANGUAGE PROCESSOR AND A SAMPLE LANGUAGE

by

Ronald Ayres

Technical Report 2276

June 12, 1978

Computer Science Department

California Institute of Technology

Pasadena, California 91125

Silicon Structures Project

sponsored by

Burroughs Corporation, Digital Equipment Corporation,

Hewlett-Packard Company, Honeywell Incorporated,

International Business Machines Corporation,

Intel Corporation, Xerox Corporation,

and the National Science Foundation

ABSTRACT

This thesis explores shared data in list structures and ambiguity in language processing. Tolerance of ambiguity is necessary to support clear and modular expression. Data sharing is necessary to support ambiguity efficiently. Data sharing is useful also in compiled programs to save memory and time.

Let us define some terms. A *rewrite grammar* is a set of replacement rules each of which specifies that a given phrase may be replaced by another given phrase. Each replacement rule expresses a local translation. A *parser* finds those sequences of replacements that bring a given text to a machine handleable form. Each such sequence represents a *meaning* or *interpretation* for the given text. Tolerance of *ambiguity* or multiple interpretations for a given text is necessary so that subsequent processing can place further constraints upon the input text.

This thesis presents a parser which efficiently handles general-rewrite grammars. To conserve computer time and memory, only essential differences among multiple interpretations are represented and processed. If several interpretations for a given text are valid, the parser yields a meaning which represents the ambiguity as locally as possible. Even an exponential number of distinct meanings may be represented in a polynomial amount of memory.

This thesis also presents a language processing system which supports semantic processing via independent rewrite grammars. Each grammar represents a distinct aspect of the language. A given sequence of grammars becomes a sequence of passes, or process steps. Each pass derives a meaning with respect to one grammar and uses that meaning to generate phrases which will be interpreted by the next pass. Although linguistic specification is usually done with context-free grammars, features of this parser which support general-rewrite grammars are essential for the integration of passes. Not only ambiguity, but also the locality of ambiguity is preserved from one pass to the next. It is necessary to preserve locality of ambiguity in order to avoid explosive computation arising from useless interaction among independent sets of interpretations.

I have implemented a general-purpose programming language called ICL with this system. The fact that ICL's datatypes are processed by a rewrite grammar makes it simple to implement both user-defined datatype coercions and functions known as polymorphic operators whose definitions depend on parameter datatypes. Datatype coercions and polymorphic operators reduce the amount of specification required in algorithms to such an extent that a user can often modify declarations and achieve optimizations and changes in concept without modifying his algorithmic specification.

ICL includes a simple and safe policy about pointers so that the user can ignore their existence completely if he wishes. ICL automatically maximizes data sharing and minimizes copying by adopting a

"copy on write" policy. This policy supports the illusion that each and every reference to a data structure generates a complete copy of that data structure. This same technique is used in the language processor itself to facilitate data sharing among multiple interpretations in ambiguous cases.

Table of Contents

INTRODUCTION

This thesis presents a programming language, ICL, and the language processor with which it was implemented. The design and implementation of ICL was facilitated by building a flexible language processor which readily admits the creation and modification of any computer language. ICL was created for two reasons. There was need for a sophisticated IC design language and there was need to see how well the language processor could support a large application language.

ICL was conceived by forming a collection of notations which would express a variety of independent concepts. The language emerged by integrating these various notations. The rules specifying how these notations could be integrated became the grammar for ICL. ICL is now defined by three independent grammars each of which imposes a different class of requirements.

The language processor supports linguistic specification in terms of general rewrite grammars. A rewrite grammar is a set of replacement rules each of which specifies that a given phrase may be replaced by another given phrase. Each replacement rule expresses a local translation. A meaning is derived from an input text by applying replacement rules upon the text in such a way as to bring the input text to a machine usable form.

The program which discovers the appropriate replacements is called a parser. The result of parsing is a record of the replacements performed, a tree structure known as a derivation. A derivation may be viewed as nested function calls, e.g., the text

1 * 2 + 3 * 4

may have the derivation

plus(*times*(1,2) , *times*(3,4)).

Viewed as nested function calls, a derivation can be *executed*. The execution of a derivation implements the intended meaning. Refer to the section ***Languages*** for a more complete description.

Ambiguity and Shared Data

This thesis is based on two ideas. One is tolerance of ambiguity and the other is automatic sharing of data in list structures. ***Ambiguity*** refers to the existence of multiple interpretations for a given expression. ***Data sharing*** refers to the representation of nearly identical structures where all those substructures which are common in the various structures are represented in memory only once. A common substructure is said to be *shared* by all structures which reference the substructure. Tolerance of ambiguity becomes practical when only essential ***differences*** among multiple interpretations are represented and processed. ***Similarities*** among multiple interpretations will be shared both in memory and in processing.

Tolerance of ambiguity supports two needs in processing. On one hand, the support of multiple interpretations allows programs to be picky. A program will be given a choice of interpretations for input and hence the program can choose those interpretations amenable to the program's needs. Making such choices reduces the number of interpretations, or the degree of ambiguity. On the other hand, a program may generate multiple interpretations for output when it finds

several valid ways to procede.

If ambiguity were to be avoided, a predictive policy would have to be adopted. For example, when a program could generate multiple interpretations, the program would instead have to predict ahead of time which interpretation will actually be utilized in subsequent processing. This is not always possible. To resolve the uncertainty in prediction, numerous systems employ backtracking so that when a prediction fails to come true, processing can be backed up to the point where the faulty prediction was made and another interpretation can be predicted in its place.

The language processor supports ambiguity by processing multiple interpretations in *parallel*. The major advantage of parallel processing is that all valid interpretations will be presented together at any point in processing. This means that similarities among the various interpretations can be known and hence the similarities can be represented and processed only once. In contrast, if only one interpretation is processed at a time, e.g., with backtracking, all interpretations will not be presented together and hence similarities among multiple interpretations are ignored. The cost of processing multiple interpretations one at a time can be exponential where a parallel implementation would incur only polynomial cost.

Parsers

A variety of parsers exist ranging from parsers tailormade for specific languages to general-purpose parsers which process large classes of grammars. The simplest kind of parser is the LR(*k*) parser presented by Knuth[1]. Such a parser avoids ambiguity by restricting the class of grammars to such an extent that it can be decided with certainty which replacement rule applies by looking ahead at most *k* characters. Another simple parsing technique known as recursive descent[2] utilizes backtracking exclusively to support uncertain decisions.

Earley's efficient context-free parser[3] handles any context-free grammar with a worst case performance of n^3 where n is the length of the text to be parsed. A context-free grammar is a rewrite grammar each of whose replacement rules substitutes a given phrase with a phrase of length one. Currently, Earley's is the most efficient parser which accepts all context-free grammars.

Thompson's REL parser[4] and Kay's Powerful Parser[5] each accepts general rewrite grammars. However, the REL parser has a worst case performance of infinity even for context-free grammars. I don't know if Kay's parser has an upper bound.

The parser presented in this thesis accepts general rewrite grammars. When given context-free grammars in particular, the parser has a polynomial upper bound as a function of the input text's length. If the context-free grammar is in Chompsky Normal Form, i.e., each replacement rule replaces a phrase of length at most two, then this

parser's upper bound is n^4. The disparity between this n^4 and Earley's n^3 comes about because Earley indexes into an array of length *n* where this parser walks a list structure of length *n*. The use of an array is cumbersome when dealing with general-rewrite grammars.

The section *An Efficient General Rewrite Parser* documents this parser.

A Language Processor

The language processor presented in this thesis supports multipass, or semantic processing via independent rewrite grammars. A given sequence of grammars becomes a sequence of passes, or process steps. Each pass derives a meaning with respect to one grammar and uses that derivation to generate phrases which will be interpreted by the grammar belonging to the next pass. The grammars in a multipass system represent the constraints and capabilities of distinct aspects of a given language. The first pass in a multipass system is usually referred to as the syntax pass and non-first passes are referred to as semantic passes. For example, ICL is implemented with three passes, a syntax pass, a datatype processing pass, and a pass which enforces proper use of data *sources* and data *sinks*. It is conceivable that a fourth pass could be added which would process the output from the third pass in terms of a *register transfer* language. Some replacement rules of the register transfer language could map certain sequences of instructions to other sequences and thereby offer alternate implementations. An optimal implementation could be chosen from these alternatives.

This multipass language processor necessitates a general rewrite parser because a derivation must be able to generate phrases of length greater than one. In analogy, where a *replacement rule* generates a phrase in place of another phrase, a *derivation* generates a phrase in place of itself. A derivation can generate a phrase by concatenating those phrases generated by its subderivations. Phrases generated either by replacement rules or by derivations may interact with surrounding phrases. Each generated phrase and each union of that phrase with surrounding phrases must be subject to processing via replacement rules.

Multipass language processing emerged originally in the days when computers had tiny memories. By running passes independently, each pass could use the whole computer memory. Communications from one pass to the next were made via a text string stored on disk. However, ambiguities which could not be resolved by one pass were not easily passed on to the next pass.

Ambiguity must be supported within and between passes so that each pass need not be overly specified and hence overly rigid. If a pass were not able to deliver its unresolved ambiguities to the next pass, each pass would have to resolve all ambiguities within the pass's limited domain. In general, this would require that each pass emulate subsequent passes so that the given pass can successfully predict which interpretation it should deliver. Because each pass represents a distinct aspect of the overall language, the requirement that a pass emulate subsequent passes forbids a truly independent specification for each independent aspect of the given language. The support of ambiguity provides the lubrication, so to speak, between the independent domains

of each pass.

For example, an ambiguity not resolvable by the first pass will be delivered to the second pass. If the ambiguity makes no distinction in the domain of the second pass, the second pass will automatically process each of the alternate interpretations and deliver them on to the third pass. The ambiguity may be resolved by the earliest pass within whose domain the ambiguity makes a distinction.

In general, each pass will not only resolve ambiguities but also generate new ambiguities. For example, in FORTRAN, the number 259 is unambiguous syntactically, but when FORTRAN considers datatypes, the number 259 becomes ambiguous because 259 must be considered as either an integer or a real number. Of course, the ambiguity is resolved when surrounding context is taken under consideration, e.g., 259 is specifically assigned to an integer variable.

To support ambiguities between passes practically, the individuality, or locality of these ambiguities must be preserved. It is not satisfactory, for example, to have each pass yield a set of unambiguous derivations, each of which will be processed independently by the next pass. Because each derivation will typically have much in common with the other derivations, processing each derivation independently will result in duplicate processing for similarities among the various derivations. Ignorance of similarities among multiple derivations can turn a polynomial cost into an exponential cost.

Both this parser and Earley's parser have the wonderful property that ambiguities which cease to provide distinction for the parsing process disappear from the parsing process. These ambiguities reappear embedded within the resulting *ambiguous* derivation. An ambiguous derivation is a derivation which may contain instances of a new kind of node called an *OR*-node. A single ambiguous derivation represents many distinct unambiguous derivations.

For example, the ambiguous derivation

OR (f(a) , f(b))

represents the meaning

either f(a) or f(b).

The ambiguous derivation

f (OR(a,b))

represents the meaning

f(either a or b)

and it is in fact equivalent to the former derivation. This latter derivation is said to be more *factored* than the former derivation because the *OR*-node is nested deeper within the latter derivation. That is, just as

f*(a+b) is more factored than f*a + f*b,

f(OR(a,b)) is more factored than OR(f(a) , f(b)).

In each example, the most factored expression shares the most data. *f* is written only once in the factored expressions whereas *f* is written twice in the unfactored expressions. For another example,

g(*OR*(a,b) , *OR*(c,d)) is more factored than

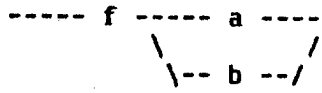
OR(g(a,*OR*(c,d)) , g(b,*OR*(c,d))) and than

OR(*OR*(g(a,c),g(a,d)) , *OR*(g(b,c),g(b,d))).

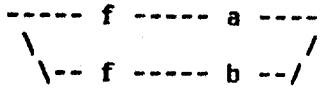
A maximally factored ambiguous derivation can represent as many as an exponential number of distinct derivations in only a polynomial amount of memory. Both Earley's and this parser yield maximally factored ambiguous derivations.

This language processor takes full advantage of ambiguous derivations. In support of multipass processing, an ambiguous derivation is used to generate an *ambiguous* phrase. An ambiguous phrase is a datastructure which represents a set of alternative phrases by sharing as many common subphrases as possible.

An ambiguous phrase is maximally factored in the sense that



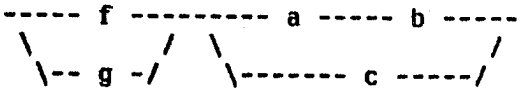
is more factored than



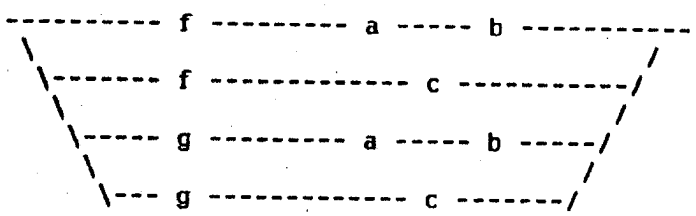
Each of these ambiguous phrases represents the phrases

f a and *f b* .

A more factored ambiguous phrase shares more data, e.g., the *f* is shared in the more factored ambiguous phrase. The ambiguous phrase



is more factored than the unfactored



With the unfactored ambiguous phrase given above, f is represented twice. An unfactored ambiguous phrase wastes not only memory space, but also processing time: With this unfactored phrase, those processes, e.g., replacement rules, which depend on f *without* reference to f's surrounding context will be duplicated simply because f is represented twice. With the factored ambiguous phrase, f is represented only once and hence those processes which depend on f without reference to f's surrounding context will execute only once. Maximally factored ambiguous derivations or phrases are said to maintain *locality of ambiguity*.

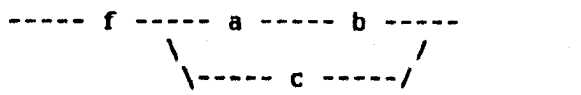
The parser presented in this thesis in fact deals exclusively with ambiguous phrases. A replacement rule affects an ambiguous phrase by placing the rule's generated phrase onto the ambiguous phrase as an alternate phrase. For example, applying the replacement rule

c replaces *a b*

upon the phrase

----- f ----- a ----- b -----

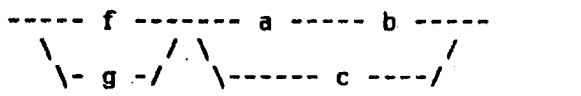
yields the ambiguous phrase



Applying the replacement rule

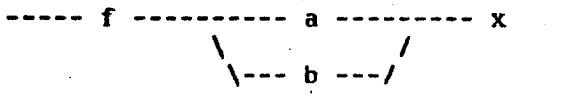
g replaces *f*

upon this result, we get the ambiguous phrase



Because replacement rules make local replacements, the parser naturally preserves the locality of ambiguity within ambiguous phrases.

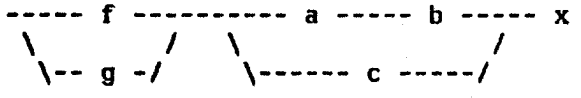
It is interesting to note that the *structure* of an ambiguous derivation generalizes the *structure* of an ambiguous phrase. The structure of an ambiguous derivation is in fact identical to the structure of an ambiguous phrase when each function in the derivation is unary, i.e., each function takes at most one parameter. The equivalence can be seen by viewing the phrase *a b c x* as the derivation *a(b(c(x)))*. For example, the ambiguous phrase



corresponds *in structure* to the ambiguous derivation

f(OR(a(x) , b(x))).

The ambiguous phrase



corresponds *in structure* to the ambiguous derivation

OR(f(*u*),g(*u*)) where *u* = OR(a(b(x)) , c(x)).

The Sample Language, ICL

ICL, the sample language implemented with the language processor, includes two major features taken from the language processor. ICL maximizes data sharing so that nearly identical datastructures indeed share common substructures. In addition, ICL utilizes the parser to process datatypes. All datatype relationships are represented in a *datatype* grammar. Refer to the section *ICL Overview* for a closer look at how these features manifest themselves in the language.

ICL was designed to support the creation of integrated circuits. Because a given integrated circuit is defined by a given set of masks, ICL includes special features for processing two-dimensional geometry.

Because the specification of IC layouts and electrical or functional properties is a relatively new endeavor, I chose to make ICL a flexible, general-purpose programming language. I envision two distinct user groups. The first group is akin to language designers; this group defines both the internal representation for IC's and the notation with which IC's are specified and manipulated. The other group defines and edits specific IC's with the system provided by the first group. Each group interacts with the other; the language group continually modifies its system by incorporating common needs found by the IC designers. In this way, a convenient IC design system evolves

and avoids obsolescence.

Paramount to this duality is the need for upward compatability. IC specification should always be successfully interpreted by the system even if the internal representation for IC's changes dramatically. Not only top-level IC specs, but also IC specs made from within existing functions should be upward compatible. In each case, the existing IC specification should be mapped optimally into the new representation.

ICL includes two essential features which readily support changes in representation. These features are known as *datatype coercions* and *polymorphic operators*. A datatype coercion is a declaration which specifies that any instance of a given datatype can be viewed as an instance of another given datatype via a given transformation. A datatype coercion differs from a function because a datatype coercion may be invoked without an explicit call. A function can be invoked only by explicitly writing the function's name.

ICL applies datatype coercions at appropriate places in a user's specification to maintain the integrity of his specification. Thus, upon changes to representation, i.e., changes to the notion of integrity, ICL will automatically apply coercions at different places if necessary. If the user had gone to the trouble of specifying all coercions via function calls, as is necessary in other programming languages, his specification would be overly rigid and less amenable to changes in representations. Upon changes in representation, the user would have to edit his specification by removing certain function calls and adding other function calls. With coercions, this is done automatically. Because ICL guarantees to minimize the number of

coercions it employs, the integrity of a user's specification will be maintained as concisely as possible.

A polymorphic operator is a function whose definition depends on the datatypes of its parameters. For example, the function name DISPLACE can have several definitions. One definition will support the displacement of a *point* by a *point*. Another definition will support the displacement of a whole *IC-mask* by a *point*. When the user specifies a call to DISPLACE, ICL will choose that definition which maintains datatype consistency. The use of polymorphic operators, like datatype coercions, reduces the necessary specification and hence the rigidity of specification. A change in representation which affects the datatypes in a call to DISPLACE can be tolerated if another definition of DISPLACE can accomodate the new datatypes.

Program integrity is preserved via the parser during ICL's second pass, when the datatype grammar is active. The parser's tolerance of ambiguity lends itself naturally to the task of discovering which coercions to apply where and what definition to use among those definitions which make up a given polymorphic operator.

What Follows

The section *Languages* presents rewrite grammars and various techniques by which rewrite grammars can be extended to encompass more linguistic specification. The language processor itself is made up of two components, a parser and a semantic evaluator. One section documents the parser. Another section documents the semantic evaluator

and its role in the language processor. The section *ICL Overview* documents the goals in designing ICL and the results of each goal.

The first appendix gives a sketch of how a language is specified to the language processor. Linguistic specification is done in the PDP-10's assembler language with the help of macros. The final appendix, the *ICL Reference Manual* formally documents ICL.

LANGUAGES

A language is a set of conventions by which a string of characters can be mapped into some corresponding meaning. On the computer, the most general form of meaning is some action which modifies either the state of memory or the state of some output devices. This section introduces rewrite grammars, terminology, and useful conventions for implementing meaning. The following sections document programs which implement ideas presented in this section.

The term *part-of-speech* will refer to the atomic elements of our space of discourse. We include all ASCII characters as parts-of-speech. All parts-of-speech excluding ASCII characters are called non-terminals and can be thought of as syntactic classes or as abstract characters. A non-terminal will be denoted by a name enclosed in angle brackets, e.g.,

<VERB> .

The term *phrase* or *string* refers to any sequence of parts-of-speech. Phrases made up solely of ASCII characters are called terminal phrases.

A *production*, or *rule*, is a pair of phrases, written as

phrase ::= *phrase* .

The phrases will be refered to as the *lefthand phrase* and the *righthand phrase* respectively.

A *grammar* is a set of productions along with a chosen part-of-speech called the *root* part-of-speech. Given a grammar, we will say that a given string is a *rewrite* of another string precisely when the given string can be obtained from the other string by a single rewrite operation:

1) Locate an occurence of some production's righthand phrase within the given string.

2) Erase that occurrence of the righthand phrase and write in its place a copy of the production's lefthand phrase.

We will say that a given string is *derivable* from another string if the given string can be obtained from the other string by a sequence of rewrites. A *derivation* is a record of the rewrite operations employed in deriving one string from another.

The *language* accepted by a grammar is the set of terminal strings from which the root phrase is derivable. The root phrase is the phrase of length one consisting of the grammar's root part-of-speech. In performing rewrite operations upon a given text, the goal is to come up with the root phrase.

For example, the set of productions

```
<DIGIT>  ::=  0
<DIGIT>  ::=  1

   . . .

<DIGIT>  ::=  9
```

state that the part-of-speech <DIGIT> can be derived from each of the characters 0, 1, ..., 9. The productions

<NUMBER> ::= <DIGIT>

<NUMBER> ::= <NUMBER> <DIGIT>

state that <NUMBER> can be derived from a single <DIGIT> or from a <NUMBER> followed by a <DIGIT>. Thus, <NUMBER> is any non-null string of digits. The grammar consisting of these productions along with <NUMBER> as the root part-of-speech defines the language consisting of all non-null strings of digits.

Meaning

The preceding description shows how a grammar can be used to specify the legal strings of a language but it fails to mention how to associate a meaning with a given string in the language. We can incorporate meaning by associating a meaning with each element in a string. That is, an element of a string will consist of not only a part-of-speech but a part-of-speech and a meaning. We can associate with each production a meaning transformation. That is, when a rewrite operation is performed, we let the chosen production define a meaning for each element it writes into the string. These meanings will be functions of the meanings associated with each of the erased elements. For example, the production

<EXPR> ::= <EXPR> + <EXPR>

can have the transformation which yields the sum of the meanings associated with each of the righthand <EXPR>s. The meaning of a string in a language will be the meaning which is associated with the root phrase derived from the given string.

We can express productions which include meaning transformations in the following concise notation:

<EXPR: sum(a,b)> ::= <EXPR:a> + <EXPR:b>

The parts-of-speech appearing in the righthand phrase include the specification of variables and the parts-of-speech appearing on the lefthand phrase each includes the specification of a meaning which is a function of the variables named in the righthand phrase. One can see how a transformation is carried out. When this production is employed in a rewrite operation, the variables *a* and *b* are set to the meanings associated with the two <EXPR> elements which are being erased. The value *sum(a,b)* is computed and associated with the new <EXPR> replacing the erased elements.

A meaning transformation associated with a production whose lefthand phrase has length greater than one defines a separate meaning for each part-of-speech in the lefthand phrase. For example, the production

<A: f(c,d)> <B: g(c,d)> ::= <C:c> <D:d>

specifies that the meaning under the <A> is *f(c,d)* and that the meaning under the <B> is *g(c,d)*. A meaning transformation associates a meaning with *each part-of-speech* in the lefthand phrase and *not* with the phrase as a whole.

The <NUMBER> grammar can be written with meaning transformations as follows:

```
<DIGIT:0>  ::=  0
<DIGIT:1>  ::=  1

   . . .

<DIGIT:9>  ::=  9

<NUMBER:a>  ::=  <DIGIT:a>

<NUMBER: 10*a+b >   ::=  <NUMBER:a>  <DIGIT:b>
```

The meaning associated with a <DIGIT> or a <NUMBER> is an integer. Looking at the <DIGIT> rules, note that the digit appearing on the righthand side is a character whereas the meaning associated with the <DIGIT> is an integer. For example, the rule

```
<DIGIT:1>  ::=  1
```

states that the character "1" has the integer 1 as its meaning when the character "1" is viewed as a <DIGIT>. The first <NUMBER> rule states that when a <DIGIT> is viewed as a <NUMBER>, the meaning for the <NUMBER> is the same as the meaning associated with the <DIGIT>. The final rule states that when a <NUMBER> followed by a <DIGIT> is viewed as a <NUMBER>, the meaning for the resulting <NUMBER> is ten times the meaning of the given <NUMBER> plus the meaning of the <DIGIT>.

Many grammars can be written which accept a given language. However, some grammars may be more suitable than others for defining meaning transformations. For example, consider the grammar given above which accepts the language consisting of strings of digits. The rule

```
<NUMBER>  ::=  <NUMBER> <DIGIT>
```

could be replaced by the rule

<NUMBER> ::= <DIGIT> <NUMBER>

without changing the language accepted by the grammar. However, it is more difficult to write the meaning transformation for the latter rule than to do so for the former rule. Given a language to implement, the language implementor generally chooses that grammar whose meaning transformations are easiest to define.

The following is a grammar for a subset of arithmetic expressions:

<ATOM: a> ::= <NUMBER:a>

<TERM: a> ::= <ATOM:a>

<TERM: times(a,b)> ::= <TERM:a> * <TERM:b>

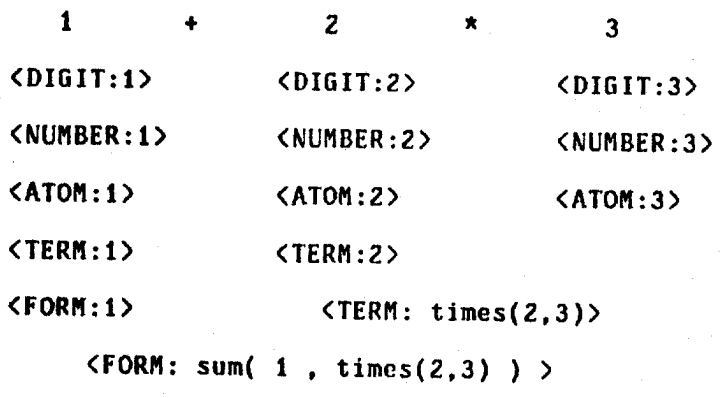
<FORM: a> ::= <TERM:a>

<FORM: sum(a,b)> ::= <FORM:a> + <TERM:b>

<ATOM: a> ::= (<FORM:a>)

This grammar admits expressions built with numbers, +'s, *'s and parentheses. The part-of-speech <ATOM> admits only numbers and parenthesized expressions. <TERM> admits products of <ATOM>s and <FORM> admits sums of <TERM>s. The separation of *arithmetic expressions* into <ATOM>, <TERM>, and <FORM> implements the standard operator precedences: *'s are grouped before +'s.

<FORM> can be derived from the string 1+2*3 by



This diagram shows only those rewrite operations which participate in deriving the final <FORM>. The final <FORM> has the accumulated meaning

sum(1 , times(2,3)).

We have not specified what kind of data *sum* and *times* take in and produce. We might assume that *sum* and *times* take in and produce numbers, i.e., the meaning for the final <FORM> could simply be 7. On the other hand, we might assume that *sum* and *times* take in and produce *programs* whose executions yield numbers. For example, a LISP program can be obtained if *sum* and *times* are defined as follows:

```
times(a,b) =  (LIST 'ITIMES a b)
sum(a,b)   =  (LIST 'IPLUS a b)
```

The string 1+2*3 would rewrite to a <FORM> whose meaning is

```
(IPLUS 1 (ITIMES 2 3))
```

A *parser* is a program which takes in a grammar and an input string and always does one of two things. If the input string is a member of the language accepted by the grammar, the parser yields the meaning associated with the input string. Otherwise, the parser rejects the input string.

Meaning as Programs

Representing meanings as programs has the advantage that the evaluation of meaning can be delayed until the completion of the rewriting process even though meaning transformations are carried out during the rewriting process. The meaning transformations can be written to take in and produce programs whose later executions will carry out the intended meaning.

There are two major reasons for having meanings represented by programs rather than by computed values. First of all, a parser will invariably have to backtrack in its effort to find the particular sequence of rewrites by which the grammar's root phrase is derived from the input string. For example, a parser might at some time consider the following rewrite sequence:

```
    1       +       2       *       3
<DIGIT:1>       <DIGIT:2>
<NUMBER:1>      <NUMBER:2>
<ATOM:1>        <ATOM:2>
<TERM:1>        <TERM:2>
<FORM:1>
 <FORM:  sum( 1 , 2 ) >
```

This final <FORM> spanning *1+2* cannot be used in any successful derivation for 1+2*3. If *sum* were an expensive computation, the time taken to compute *sum(1,2)* would be a major loss. In addition, if *sum(1,2)* involved side effects, the side affects would have to be undone at some time. However, we can make *sum* both inexpensive and free of side effects by having *sum* return as its value a program whose later

execution will perform the expensive computation. The program can simply be represented by the address of a function along with two parameters. In this example, since *sum(1,2)* won't be a part of any successful derivation for 1+2*3, the program given by *sum(1,2)* will never be executed.

The notation

//[a;b;] *program* \\

will denote the datastructure which represents *program* along with the predefined parameters *a* and *b*. For example, *sum* can be defined by

sum(a,b) = *//*[a;b;] *expensive computation* \\

A call to *sum* yields a datastructure which contains the current values of *a* and *b* and the address of the program which implements *expensive computation*. To the *expensive computation*, the variables *a* and *b* always appear to contain the values they contained at the time *sum* was called. The notation

EX(x)

(EXecute) will denote the invocation of x where x is a program with predefined parameters. Thus, if we assign x as in

x := sum(a,b) ;,

performing

EX(x)

will invoke the *expensive computation*.

The second and perhaps more fundamental reason for representing meanings as programs rather than as computed values is simply that some meanings have values which depend on context not yet available at the time a particular rewrite is carried out. For example, consider a grammar where numbers can include radix specification, e.g.,

101 base 8 would be 65 base 10.

The number rules could be

<NUM> ::= <DIGIT>

<NUM> ::= <NUM> <DIGIT>

<NUMBER> ::= <NUM> BASE <DIGIT>

The part-of-speech <NUM> represents numbers without radix specification, i.e., strings of digits. A <NUMBER> is formed by appending a base specification to a <NUM>. Consider the meaning transformations for these rules. One is tempted to write

<NUM: a> ::= <DIGIT:a>

<NUM: *radix**a+b> ::= <NUM:a> <DIGIT:b>

<NUMBER: ?? > ::= <NUM:a> BASE <DIGIT:b>

The problem is that the *radix* won't be known when the second rule is applied. The *radix* becomes known only after the third rule applies. By agreeing that the meaning of a <NUM> will be not an integer, but a program whose execution will yield an integer, we can write the rules as follows:

<NUM: //[a;] a \\ > ::= <DIGIT:a>

<NUM: //[a;b;] *radix**EX(a)+b \\ > ::= <NUM:a> <DIGIT:b>

<NUMBER: *radix*:=b; EX(a) > ::= <NUM:a> BASE <DIGIT:b>

The first rule yields a program whose execution simply yields the <DIGIT>'s meaning. The second rule yields a program which includes the global variable *radix* as a free variable. The third rule, the <NUMBER> rule, produces an integer as its meaning by first setting the global variable *radix* and then invoking the program associated with <NUM>. Thus, the computation which must involve *radix* has been delayed until a time when *radix* is available.

In general, if meanings are represented as programs, the meaning transformation associated with a production can adequately control the context in which any of its parameters is evaluated. We will use the term *top-down context* to refer to context which is set by a routine for the evaluation of one of its parameters. For example, the <NUMBER> rule uses the global variable *radix* as top-down context. Top-down context can generally be used only if the evaluation of meaning is delayed until the completion of the rewriting process.

Parts-of-speech Are Datatypes

A *datatype* in its most general form is a set of conventions by which a datum exists. A datum is an *instance* of a datatype precisely when the datum obeys the conventions of the datatype. We can see how the parts-of-speech of a grammar serve as the datatypes over the space of meanings.

In defining a grammar and the routines which implement the meaning transformations, one must agree on how a meaning is represented. What sorts of actions will be performed by the evaluation of a meaning? For example, if we implement the routine *sum* for the rule

<FORM: sum(a,b)> ::= <FORM:a> + <TERM:b>

we must know three things: What kinds of objects are *a* and *b* and what kind of object must be associated with the resulting <FORM>? We can deduce that the type of data yielded by *sum(a,b)* must be the same as the type of data represented by *a* because *a* itself could be set to the result of a *sum*, e.g., the rewriting process might employ the <FORM:*sum(a,b)*> generated by this rule as the <FORM:*a*> in another application of this rule, e.g.,

c + d + b

<FORM: sum(c,d)>

<FORM: sum(*sum(c,d)* , b) >

When *a* is set to the meaning of a <FORM>, we cannot tell which production generated the <FORM>. Hence, each production which generates a <FORM> must associate a meaning which follows the same conventions as the meaning associated with any other <FORM>. In general, the only thing that can be known about a meaning is the part-of-speech with which the meaning is associated.

It is therefore advantageous to establish conventions for meaning on a part-of-speech by part-of-speech basis. That is, for a given part-of-speech, one should state exactly what can be expected of an associated meaning. A part-of-speech serves as the name for the conventions obeyed by any meaning associated with the part-of-speech. In the example production given above, we can assume that *a* follows the <FORM>-conventions and that *b* follows the <TERM>-conventions and finally that *sum(a,b)* had better follow the <FORM>-conventions. From the point of view of *sum*'s definition, these requirements appear as datatype

constraints:

```
sum(a:FORM  b:TERM) = FORM:  ...
```

Sum is a function which maps a FORM and a TERM to a FORM. Parts-of-speech are the datatypes for the input and output parameters in any function which implements a meaning.

For example, suppose we wish to write a compiler with the rules

```
<ATOM: load(a)>  ::=  <VARIABLE:a>
<PROGRAM: assign(a,b)> ::=  <VARIABLE:a> := <FORM:b> ;
```

We can make the following conventions: The *evaluation* of a meaning associated with

1) a <VARIABLE> yields a memory address and generates no machine code

2) an <ATOM>, <TERM>, or <FORM> generates machine code which will push an integer onto the stack

3) a <PROGRAM> generates machine code which will leave the stack level unchanged.

The meaning transformations for the two rules can be written as follows:

```
load(a:VARIABLE)= ATOM:
   //[a;]    Address := EX(a);
             Assemble   ' PUSH  Address '      \\
assign(a:VARIABLE  b:FORM)= PROGRAM:
   //[a;b;]
       EX(b)     " Generate code which will push the right
                   side of the assignment onto the stack "
       Address := EX(a);     " Where to store the result "
```

Assemble ' POP Address ' \\

Load maps a VARIABLE to an ATOM by producing a program whose execution will generate machine code which will push an integer onto the stack. The program produced by *load* uses its VARIABLE parameter by evaluating it to obtain the address for the variable. *Assign* maps a VARIABLE and a FORM to a PROGRAM by producing a program whose execution will generate machine code which will leave the stack level unchanged. The program produced by *assign* evaluates the FORM parameter to generate code which will push one word onto the stack. The program produced by *assign* finally evaluates the VARIABLE parameter and assembles a POP instruction to complete the assignment which brings the stack level back down.

The rule

<FORM: sum(a,b)> ::= <FORM:a> + <TERM:b>

can be added if we define *sum* as follows:

```
sum(a:FORM  b:TERM)= FORM:
   //[a;b;]
       EX(a)     " Generate code which will push
                   one word onto the stack "
       EX(b)     " Generate code which will push
                   another word onto the stack "
       Assemble an ADD instruction which pops two words
          off the stack and which finally pushes the sum
          back onto the stack    \\
```

The program produced by *sum* will indeed generate machine code which will push one word onto the stack.

Given an input string, if <PROGRAM> can be derived from a given input string, we can generate machine code which will implement the given string by performing

EX(*p*)

where *p* is the meaning associated with the derived <PROGRAM>.

Notice that if <PROGRAM> can be derived from the input string, the datatype requirements for the routines *load*, *assign*, and *sum* are automatically satisfied. The correctness of the compiler can be proven simply by proving

1) the correctness of each meaning transformation *and*

2) that each meaning associated with a given part-of-speech satisfies the established conventions for that given part-of-speech.

Ambiguity

It may be the case that a grammar's root phrase can be derived from a given input string in more than one way. For example, with the grammar

<FORM:a> ::= <NUMBER:a>

<FORM: exponent(a,b)> ::= <FORM:a> ↑ <FORM:b>

<FORM> can be derived from the string 2↑3↑4 in two ways:

```
    2       ↑       3       ↑       4
<NUMBER:2>        <NUMBER:3>        <NUMBER:4>
<FORM:2>          <FORM:3>          <FORM:4>
<FORM:    exponent( 2 , 3 ) >
<FORM:  exponent(  exponent( 2 , 3 ) , 4 ) >
- - - - - - - - - - or - - - - - - - - - -
    2       ↑       3       ↑       4
<NUMBER:2>        <NUMBER:3>        <NUMBER:4>
<FORM:2>          <FORM:3>          <FORM:4>
                  <FORM:   exponent( 3 , 4 ) >
<FORM:  exponent(  2 , exponent( 3 , 4 ) ) >
```

In the first case, the string is interpreted as (2↑3)↑4 whereas in the second case, the string is interpreted as 2↑(3↑4). We say that the string 2↑3↑4 is ambiguous with respect to the given grammar. This grammar could be modified so that it always groups 2↑3↑4 in one way and not the other. For example, to group left to right, substitute the second rule with

<FORM: exponent(a,b)> ::= <FORM:a> ↑ <NUMBER:b> .

To group from right to left, use

<FORM: exponent(a,b)> ::= <NUMBER:a> ↑ <FORM:b> .

It is desirable to have unambiguous languages, i.e., languages where a given string can have at most one meaning. However, it may be advantageous to have a grammar which produces multiple meanings for a given string so that some of these meanings can disqualify themselves on grounds other than syntactic structure. For example, ICL has the operators + and # where + is used to add either numbers or points and where # is used to combine two numbers to yield a point, e.g.,

1#2 is the point at x=1 and y=2,

1#2 + 3#4 is the point 4#6, and

1+2 # 3 is the point 3#3.

Consider how A+B#C might be grouped. If A, B, and C are numbers, A+B#C must be grouped as

(A+B)#C

because the grouping

A+(B#C)

would force + to add a number and a point. On the other hand, if A is a point and B and C are numbers, the latter grouping must prevail lest + be forced to add a point and a number. If A, B, and C are the names of program variables, the grouping decision can't be made until the types of A, B, and C are known. Since the types associated to variables are not known until declarations are processed and because declarations can't be processed until syntax analysis is complete, the grouping

decision cannot be dictated by the syntax grammar. The syntax grammar therefore must admit both groupings, i.e., yield two meanings for the string A+B#C. During the evaluation of meaning, the types for A, B, and C will become known and hence one of the meanings will disqualify itself.

In general, ambiguity is necessary when insufficient information is available for making a decision.

Multipass Language Processing

Consider two grammars which describe different aspects of a subset of FORTRAN's arithmetic expressions. The first grammar is the grammar relating the parts-of-speech <FORM>, <TERM>, and <ATOM> presented earlier. The second grammar is written in terms of the parts-of-speech <INTEGER> and <REAL>:

```
<INTEGER: addi(a,b)> ::= <INTEGER:a> + <INTEGER:b>
<REAL: addr(a,b)>    ::= <REAL:a> + <REAL:b>
<INTEGER: muli(a,b)> ::= <INTEGER:a> * <INTEGER:b>
<REAL: mulr(a,b)>    ::= <REAL:a> * <REAL:b>
<INTEGER:a>          ::= ( <INTEGER:a> )
<REAL:a>             ::= ( <REAL:a> )
<REAL: float(a)>     ::= <INTEGER:a>
```

This latter grammar states FORTRAN's datatype requirements and ignores operator precedence. The former grammar states FORTRAN's operator precedence but ignores datatype requirements. However, any legal arithmetic expression must be accepted by both grammars. For brevity, we will call the former grammar the syntax grammar and the latter grammar the type grammar.

Both grammars can be incorporated by agreeing that meanings associated with <ATOM>, <TERM>, and <FORM> are programs whose executions will generate phrases in the language accepted by the type-grammar. During the generation of these phrases, the type grammar instead of the syntax grammar will be active. Thus, each generated phrase will be subject to rewrites via the productions of the type-grammar. For example, *sum* can be defined as

```
sum(a:FORM  b:TERM) = FORM:
  //[a;b;]
     EX(a)   " Generate a phrase in the type language "
     Generate a "+" to the right of the phrase
          generated by a
     EX(b)   " Generate a phrase to the right of the + " \\ .
```

If *sum*'s *a* parameter generates the phrase

<INTEGER>

and if *b* generates the phrase <REAL>, the program produced by *sum* will generate the phrase

<INTEGER> + <REAL>

Since the type grammar is active during these phrase generations, <REAL> will be derived from this phrase:

<INTEGER:a> + <REAL:b>

<REAL: float(a)>

<REAL: addr(float(a) , b) >

As such, one of the phrases generated by *sum* is <REAL> standing alone.

To be more specific, the multipass scheme works as follows:

1) Process the input string with respect to the first grammar.

2) The result will be a meaning associated with the root phrase of the first grammar.

3) Evaluate the resulting meaning with respect to the second grammar.

4) From all the phrases generated by the evaluation, choose the root phrase of the second grammar.

The meaning associated with the second grammar's root phrase is now the meaning for the string with respect to both grammars. This multipass scheme indeed requires that the input string be accepted by both grammars.

More than two passes can be implemented by agreeing that the meaning transformations associated with one grammar will generate phrases in the language accepted by the next grammar in the sequence. The meaning transformations for the final grammar in the sequence will be responsible for carrying out the originally intended meaning.

Two successive grammars can be radically different so long as the meaning transformations associated with the first grammar can indeed generate useful phrases in the language accepted by the next grammar. The successive grammars need not be refinements of one another, e.g., it is not necessary for the type grammar to consist of several productions per syntax production. For example, the syntax production

<ATOM:a> ::= (<FORM:a>)

need not have any counterparts in the type grammar. Furthermore, several syntax productions might indeed generate the same phrase in the type language where the ultimate distinction between the two syntax productions resides in the *meanings* associated with the elements of the generated phrases.

Production Schema

We will now consider productions whose parts-of-speech may themselves be variables. A whole scheme of productions may be implemented by one or a few productions whose parts-of-speech are variables. For example, consider the set of productions

<*X*> ::= <*X*> EQUALS <*X*>

where <*X*> stands for any part-of-speech. Each of the productions

<INTEGER> ::= <INTEGER> EQUALS <INTEGER>

<REAL> ::= <REAL> EQUALS <REAL>

is a member of the production scheme given above.

The production scheme

<*X*> ::= IF <BOOL> THEN <*X*> ELSE <*X*>

represents the type requirements for the IF-THEN-ELSE construct. The type of an IF-THEN-ELSE expression is precisely the type of the THEN-clause when the type of the ELSE-clause matches the type of the THEN-clause. If the types of the THEN-clause and the ELSE-clause differ, this production does not apply.

One can write production schema where the variables representing parts-of-speech accept only a limited range of values. For example, let <EXPR> denote the array of parts-of-speech

$\langle EXPR_1 \rangle, \langle EXPR_2 \rangle, \ldots, \langle EXPR_n \rangle$

and let <BOP> (binary operator) represent the parts-of-speech

$\langle BOP_1 \rangle$, $\langle BOP_2 \rangle$, ..., $\langle BOP_n \rangle$

A precedence grammar is implemented by the production scheme

$\langle EXPR_i \rangle$ ::= $\langle EXPR_u \rangle$ $\langle BOP_i \rangle$ $\langle EXPR_v \rangle$

where u is required to be less than or equal to i and where v is required to be strictly less than i. This scheme is a generalization of the precedence grammar given earlier which included the parts-of-speech <ATOM>, <TERM>, and <FORM>. In this more general setting, we can agree that

$\langle ATOM \rangle = \langle EXPR_1 \rangle$,

$\langle TERM \rangle = \langle EXPR_2 \rangle$, *and*

$\langle FORM \rangle = \langle EXPR_3 \rangle$

and that the binary operators + and * have the rules

$\langle BOP_2 \rangle$ ::= *

$\langle BOP_3 \rangle$ ::= +

The requirements on u, i, and v in the rule scheme impose the same precedence requirements inherent in the <ATOM>-<TERM>-<FORM> grammar. The parenthesis rules

<ATOM> ::= (<FORM>)

has the counterpart

$\langle EXPR_1 \rangle$ ::= ($\langle EXPR_i \rangle$)

where i is required to be less than or equal to 3.

The implementation of operator precedence via this rule scheme is more efficient than the implementation offered by the <ATOM>-<TERM>-<FORM> grammar. The <ATOM>-<TERM>-<FORM> grammar includes

the bookkeeping rules

<TERM> ::= <ATOM>

<FORM> ::= <TERM>

Thus, whenever an <ATOM> is generated, the <ATOM> is rewritten to a <TERM> and the <TERM> is rewritten to a <FORM>, e.g.,

(1)

<ATOM>

<TERM>

<FORM>

< A T O M >

This cascading effect is absent in the precedence scheme: It is *not* the case that the $\langle EXPR_1 \rangle$ rewrites to $\langle EXPR_2 \rangle$ and finally to $\langle EXPR_3 \rangle$ like

(1)

$\langle EXPR_1 \rangle$

$\langle EXPR_2 \rangle$

$\langle EXPR_3 \rangle$

$< E X P R_1 >$

The production scheme doesn't need to include the bookkeeping rules

$\langle EXPR_{i+1} \rangle$ ::= $\langle EXPR_i \rangle$

because the precedence conditions, i.e., the conditions upon u, i, and v, are stated in terms of *inequalities* rather than in terms of equalities. Thus, for example, the string *(1)* parses simply as

(1)

$\langle EXPR_1 \rangle$

$< E X P R_1 >$

because the parenthesis rule requires only that i be less than or equal to 3 and *not* that i be equal to 3.

As the reader may recall, the conventions set upon *meanings* associated with the parts-of-speech <ATOM>, <TERM>, and <FORM> were identical. The distinction among these parts-of-speech was solely for syntactical rather then semantic reasons. Thus, grouping all the $\langle EXPR_i \rangle$ into one conceptual part-of-speech is natural from the point of view of setting up conventions for meaning.

The production schema presented thus far include variable parts-of-speech which admit either any part-of-speech or a specific array of parts-of-speech. However, the conditions placed on a variable part-of-speech can be of any sort we wish. The following example involves parts-of-speech which are themselves general datastructures.

Consider the datatype declaration

```
TYPE  A = { B } ;
```

This defines A to be a string, or array, each of whose elements is of type B. One effect of this declaration is the creation of the following datatype production:

```
<B>  ::=  <A> [ <INTEGER> ]
```

This production states that the result of indexing into an object of type A is an object of type B. If the user were to declare

```
TYPE  C = { D } ;
      E = { F } ;
      G = { H } ;
```

we would have the rules

<D> ::= <C> [<INTEGER>]

<F> ::= <E> [<INTEGER>]

<H> ::= <G> [<INTEGER>]

The meaning transformations associated with each of these rules are identical; each performs an indexing operation which does not depend on the datatypes involved. Because the meaning transformations are the same, we can take this opportunity to write one rule which will act as each of these individual rules:

$\langle type_1 \rangle$ ::= $\langle type_2 \rangle$ [<INTEGER>]

where $\langle type_2 \rangle$ = a type declared to be a string of elements of type $\langle type_1 \rangle$.

This rule scheme can be implemented if the part-of-speech $\langle type_2 \rangle$ is itself a datastructure which represents the structure of $type_2$. That is, the rule scheme can be written as

?? ::= <X> [<INTEGER>]

where the meaning transformation looks at the wild-card part-of-speech <X> to determine if it is a string of some other type. If <X> is not a string, the meaning transformation inhibits the application of the rule. If <X> is a string, the meaning transformation obtains that datatype of which <X> is a string and supplies this as the lefthand part-of-speech, the ??. The meaning transformation finally generates a meaning in terms of the meanings under <X> and the <INTEGER> and gives this as the meaning associated with the lefthand part-of-speech.

Rule schema are useful for minimizing the size and maximizing the readability of a grammar.

Reluctant Productions

Consider the productions

<INTEGER: addi(a,b)> ::= <INTEGER:a> + <INTEGER:b>

<REAL: addr(a,b)> ::= <REAL:a> + <REAL:b>

<REAL: float(a)> ::= <INTEGER:a>

This grammar has an inherent ambiguity. If the string 1+2 is viewed as a REAL, there are two possible derivations:

```
    1       +       2
<INTEGER:1>       <INTEGER:2>
<INTEGER:   addi( 1 , 2 )  >
<REAL:    float( addi(1,2) ) >
- - - - - - or - - - - - - -
    1       +       2
<INTEGER:1>       <INTEGER:2>
<REAL:float(1)>   <REAL:float(2)>
<REAL:  addr( float(1),float(2)) >
```

The first derivation employs *integer add* whereas the second derivation employs *real add*. The first derivation applies *float* to the result of the sum whereas the second derivation applies *float* to each operand previous to the sum.

To remove this ambiguity, we can introduce a notion of reluctant productions, i.e., productions which in some sense avoid being used. The notation

<REAL:float(a)> ::== <INTEGER:a>

will denote a reluctant rule. The reluctance of a rule manifests itself not during the rewrite process but after the rewritting process is complete. Upon completion of the rewrite process, all possible derivations for the input string are available. At this time we can choose those derivations which involve the minimum number of reluctant rules. For the example given above, the first derivation will win over the second derivation because the first involves one application of the *float* production whereas the second involves two applications.

For another example, consider the grammar

```
<A>  ::=  <B>
<B>  ::=  <C>
<C>  ::=  <D>

<A>  ::=  <X>
<X>  ::=  <D>
```

<A> can be derived from <D> in two ways:

D -> C -> B -> A *and*

D -> X -> A

If all these productions are reluctant, the path via X will be chosen over the path involving C and B.

Although reluctant productions can remove many ambiguities, there are some stubborn cases which defy disambiguation by this method. A notable example involves unary operators. With the rules

```
<INTEGER: minusi(a)>  ::=  -  <INTEGER:a>
<REAL:  minusr(a)>    ::=  -  <REAL:a>
```

the string

```
- 1
```

is ambiguous when viewed as a REAL and in fact both derivations involve the same number of reluctant productions. That is, does the *float* apply before the minus or after?

With a scheme of reluctance, we can afford to make a grammar more ambiguous and achieve a measure of optimization. For example, exponentiation is more efficient when the exponent is an integer. The rules

```
<INTEGER: expi(a,b)>  ::=  <INTEGER:a> ↑ <INTEGER:b>
<REAL:  expri(a,b)>   ::=  <REAL:a>  ↑  <INTEGER:b>
<REAL:  exprr(a,b)>   ::=  <REAL:a>  ↑  <REAL:b>
```

represent the three ways exponentiation is usually carried out. The second rule is clearly redundant because with the *INTEGER-to-REAL* coercion, the third rule alone supports all uses of the second rule. For example, the string 1.2↑3 parses either as

```
  1.2          ↑          3
<REAL:1.2>              <INTEGER:3>
                        <REAL: float(3)>
<REAL:   exprr ( 1.2 , float(3) ) >
- - - - - - or as - - - - - - - -
  1.2          ↑          3
<REAL:1.2>              <INTEGER:3>
<REAL:   expri ( 1.2 ,  3  ) >
```

The ambiguity generated by the inclusion of the second rule is welcome because we now have a choice of derivations. The second derivation will win over the first derivation because the reluctant *float* rule is not used in the second derivation whereas it is used in the first. Thus, the inclusion of rules which obviously admit ambiguity can indeed serve towards optimization in a scheme where some rules are reluctant.

AN EFFICIENT GENERAL REWRITE PARSER

This section documents an efficient, general rewrite parser. This parser accepts any general rewrite, or type 0 grammar whose productions have non-null righthand phrases. If the parser terminates, it yields all possible derivations in a factored form.

If we restrict our attention to context-free grammars, the parser works at a worst case expense equal to a polynomial function of the input character string length. The degree of the polynomial is equal to two plus the maximum length of each production's righthand phrase. If the number of parts-of-speech encompassed by the grammar is p, if the longest righthand phrase has length L, and if n is the input string length, then the worst case memory and time is bounded above by $(np)^{2+L}$. This is calculated for a grammar having all possible context-free productions with righthand phrases of length less than or equal to L.

The polynomial upper bound for memory includes the space taken by the resulting derivations. Even though there may be an exponential number of derivations, all the derivations together fit in polynomial space. There are two factors which yield this result. First of all, many derivations share common subderivations. This factor alone does not achieve the polynomial space but it does make possible the effectiveness of the second factor. The second factor involves extending the notion of derivation to include *ambiguous* derivations. An ambiguous derivation is a derivation which may contain instances of a new kind of node called an *OR*-node. A single ambiguous derivation can represent many distinct unambiguous derivations. The big payoff comes

when an ambiguous subderivation is shared by several derivations.

For example, the grammar

<X> ::= 1

<X> ::= <X> + <X>

represents all strings of characters representing sums of 1, e.g.,

1+1 *or* 1+1+1+1+1 *etc.*

This grammar gives rise to many derivations for a string having three or more *1*s because no preference is given to left or right associativity. The string

1+1+1+1

parses as any of

((1+1) + 1) + 1 *or*

(1 + (1+1)) + 1 *or*

(1+1) + (1+1) *or*

1 + ((1+1) + 1) *or*

1 + (1 + (1+1)).

The number of derivations arising from a string having *n 1*s equals the number of ways parentheses can be applied to the given string. This number excedes 2^{n-2}.

We can begin to see how all 2^{n-2} derivations fit in polynomial space by noting two examples. First of all, the initial "(1+1)" in both the first and third derivations can be represented once and can be shared. Secondly, the *ambiguous* derivation consisting of both

[(1+1) + 1] *and*

[1 + (1+1)]

can be shared by each of

(1 + *[1+1+1]*) + 1 *and*

1 + (*[1+1+1]* + 1).

A complete explanation for how the 2^{n-2} derivations fit into polynomial space will come when we prove the polynomial upper bound for the parser's expense.

The upper bound for expense applies even if the grammar has rules like

<X> ::= <X>

or like

<REAL> ::= <INTEGER>

<INTEGER> ::= <REAL>

Such "infinite loop" rules can give rise to infinitely many derivations. An infinite number of derivations is represented by a derivation containing cycles. As we shall see, rules like these come up' in many applications.

We will address the problems and advantages that come with processing an ambiguous derivation after the workings of the parser are presented. We shall see how an exponential number of derivations represented in polynomial space can often be processed in polynomial time.

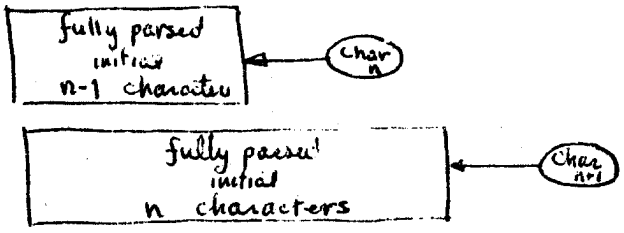
Although this parser was conceived independently, there are similarities with Earley's *Efficient Context-free Parser*[3] and with Thompson's REL parser[4]. Earley's efficient context-free parser has an upper bound on both memory and time of n^3 where n is the length of the input string of characters. The parser described here has an upper bound of n^4 for context-free grammars in Chompsky Normal Form. The disparity between this n^4 and Earley's n^3 comes about because Earley indexes into an array of length *n* where this parser walks a list structure of length *n*. The use of an array is cumbersome when dealing with general-rewrite grammars. The REL parser works for general rewrite grammars whose righthand phrases are non-null but its memory and time expense is unbounded even for context-free grammars. The key factor leading to Earley's efficiency is a continual factoring process which avoids duplicating work emanating from identical states.

The parser described here, like REL's, is bottom-up. The input string of characters is mapped into a list and this list is expanded to include nodes representing parts-of-speech spanning various substrings of the input string of characters. However, unlike REL's parser, a new node will *not* be added to the parsing graph if there already exists a node representing the identical part-of-speech and span. When an identical node is proposed, the derivation associated with the existing node is replaced by an *ambiguous* derivation consisting of both the derivations from the existing node and the new node. The grammar is *not* consulted with this new node because any responses by the grammar will have already occured once before when the existing node was proposed. The replacement of derivations is done in such a manner that all derivations which already reference the existing node's derivation will

automatically reference the ambiguous derivation.

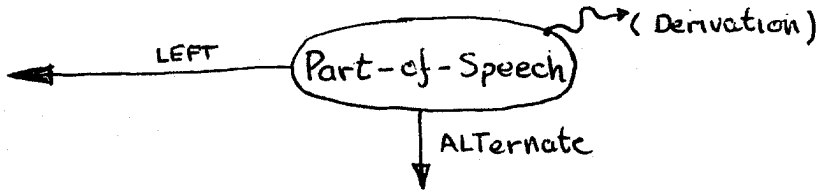
How The Parser Works

The parser works by taking in one character and appending it onto the righthand side of a completely parsed initial string of characters. The parser then calls on the grammar to achieve a complete parsing of the extended initial string of characters. This cycle repeats forever; the grammar is responsible for processing a command when it recognizes one.



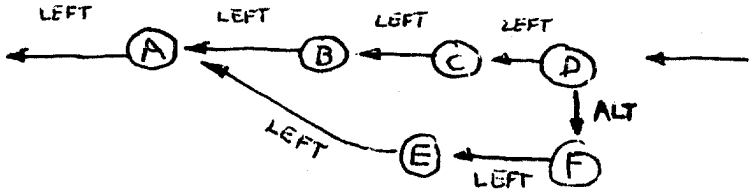
The Parsing Graph

The parser revolves around a central datastructure called the parsing graph. A parsing graph is a list structure each of whose memory elements has four fields:

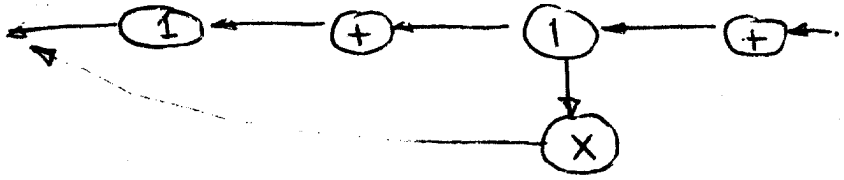


The LEFT and ALTernate fields each either contains NIL or points to another memory element of this same type. The part-of-speech is a scalar and the derivation is a reference to an arbitrary datastructure. A parsing graph provides a concise representation for an ambiguous phrase. In its basic form, a parsing graph is simply a phrase, e.g.,

An alternate subphrase may be incorporated by plugging the alternate subphrase into the ALTernate field of an existing node, e.g.,



For example, the parsing graph



represents the strings

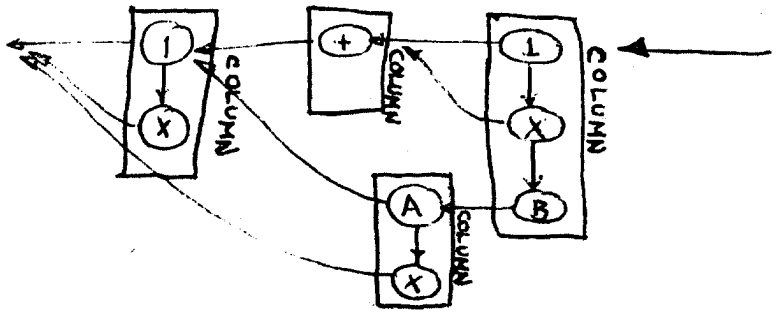
1 + 1 + *and* X +

For clarity, illustrations will exclude the derivation field.

Properties of the Parsing Graph

It is very useful to view the parsing graph in terms of *phrases* and *columns*. A *column* is any list of nodes linked together via their ALTernate pointers. A single column represents a set of alternative phrases, or paths. Each node in a column is the righthand element of a

phrase where the phrase is accessed by following nodes towards the left via LEFT pointers. At each step in traversing a phrase, the viewer confronts a column and hence has a choice for continued traversal.



A phrase is said to emanate from the column containing the phrase's rightmost node. It is also said that a column *contains* a *phrase* when the column contains the phrase's rightmost node. A node in a column is said to *reside* on that column.

We say that a given phrase is represented in the parsing graph iff there exists a sequence of parsing graph nodes such that

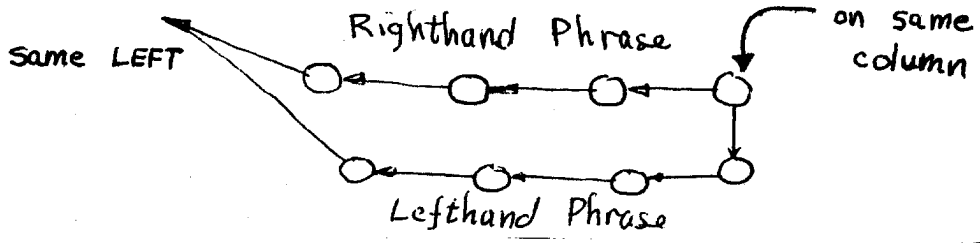
1) The parts-of-speech of the nodes match the parts-of-speech in the given phrase from right to left, and

2) Each node resides on the column referenced from the previous node's LEFT field.

We say that two phrases have the same *span* iff they both emanate from the same column and if the leftmost node in each phrase references the identical memory address via its LEFT field.

We say that a parsing graph is fully parsed when, for each phrase represented in the parsing graph, the following is true: *If* that phrase matches some production's righthand phrase, *then* an instance of the production's lefthand phrase also resides in the parsing graph and

indeed has the same span.

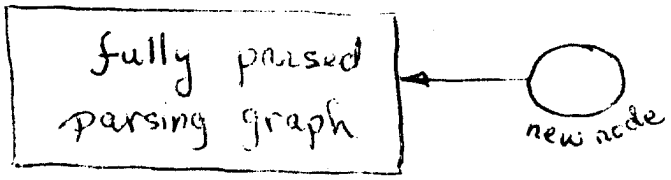


The Algorithm

The parser is implemented by two routines, one which enlarges the parsing graph and one which examines the parsing graph. The enlargement routine maintains the fully parsed property by consulting the grammar. The grammar is a program which calls on these two routines; a rule's righthand phrase examines the parsing graph to determine the rule's applicability and when applicable, the rule's lefthand phrase enlarges the parsing graph.

The basic idea is to give the grammar sight to each phrase represented in the parsing graph without giving sight to the same phrase twice in the same context. This is done in an incremental manner. If we assume that a given parsing graph is fully parsed, i.e., the grammar has already seen every phrase in the given parsing graph, then we can enlarge the parsing graph and see to it that the grammar sees each *new* phrase represented in the extended parsing graph and in fact sees each new phrase only once.

We allow a parsing graph to be extended in only one way: A new node may be placed to the right of a fully parsed parsing graph, i.e.,

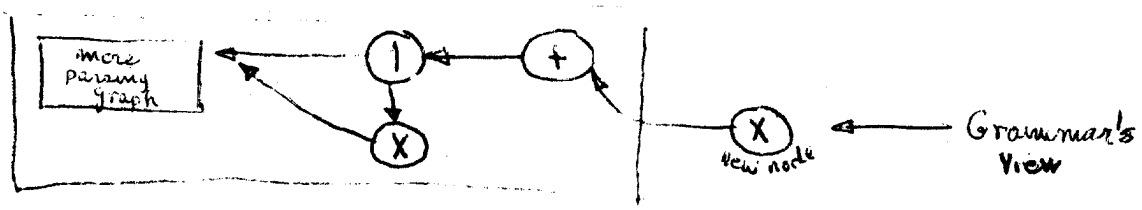


Whenever a new node is generated, the parser gives the grammar sight to the extended parsing graph. The grammar sees only those phrases which include the new node. The grammar sees no phrase which lies completely within the fully parsed parsing graph to the left of the new node.

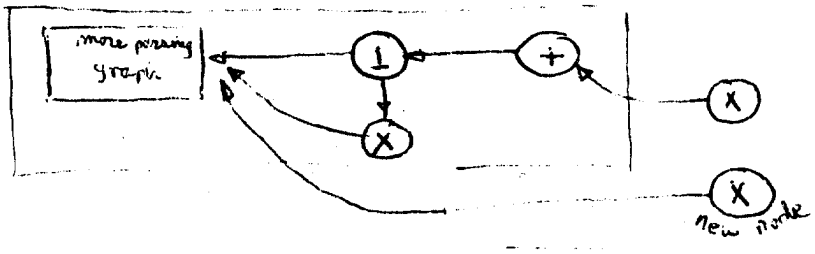
The grammar responds to each visible phrase in the extended parsing graph which matches a production's righthand phrase. For example, the grammar

```
<X>  ::=  1
<X>  ::=  <X> + <X>
```

responds to



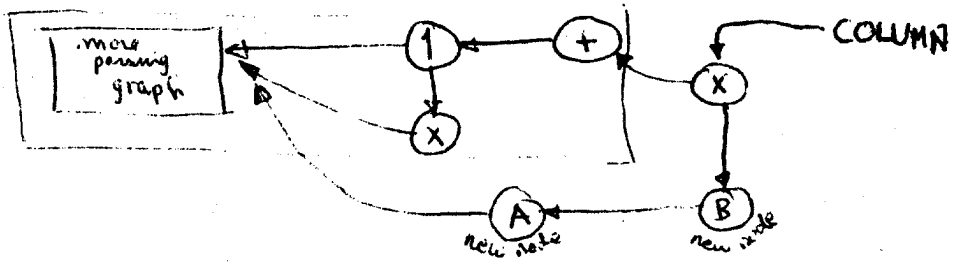
because the second production sees an <X>+<X> phrase. The response of a production is to enlarge the (sub)parsing graph residing to the left of the matched phrase. In this example, the second production generates a new X-node.



A general rewrite rule, e.g.,

<A> <B> ::= <X> + <X>

enlarges the parsing graph by generating a node for each part-of-speech on the lefthand side from left to right, e.g.,



First the A-node is generated and the grammar responds to those phrases visible from the new A-node. Finally, the B-node is generated and the grammar responds to all phrases visible from the new B-node.

Whenever a new node is generated, besides consulting the grammar, the parser places the new node on the column referenced by the global variable named COLUMN. COLUMN represents the righthand edge of newly generated phrases.

All new phrases reside initially on COLUMN. New phrases become incorporated into the parsing graph when a newer node is created which references COLUMN via the newer node's LEFT field.

Following are precise descriptions for the routine which generates parsing graph nodes, the routine which examines the parsing graph, and the routine which acts as the grammar.

Parsing Graph Generation

The routine which enlarges the parsing graph, NEWNODE, takes the following parameters:

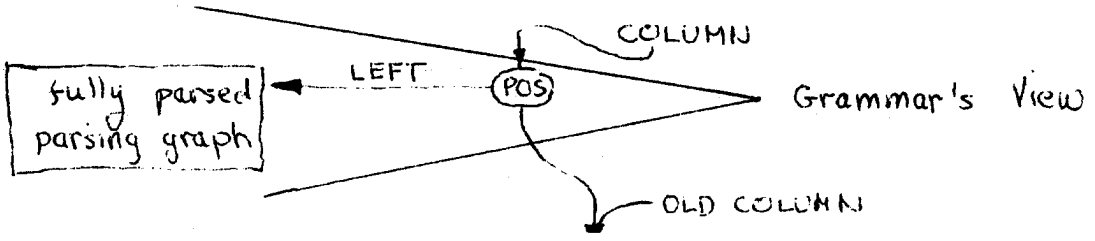
POS: *the part-of-speech for the new node*

SEM: *the derivation to be associated with the new node*

LEFT: *a fully parsed parsing graph which is to reside to the left of the new node.*

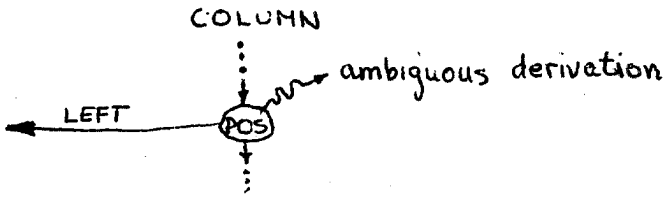
LEFT acts as the lefthand edge for the new node and the global variable COLUMN acts as the righthand edge. NEWNODE operates as follows: Look thru the list COLUMN for a node whose part-of-speech equals POS and whose LEFT equals the parameter LEFT.

If no match is found, form an extended parsing graph by constructing a node whose LEFT, part-of-speech, and derivation are the parameters LEFT, POS, and SEM resp. Put the new node on COLUMN and call the grammar passing this new node as point of reference. When the grammar returns, NEWNODE returns.



If a match is found, do not modify the parsing graph and do not

call the grammar. Rather, refer to the memory element which represents the derivation associated with matched existing node. *Modify* that memory element to represent the *ambiguation* of both the original derivation and the parameter SEM.



NEWNODE affects the global variable COLUMN only by appending to it.

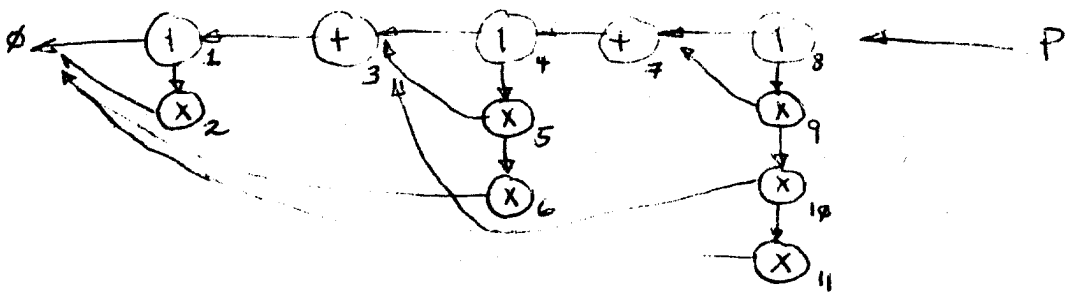
Parsing Graph Selection

The routine which examines the parsing graph, FIND, takes as parameters:

P: *the parsing graph, or column, to be examined*
RHS: *a phrase to be sought*
ACTION: *a process which is to be performed upon each match.*

FIND examines the parsing graph, P, looking for instances of RHS, the phrase to be sought. FIND looks only for phrases which emanate from the column immediately referenced by P. FIND views the parsing graph simply as a sideways tree; the ALTernate links are seen as brother links and the LEFT links are seen as son links. For each part-of-speech in RHS from right to left, FIND looks down a column for a node having the same part-of-speech, where upon finding a match, FIND continues the search by looking in the column referenced by the matched node's LEFT field. FIND will catch every matching phrase which emanates from the column P.

For example, given the following parsing graph in P



and the phrase

<X> + <X>

in RHS, FIND matches the phrases

$\langle X_5 \rangle \ +_7 \ \langle X_9 \rangle$ with LEFT= $+_3$

$\langle X_6 \rangle \ +_7 \ \langle X_9 \rangle$ with LEFT= $+_0$

$\langle X_2 \rangle \ +_3 \ \langle X_{10} \rangle$ with LEFT= $+_0$

Upon each phrase match, ACTION is performed. Available to ACTION are the derivations associated with each of the matched nodes. In addition, ACTION has access to the LEFT field of the leftmost matched node. ACTION is typically a process which, representing a rule's lefthand phrase, calls NEWNODE with each part-of-speech in the lefthand phrase. Along with each part-of-speech, ACTION will pass to NEWNODE a newly constructed derivation which references those derivations associated with each of the matched nodes.

The only backtracking in this parser occurs in FIND. The depth of backtracking is limited by the length of RHS, the phrase being sought. It turns out that FIND is always called with some production's righthand phrase shortened by deleting its rightmost part-of-speech. The expense

upper bound for this parser is based on the time spent in FIND where we know the maximum size of the parsing graph P.

The Grammar

The grammar is a program which accepts an extended parsing graph as parameter. An extended parsing graph is a single node whose LEFT references a fully parsed parsing graph. The grammar is always called from NEWNODE. We will call the single node the *new node*. It is a property of NEWNODE that the *new node* resides on the global variable COLUMN. However, even though the *new node* resides on COLUMN, the grammar will not consider any other node on COLUMN.

The grammar consists of a series of statements, one for each production. Let *RHS* denote the production's righthand phrase. Each statement has the form

IF the *new node* matches *RHS*'s rightmost part-of-speech THEN
    Call FIND with P= *new node*'s LEFT,
                   RHS= *RHS* less the rightmost
                        part-of-speech
                   ACTION= a process which generates .
                        this rule's lefthand phrase .

If *RHS* has only one part-of-speech, the call to FIND does not appear, rather, ACTION itself is performed where LEFT is set to *new node*'s LEFT, i.e.,

IF the *new node* matches *RHS*'s rightmost part-of-speech THEN

LEFT:= *new node*'s LEFT

generate this rule's lefthand phrase .

In each case, the rule generates its lefthand phrase in a context where LEFT references the parsing graph residing to the left of the matched phrase and where COLUMN contains the rightmost node in the matched phrase, the *new node*.

The process which generates the rule's lefthand phrase takes one of two forms. First, if the lefthand phrase has length one, e.g.,

<A> ::= ...

then the generating process is

POS:= *<A>*

SEM:= *some new derivation*

Call NEWNODE

NEWNODE places the new <A> node on the same column from which emanates the matched phrase, the column referenced by COLUMN. The LEFT for the new node references the same (sub)parsing graph which is referenced by the LEFT of the matched phrase. Indeed, both the matched righthand phrase and the generated lefthand phrase have the same span.

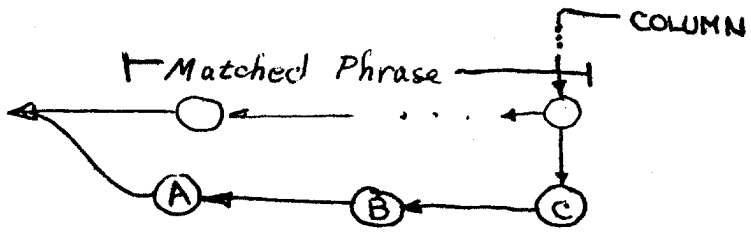
If the lefthand phrase has length greater than one, e.g.,

<A> <B> <C> ::= ...

then the generating process is as follows. Notice how each call to NEWNODE occures in a context where the resulting column from the previous call appears as the LEFT in the current call.

```
        OLD_COLUMN:= COLUMN        " Save COLUMN locally "
        COLUMN:= NIL

SEM:=      some new derivation
POS:=      <A>
Call NEWNODE

        LEFT:=    COLUMN
        COLUMN:= NIL

SEM:=      some new derivation
POS:=      <B>
Call NEWNODE

        LEFT:=    COLUMN
        COLUMN:= OLD_COLUMN        " Restore COLUMN "

SEM:=      some new derivation
POS:=      <C>
Call NEWNODE
```

The italicized sections set up position context. COLUMN is set to NIL for all but the rightmost node. That is, the positions for the <A> and <B> nodes have no place in any currently existing column. However, the rightmost node, the <C> node, is placed in the original column so that the new <A><B><C> phrase emanates from the original column.



Notice that the LEFT for the first node, the <A> node, is externally defined for this process. The LEFT upon entry to this process is, as always, the LEFT of the matched phrase. Indeed, the generated <A><B><C> phrase has the same span as the matched phrase, and in fact, starting from the <C> node, the <B> node resides in the column referenced by <C>'s LEFT and the <A> node resides in the column referenced by <B>'s

LEFT.

It should be noted that the THEN clause for each production modifies the variable COLUMN only by appending more nodes to the list. Hence, no matter in what order we assemble the productions, each production is entered with COLUMN still containing the *new node*, the rightmost node in any matched phrase.

In summary, each production requires that the *new node*'s part-of-speech matches the production's righthand phrase's rightmost part-of-speech. The rest of the righthand phrase is matched by FIND. Upon each match, the rule generates its lefthand phrase having the same span as the matched righthand phrase.

For example, the grammar

<X> ::= 1

<X> ::= <X> + <X>

translates to

IF *new node*'s POS= "1" THEN

Call NEWNODE with LEFT= *new node*'s LEFT

POS= <X> and

SEM= some new derivation

```
IF new node's POS= <X>  THEN
    Call FIND with  P= new node's LEFT
                    RHS=  the phrase  <X> +
                    ACTION= " Call NEWNODE with
                                POS= <X> and
                                SEM= some new derivation " .
```

Sample Run

Suppose the grammar is

<X> ::= 1

<X> ::= <X> + <X>

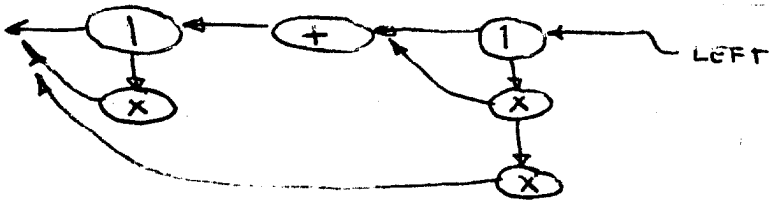
If we call NEWNODE with the following parameters:

COLUMN: NIL

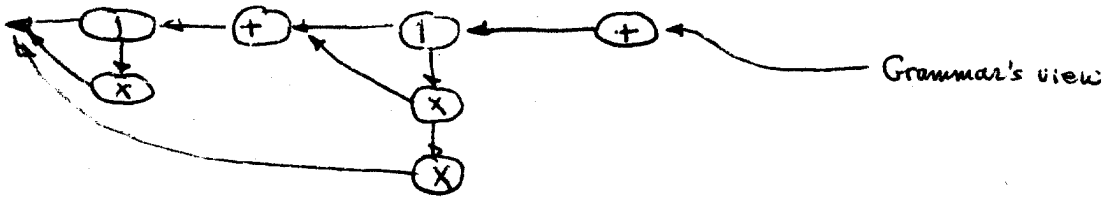
POS: the part of speech "+"

SEM: the NIL derivation

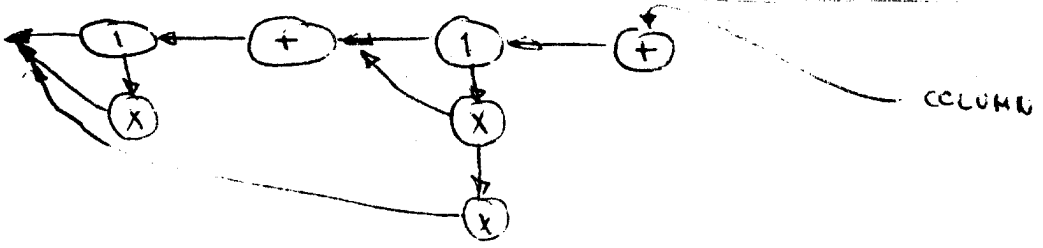
LEFT: a reference to the fully parsed parsing graph for input "1+1":



NEWNODE consults the grammar with the extended parsing graph



and returns the fully parsed parsing graph in COLUMN.



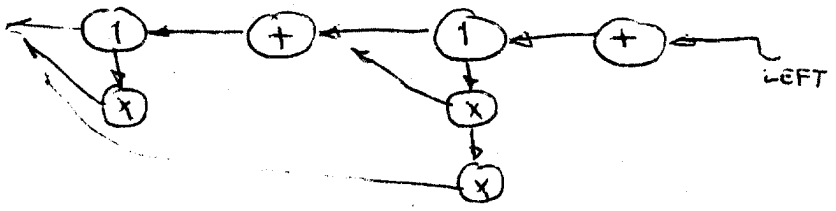
The grammar has added no additional nodes because the grammar has no rule whose righthand phrase can match the extended parsing graph *using* the *new* "+" node: No rule's righthand phrase has "+" as its rightmost part-of-speech.

Let us perform

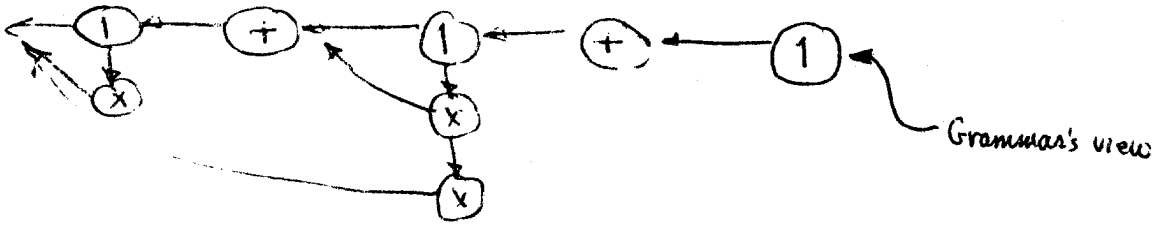
LEFT:= COLUMN *and*

COLUMN:= NIL

This moves our point of view to the right:



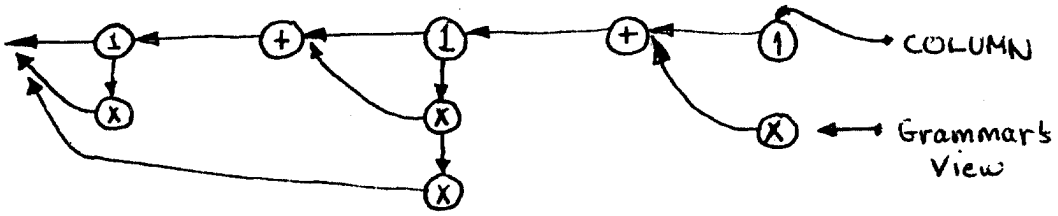
Now, if we call NEWNODE with a "1", NEWNODE will consult the grammar with the extended parsing graph



The grammar's production

<X> ::= 1

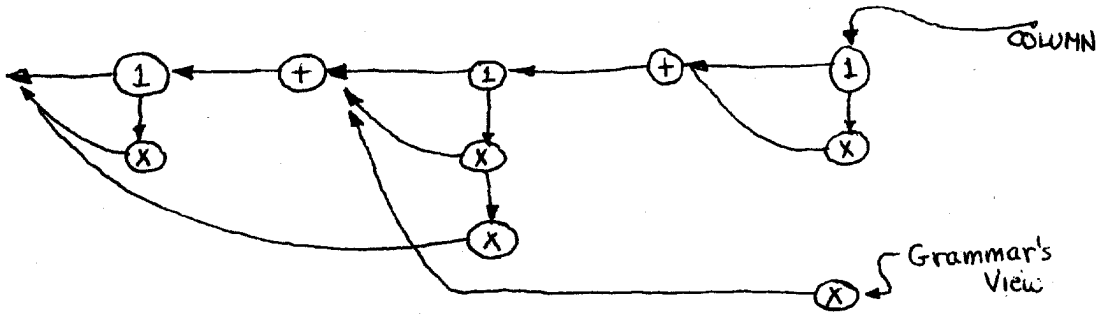
responds by calling NEWNODE with POS= <X> and with LEFT= *the LEFT of the 1-node*. The new incarnation of NEWNODE consults the grammar with the extended graph



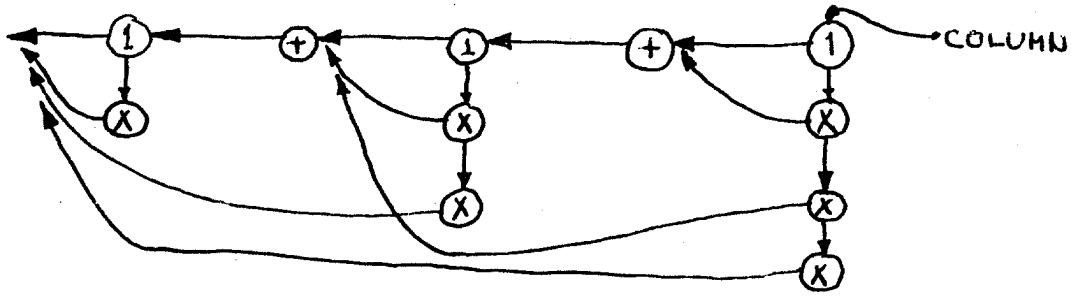
This time, the production

<X> ::= <X> + <X>

responds and in fact it responds twice. This production's righthand phrase has a choice of two X-nodes to the left of the +-node. The first response calls NEWNODE with POS= <X> and with LEFT= *the LEFT of the leftmost node in one of the matched <X>+<X> phrases*. NEWNODE consults the grammar with the extended parsing graph

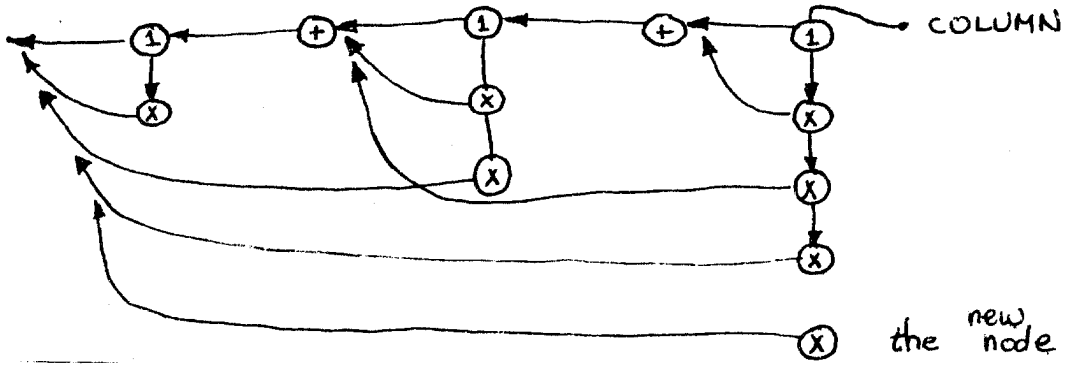


Without following further recursion, NEWNODE returns with



The full spanning X-node represents the parsing 1+(1+1). The second

response calls NEWNODE with POS= ⟨X⟩ and with LEFT= *the LEFT of the leftmost node in the other matched ⟨X⟩+⟨X⟩ phrase.*



The newly proposed X-node represents the parsing (1+1)+1. This time, however, NEWNODE does not consult the grammar; COLUMN already contains an X-node having the identical LEFT. Instead, NEWNODE modifies the derivation associated with the existing X-node so that it now represents the *ambiguous* derivation for both parsings.

Why The Parser Works

This section shows that if the parser terminates then *if and only if* the given input string can have a derivation in terms of the given grammar, the parsing graph will contain the phrase which consists solely of the grammar's root part-of-speech and which spans the whole input string. The next section places time and memory bounds on this parser for context-free grammars. These two facts together prove that the parser works at least for context-free grammars.

Let us assume that the character input routine works as follows:

```
COLUMN:= NIL
WHILE  there are more characters  DO
    LEFT:= COLUMN          " Step Right "
    COLUMN:= NIL

    POS:=  NEXT_CHARACTER
    SEM:=  NIL
    Call  NEWNODE
```

Each new character is placed to the right of the parsing graph which represents the previous characters. The input string *exists* as a phrase in the final parsing graph referenced from COLUMN because each character resides on that column which is referenced by the LEFT field of the node representing the next character.

To lend some precision to the following arguments, we shall state the following lemmas and assumption. The lemmas will be proven at the end of this section.

Lemma 1:

Any node which is accessible from some node's LEFT is never modified.

Lemma 2:

If two given nodes reside on the variable COLUMN at some time, then any column referenced by some node's LEFT either contains both of the given nodes or contains neither node.

Assumption 1:

No two parsing graph nodes reference the identical derivation node. In other words, any given derivation node is referenced by at most one parsing graph node. In the description for the parser given in the previous section, the assignment

SEM:= *some new derivation*

is meant to imply that SEM is set to reference a node which is currently referenced from nowhere else. This implication supports this assumption.

It is essential to note that a parsing graph node is examined only from the point of view of some other node's LEFT. The parsing graph examination routine, FIND, is always called with P containing a column which is taken from some existing node's LEFT. Thus, a node's appearence is important only when that node resides on a column which is referenced from some node's LEFT. Thus, the two lemmas help remove time considerations for all nodes which can be examined.

*Any parsing graph is always fully parsed if each node is entered by **NEWNODE**:* Choose any production and choose any phrase in the parsing graph which matches the production's righthand phrase. Consider this phrase's rightmost node.

This rightmost node was created at some time. By lemma 1, the parsing graph to the LEFT of this new node is now exactly what it was at the time the new node was created. Hence, the chosen righthand phrase was represented in the extended parsing graph at the time NEWNODE consulted the grammar with this rightmost node. *Assuming that **NEWNODE** did consult the grammar*, we know that the chosen production matched this phrase and hence generated its lefthand phrase having the same span. Because at this time COLUMN contained both the left and righthand phrases, lemma 2 guarantees that the lefthand phrase will always be seen to reside on any column which contains the righthand phrase. Therefore, both the righthand and the lefthand phrases exist in the parsing graph sharing the same span. In addition, the derivations associated with each node in the lefthand phrase were indeed created by this production having access to the derivations of the righthand phrase.

What about those cases where NEWNODE does not consult the grammar, i.e., those cases when there already exists a node in COLUMN having identical part-of-speech and LEFT? Because the existing node was itself generated by NEWNODE at an earlier time, the production's lefthand phrase already exists in the parsing graph. Consider what would happen if NEWNODE did indeed consult the grammar. The grammar would be called in exactly the same context in which it was called when the existing node was generated except that the parameter SEM may have a different

value. However, because this parser never looks at a derivation, i.e., a derivation is used only in constructing new derivations which reference the given derivation, the parameter SEM in no way affects the running of the parser. Therefore, consulting the grammar with the new node would generate only copies of existing nodes differing only in their associated derivations.

Even if NEWNODE does not consult the grammar, NEWNODE guarantees that each new derivation which would be generated by consulting the grammar will indeed be represented. Because the parameter SEM in no way affects the running of the parser, we can imagine that when the existing parsing graph node was generated, the parameter SEM could have been substituted with the new value for SEM. Exactly the same processing would ensue. Hence, by generating the existing node twice, once with old SEM and once with new SEM, we would come up with identical derivations which differ only by the value of SEM. By making SEM represent an *ambiguous* derivation consisting of both *old* and *new* SEM, all the derivations which would be generated by consulting the grammar with the newly proposed node will indeed be represented.

NEWNODE makes SEM represent the ambiguous derivation by modifying the derivation node referenced from the existing parsing graph node. NEWNODE replaces the original derivation by the ambiguous derivation. This guarantees at least that any derivation which accesses old SEM will now access the ambiguous derivation. That is, any derivation generated with reference to old SEM now represents both the original derivation and the same derivation where old SEM is substituted with new SEM.

We must verify that each node which accesses the modified derivation is a node which should see the ambiguous derivation in place of either the old or new derivations. Because the existing parsing graph node is the only parsing graph node which references old SEM by *assumption 1*, we can see that the only derivations which access the ambiguous derivation are those derivations which were built on account of generating the existing parsing graph node. These are precisely the derivations which should see the ambiguous derivation.

Any parsing graph node whose LEFT references a column containing the existing node would reference a column containing both the existing node and the newly proposed node if NEWNODE were to place the newly proposed node on COLUMN by lemma 2. Therefore, from any node's LEFT, both derivations would always be represented under identical parsing graph nodes even if NEWNODE were not to modify the existing node's derivation. In other words, there is no parsing graph node whose LEFT should see the original derivation instead of the ambiguous derivation.

Because this parser's correctness depends on the fact that a derivation is not examined during the parser's operation, anyone who writes a grammar must avoid examining derivations associated with parsing graph nodes. That is, the grammar cannot depend on derivations until the parsing is complete. The only information available at syntax, or parsing time is parts-of-speech and not derivations.

If the input string is a valid string in the language generated by the grammar, then the grammar's root part-of-speech exists as a phrase of length one sharing the same span with the input string: Assuming that the input string is an element of the language, we know that there exists a sequence of strings, each of which is derived from the previous string by rewriting some occurrence of some production's righthand phrase into that production's lefthand phrase. The final string in the sequence is the grammar's root part-of-speech standing alone. It is a fact that each string in this sequence exists in the fully parsed parsing graph and each shares the same span.

First of all, the input string exists in the parsing graph. Secondly, assuming that a given string in the sequence exists in the parsing graph, we can see that the next string in the sequence also resides in the parsing graph. Look at the portion of the given string which is rewritten to yield the next string. This portion is an instance of some production's righthand phrase. Because the parsing graph is fully parsed, we know that the parsing graph also contains an instance of the production's lefthand phrase having identical span. Hence we know that the next string in the sequence also resides in the parsing graph.

To be more precise, we can say that the righthand phrase is contained in the column referenced by the LEFT field of the string node to the right of the righthand phrase. By lemma 2 and because both the left and righthand phrases resided on COLUMN at some time, we can conclude that the column containing the righthand phrase also contains the lefthand phrase. Thus, the lefthand phrase is accessible from the string node

which references the righthand phrase.

Each full spanning string represented in the parsing graph is a string which can be obtained from the input string by performing legal rewrites. This says that the grammar's root part-of-speech can exist as a full spanning phrase only if the input string is a valid string in the language generated by the grammar. Suppose there is some full spanning string represented in the parsing graph which cannot be obtained by legally rewriting the input string. Let us call any such string an *illegal* string. A legal string is any string which can be obtained by performing legal rewrites upon the input string. We want to show that each full spanning string in the parsing graph is a legal string.

Let LHS denote any phrase in the parsing graph each of whose nodes was created by a single production. A LHS is the set of nodes generated by one execution of one production's lefthand phrase *not including* those nodes generated by further calls to the grammar. For example, a phrase ABC in the parsing graph is a LHS only if 1) there exists a production whose lefthand phrase is ABC and 2) each of the nodes representing A, B, and C were explicitly created in the same execution of the THEN clause for the production.

Any node ever generated is either an input character or a member of some LHS because any node not generated by the input routine is generated by the lefthand phrase of some production. Furthermore, any node is a member of at most one LHS because any given node is created by at most one execution of one production's lefthand phrase.

The following are two lemmas about LHSs:

Lemma 3:

Any column contains at most one node which is *not* the rightmost node in some LHS.

Lemma 4:

Let S be any string represented in the parsing graph. Let N be any node in S which is also the rightmost node in some LHS. One of the following must be true:

1) Each node in the LHS is a node in S, *or*

2) There exists a node in S which is both the rightmost node in some other LHS and which is accessible from N's LEFT.

Consider any full spanning string in the parsing graph except the original input string. This string, S, has the property that some LHS makes up a segment of S. That is, *there is some LHS each of whose nodes is in S.* Since S is not the input string, S contains a node, N, which is not an input character. N is therefore an element of some LHS. Let N be the rightmost node in S which is a member of some LHS. In S, the node to the right of N must be an input character and hence it must reference a column containing both N and an input character. By lemma 3, N must be the rightmost node in some LHS. By applying lemma 4, we see that either S includes all of LHS or S includes a node to the LEFT of N *which is itself the rightmost node in some LHS.* In fact, we can repeatedly apply lemma 4 as long as condition (2) shows up. However, each such application of the lemma increases a lower bound for the length of S. Therefore, condition (1) of the lemma must become true at

some time lest S be infinite.

We will define the *age* of a full spanning string, S, by averaging the ages of each node in S. The age of a node is precisely the amount of time which has elapsed since the node was created.

Suppose there exists an illegal string represented in the parsing graph. Let S be an illegal full spanning string of maximal age. That is, in some sense S is one of the first illegal strings created. From the preceding argument, we know that some segment of S is a LHS. We know that at the time LHS was on COLUMN, the righthand phrase of the production which created the LHS was also on COLUMN sharing the same span. By lemma 2, the column containing LHS also contains this righthand phrase. Hence, the string formed by substituting the righthand phrase for LHS in S is a string which resides in the parsing graph. Let *S1* denote the string formed from S by substituting LHS with the righthand phrase.

Because LHS was generated only after the righthand phrase had already been created, the age of each node in LHS is less than the age of each node in the righthand phrase. Thus, the age of S is less than the age of S1. By S's maximality, this older string, S1, must be a legal string. However, S is merely this older string where a righthand phrase has been rewritten to LHS. Hence, S itself is legal and we have a contradiction.

The Lemmas

All memory elements generated by this parser are of one of two kinds. One kind is a parsing graph node and the other kind is a derivation node. The following discussion is concerned mainly with parsing graph nodes and hence we will use the unqualified term node to refer to parsing graph nodes.

We will use the term *reference* to mean direct or immediate reference, i.e., a pointer references only the node which resides at the address contained in the pointer; the pointer does not reference nodes which are referenced from pointers within the referenced node. In contrast, we will use the term *accessible* to mean the transitive closure of reference, i.e., a given node is accessible from a given pointer iff there exists a sequence of nodes where the first node is referenced from the pointer and each node in the sequence contains a pointer referencing the next node and the final node in the sequence is the given node.

Def 1)

A column is any sequence of nodes where each node in the sequence references the next node via its ALT link and where the last node in the sequence has NIL as its ALT link. The head of a column is the first node in the sequence. A column contains a node iff the node is a member of the sequence.

The following definitions refer to the global variable named COLUMN at any given time:

Def 2)

A node resides on COLUMN iff that node is a member of the column whose head is referenced by COLUMN.

Def 3)

A node resides on OLD_COLUMN iff some existing incarnation of the local variable OLD_COLUMN references a column which contains the given node.

LEMMA A:

At any given time, the only nodes which might be modified are either nodes which reside on COLUMN or derivation nodes referenced from nodes on COLUMN.

LEMMA B:

Once a node ceases to reside on both COLUMN and OLD_COLUMN, the node will never again reside on COLUMN or OLD_COLUMN.

LEMMA C:

At a given time, no node resides both on COLUMN and on OLD_COLUMN.

LEMMA D:

Suppose two given nodes reside on COLUMN at some given time. From this time forward, we will see either both nodes residing on COLUMN or neither node residing on COLUMN.

LEMMA E:

A given node's LEFT references a value which was held by COLUMN at a time before the given node was created. In addition, before the given node was created, each node on the column referenced by the

given node's LEFT ceased to reside on COLUMN. More specifically, the value taken from COLUMN for a node's LEFT is a value which was held by COLUMN precisely at the time immediately before each node on COLUMN ceased to reside on both COLUMN and OLD_COLUMN.

LEMMA F:

Each node on the column referenced by some node's LEFT never resides on COLUMN.

Proof of A:

NEWNODE is the only routine which modifies nodes. NEWNODE will modify a node in one of two ways:

1) NEWNODE inserts a new onto the list COLUMN

2) NEWNODE modifies the derivation node which is referenced from a node on COLUMN.

Proof of B:

COLUMN is modified in one of three ways:

1) NEWNODE puts a newly created node on COLUMN

2) COLUMN is set to NIL

3) COLUMN is set to OLD_COLUMN.

(2) and (3) occur in a general rewrite production's lefthand phrase and only (2) occurs in the character input routine.

Suppose a given node is not on COLUMN and not on OLD_COLUMN. The only way a node gets onto COLUMN is by (1) and by (3). Because we are assuming that at some time the given node did reside on COLUMN or OLD_COLUMN, we know that the given node is one which already exists. Hence, (1) cannot put the given node on COLUMN. (3) cannot put the given node on COLUMN because the given node is not on OLD_COLUMN.

A given node is put on OLD_COLUMN only by

```
OLD_COLUMN:= COLUMN
```

However, because the given node is not on COLUMN, this assignment can't put it on OLD_COLUMN.

Proof of C:

Let B reside both on COLUMN and on OLD_COLUMN. B first resides on COLUMN because all newly created nodes first reside on COLUMN. B gets put on OLD_COLUMN only in the program text

```
OLD_COLUMN:= COLUMN
COLUMN:= NIL
```

After this operation, B no longer resides on COLUMN. B gets put back on COLUMN only by

```
COLUMN:= OLD_COLUMN
```

This occurs just before the generation of the last node in a general rewrite production. We may insert the statement

```
OLD_COLUMN:= NIL
```

immediately after the assignment to COLUMN because this incarnation of the local variable OLD_COLUMN will no longer be used. Hence, when B gets put back onto COLUMN, B no longer resides on OLD_COLUMN.

Proof of D:

Let A and B be nodes both of which reside on COLUMN at some time. Consider the first operation which deletes either A or B from COLUMN. This operation is one of

2) COLUMN:= NIL *or*

3) COLUMN:= OLD_COLUMN .

If the operation is (2), both A and B are removed from COLUMN. Operation (3) also removes both A and B from COLUMN because neither A nor B resides on OLD_COLUMN by lemma C. Thus, the first

operation which removes one of A and B from COLUMN removes both A and B.

Consider the operation which puts one of A or B back onto COLUMN. This occurs by

COLUMN:= OLD_COLUMN .

If both A and B reside on OLD_COLUMN, then both A and B will return to COLUMN. If it can ever be that exactly one of A or B resides on OLD_COLUMN, let us consider the first operation which puts exactly one of A or B on OLD_COLUMN:

OLD_COLUMN:= COLUMN .

COLUMN itself must have contained exactly one of A and B at some previous time. However, looking at the two assignments above, we can see that COLUMN can enter this state only if OLD_COLUMN contained exactly one of A and B at an earlier time.

Proof of E:

Let N be any node. Because the LEFT field of any node is never changed once the node is created, N's LEFT is still exactly what it was when N was created. N's LEFT is therefore the value held by NEWNODE's parameter LEFT at the time N was created. NEWNODE's parameter LEFT gets set in one of three ways:

1) LEFT is taken from the LEFT field of an existing node.

2) LEFT:= COLUMN

COLUMN:= NIL

3) LEFT:= COLUMN

COLUMN:= OLD_COLUMN

NEWNODE is called immediately after one of these operations and hence N is created after one of these operations is complete.

Consider (2) and (3) first. In each case, LEFT is indeed set to a value held by COLUMN. *In fact, immediately after LEFT is assigned COLUMN, each node on COLUMN ceases to reside on both COLUMN and OLD_COLUMN:* Let C be a node which is initially on COLUMN. C's residence on COLUMN implies that C is not on OLD_COLUMN by lemma C. Thus in both (2) and (3), COLUMN is set to a value on which C does not reside.

In the case of (1), the value for LEFT is taken from the LEFT field of an existing node. Let M be the first node ever created whose LEFT field is that of N. Since M is the first node created with the given value for LEFT, M's LEFT had to have been set by (2) or by (3). Thus, M's, and hence N's LEFT is indeed a value held by COLUMN at the time immediately before each node on COLUMN ceases to reside on both COLUMN and OLD_COLUMN.

Proof of F:

Let C be a node on the column referenced by the LEFT field of another node, N. By lemma E, C ceased to reside on COLUMN and OLD_COLUMN before N was created. By lemma B, each node on the column referenced by N's LEFT will never reside on COLUMN now that N exists.

Proof of Lemma 1

By lemma A, we merely need to show that any node accessible from a given node's LEFT never resides on COLUMN. All nodes accessible from the given node's LEFT are precisely

1) the nodes residing on the column referenced by the given node's LEFT *and*

2) all nodes accessible from the LEFT field of each node on the column.

Thus, all nodes accessible from the given node's LEFT reside on columns which are themselves referenced from some nodes' LEFTs. Hence by lemma F, each node accessible from the given node's LEFT can never reside on COLUMN.

Proof of Lemma 2:

Suppose nodes A and B reside on COLUMN at some time and suppose that N is a node whose LEFT references a column containing A. We will show that N's LEFT references a column containing both A and B.

By lemma E, N's LEFT references a value held by COLUMN at some time. By lemma F, the column referenced by N's LEFT never changes once N is created. Thus, N's LEFT is a value which was held by COLUMN when A resided on COLUMN. If B also resided on COLUMN at this time, then the column referenced by N's LEFT always contains A and B.

Since both A and B reside on COLUMN at some time by hypothesis, lemma D guarantees that both A and B reside on COLUMN immediately before A ceases to reside on both COLUMN and OLD_COLUMN; A will never again reside on COLUMN. Lemma E guarantees that N's LEFT was taken from COLUMN immediately before A ceased to reside on both COLUMN and OLD_COLUMN. Therefore, COLUMN contained both A and B when N's LEFT was set to the value in COLUMN.

Proof of 3:

Consider how the grammar puts nodes onto COLUMN. Look at a production's lefthand phrase. Each non-rightmost node is placed on COLUMN in the context

```
COLUMN:= NIL
Call  NEWNODE
```

Thus, when a non-rightmost node is placed on COLUMN, no other nodes reside on COLUMN. Hence COLUMN always contains at most one non-rightmost node of a LHS.

Similarly, each node generated by the character input routine is placed on COLUMN in the context

```
COLUMN:= NIL
Call  NEWNODE
```

Thus, COLUMN can contain at most one of either an input character or a non-rightmost node in a LHS. The only other nodes on COLUMN are the rightmost nodes of LHSs.

Because the LEFT field of any node is a value once held by COLUMN, any accessible column contains at most one node which is not the rightmost element in a LHS.

Proof of 4:

Suppose there is a node in LHS *not* in S. Let K be the rightmost node in LHS which is *not* in S. K is not the rightmost node in LHS because the rightmost node in LHS is in S by assumption. Let R be the node in LHS just to the right of K. R is in S. Therefore, R's LEFT references a column containing both K and a node in S. By lemma 3, we conclude that the node in S on the column containing K is the rightmost node in some LHS because K itself is not the rightmost node in the LHS containing K. Futhermore, the column containing K is accessible from N's LEFT because N is the rightmost node in LHS.

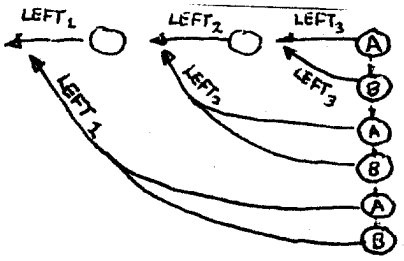
An Upper Bound for Parser Expense with Context Free Grammars

The time and memory expense for this parser with context free grammars is a polynomial function of the input string length. We find this because a context free grammar always gives rise to a special sort of parsing graph. Any given node's LEFT field is precisely the LEFT field of a node representing an input character. Therefore, if n characters have been taken in, any node in the whole parsing graph can have one of at most n possible values in its LEFT field. Furthermore, because NEWNODE avoids placing duplicate nodes on COLUMN, COLUMN can contain at most $n*p$ nodes where p is the number of parts-of-speech encompassed by the grammar. Knowing that there are at most n values for LEFT and that the column referenced by a LEFT has at most $n*p$ nodes, we can conclude that the parsing graph has at most n^2*p nodes. This size limits the number of phrases which can be matched and hence the number of times that each of NEWNODE, FIND, and the grammar can be called.

The rest of this section presents a more precise formulation for the memory and time bounds in context free grammars. Let n be the number of input characters processed up to now. Let p be the number of parts-of-speech encompassed by the grammar. Let L be the maximum length of any production's righthand phrase.

There are at most n values for LEFT. The input routine has created n values for LEFT by having taken in n characters. The context free grammar always sets LEFT to the value of the LEFT field in some existing node because each production has a lefthand phrase of length one.

At any time, COLUMN contains at most $n*p$ nodes. Consider that NEWNODE does not place a node on COLUMN if COLUMN already contains an identical node. NEWNODE considers two nodes identical when both the LEFT fields match and when the parts-of-speech match. Since there are n possible values for LEFT and p possible parts-of-speech, there are at most $n*p$ distinct nodes which can reside on COLUMN at any one time.



2 parts-of-speech: A and B
3 values for LEFT

Because the LEFT field for a node references a value once held by COLUMN, any node's LEFT references a column of length at most $n*p$. Thus, the maximum number of phrases which have length less than or equal to L and which emanate from a given column is $(n*p)^L$. Going from right to left, there are $n*p$ choices at each of L stages.

Consider how much time it takes to build one column. For each distinct node which NEWNODE places on COLUMN, NEWNODE calls the grammar. We can conclude that NEWNODE calls the grammar at most $n*p$ times in forming a single COLUMN. Upon each call, the grammar can match at most $(n*p)^{L-1}$ phrases which include the new node. Thus, the time spent in a single call to the grammar *excluding the grammar's calls to NEWNODE* is $(n*p)^{L-1}$. The grammar can call NEWNODE at most $(n*p)^{L-1}$ times because each phrase match can generate at most one node.

Each time the grammar calls NEWNODE, NEWNODE takes at most $n*p$ time to see if the newly proposed node already resides on COLUMN. If NEWNODE does not consult the grammar, then no futher time is taken by NEWNODE. If, on the other hand, NEWNODE does consult the grammar, we will add zero time because we will count this call to the grammar as one of the total $n*p$ times that NEWNODE calls the grammar with the current COLUMN.

Therefore, the time taken to form one COLUMN is the product of

1) $n*p$ calls to the grammar

2) $(n*p)^{L-1}$ in each call to the grammar excluding the grammar's calls to NEWNODE *and*

3) $n*p$ in each of the grammar's calls to NEWNODE.

This yields $(n*p)^{L+1}$.

The nodes on COLUMN cease to reside on COLUMN precisely when the input routine takes in another character. Therefore, the cost of processing n characters is at most n times the cost of building one column. Hence, this parser processes n characters in at most $k*n^{L+2}$ time where k does not depend on n.

The number of both parsing graph and derivation nodes built by the parser is bounded by the amount of time spent in the parser. We can conclude that the parser creates at most $k*n^{L+2}$ nodes where k does not depend on n. Even though there are at most $p*n^2$ parsing graph nodes, there are a lot of derivation nodes. Consider that each time NEWNODE is called, NEWNODE either creates a parsing graph node or creates a derivation *OR*-node. Since NEWNODE may be called n^L times in forming one column, NEWNODE may indeed create n^L-n derivation *OR-nodes*.

A SEMANTIC EVALUATOR

This section documents a set of programs by which a derivation, or meaning, is evaluated. We will assume that meanings are represented by programs as suggested in the section about languages. If meanings were not represented by programs, there would be no need to evaluate a meaning. The operator EX() is one of the main semantic operators. EX has already been described in the section about languages; EX is equivalent to LISP's EVAL function. All the other semantic operators are for dealing with and optimizing the evaluation of ambiguous derivations.

Let us see how an ambiguous derivation comes to be. With the productions

```
<A: f(b)>  ::=  <B:b>
<B: g(c)>  ::=  <C:c>

<A: h(c)>  ::=  <C:c>
```

<A> can be derived from <C> in two ways. The string <C:c> parses as

```
<C:c>
<B: g(c)>
<A: f(g(c))>
<A: h(c)>
```

Referring to the parser presented in an earlier section, when the grammar proposes the second <A> node, NEWNODE sees that there already exists an <A> node having identical span. NEWNODE, therefore, does not make a new <A> node, rather, NEWNODE modifies the derivation associated with the existing <A> node so that it now represents both derivations.

Thus, we really get

<C:c>

<B: h(c)>

<A: OR(f(h(c)) , g(c)) >

Instead of having two <A> nodes, we have one <A> node which has an ambiguous derivation. If we add the rule

<D: y(a)> ::= <A:a>

<D> will be derived from <C:c> with the meaning

y(OR(f(h(c)) , g(c)))

Thus, OR-derivation elements may be arbitrarily nested within derivations.

OR-Derivation Nodes and The Routine SEMOR

An OR-node references a routine called SEMOR. That is, EX applied to an OR-node simply calls SEMOR exactly as EX would call any other program which implements a meaning. SEMOR is a routine not supplied by the language writer; SEMOR comes with the semantic evaluator because the parser may generate OR-derivation nodes independent of language.

What does SEMOR do? Because an OR-derivation node may show up as the meaning associated with any given part-of-speech, SEMOR must be compatible with all possible meaning conventions. This presents a major problem. Following are three classes of meaning conventions, each of which requires a different action to be performed by SEMOR.

Meanings of the First Kind

The simplest and most restrictive meaning conventions are those which agree that a meaning is a computed value rather than a program. A part-of-speech which adheres to such a meaning convention can never afford to have an OR-node involved in its meanings. For example, if the meaning associated with a given part-of-speech is supposed to be an integer, the appearance of an OR-derivation node in place of a single integer will undoubtedly result in a faulty meaning when the OR-node is interpreted as an integer. Because the meaning associated with such a part-of-speech is not evaluated, SEMOR will never even gain control. Thus, no matter how SEMOR is defined, meanings which are not programs cannot afford OR-nodes. A part-of-speech which has such meaning conventions must be one which can be derived from any given input string in at most one way, lest an OR-node show up in its meaning.

Meanings of the Second Kind

The second kind of meaning conventions are those which agree that a meaning is a program *where the program may produce side effects* or where the program yields a datastructure which is not capable of representing ambiguity. Such conventions differ from the previously mentioned conventions in that a meaning will be evaluated rather than simply fetched. Thus, the appearance of an OR-node in such a meaning will at least give control to SEMOR. For example, a meaning which adheres to such conventions is a meaning whose evaluation generates machine code in some global array. Another example is a meaning whose evaluation yields an integer. The type *integer* is not capable of representing an

ambiguous integer, i.e., two integers.

For meanings of the second kind, SEMOR must not evaluate both of its parameters. The best SEMOR can do in general is to evaluate exactly one of its parameters. This makes SEMOR an identity function, and as such, SEMOR is compatible with all meaning conventions. However, this particular arrangement for SEMOR introduces arbitrary decisions and hence should be used only as the last resort in language processing. We can expect that this will be a legitimate treatment for ambiguity which is not resolvable by the given language. It may be appropriate for SEMOR to inform the user of the existence of an ambiguity.

Meaning conventions of the first kind can be mapped into meaning conventions of the second kind by agreeing that a meaning will be a program whose evaluation will simply yield a previously computed value. This reorganization has the advantage that the existence of an ambiguity will at least be detected.

Meanings of the Third Kind

The third kind of meaning conventions is the most general and perhaps the most useful. A meaning of the third kind is one whose evaluation generates a datastructure which itself is capable of representing ambiguity. Under such conventions, SEMOR should evaluate each of its parameters and yield the datastructure which represents the ambiguation of both results. This technique introduces no arbitrary decisions and properly preserves ambiguity. An example of a meaning of this kind is a meaning whose evaluation generates a parsing graph. A parsing graph is definitely a datastructure capable of representing

ambiguity. In this example, SEMOR should evaluate each of its two parameters and merge the two resulting parsing graphs. A precise description for this scheme will follow shortly.

What SEMOR Does

In the implementation, SEMOR's default action supports meanings of the third kind where the ambiguous datastructure is required to be a parsing graph. For meanings of the second kind, prior to their evaluation, some program must modify SEMOR so that it acts as the identity and as such evaluates only one of its parameters. There is never any question as to which action SEMOR should be set to perform. Since any meaning is acquired with relation to a particular part-of-speech, the conventions for meaning under that part-of-speech clearly imply whether the meaning is of the second or third kind. As has turned out in practice, there have been very few places where SEMOR must be redefined. Typically, meanings referenced from within a given meaning are all of the same kind.

For example, in the ICL compiler, most meanings are of the third kind. ICL is a three pass compiler implemented as described in an earlier section about multipass language processing. The meanings associated with the first pass generate phrases in the language of the second pass and likewise from the second to the third pass. Thus, the meanings associated with the first and second passes are meanings of the third kind. An exception is made for the processing of declarations: Under the syntax part-of-speech for declarations, meanings are of the second kind; their executions make global modifications to both the

symbol table and the grammar of the second pass. Finally, the meanings associated with the third pass are meanings of the second kind; their execution generates machine code in a global array.

How An Ambiguous Derivation Generates a Parsing Graph

The following is a set of conventions by which a meaning can generate a parsing graph. A meaning will have access to two global variables named LEFT and COLUMN. These two variables will define the span of a phrase just as they do in the parser during the generation of a production's lefthand phrase. A meaning, therefore, will use LEFT and COLUMN and act exactly like a production's lefthand phrase. For example, *sum* in the rule

<FORM: sum(a,b)> ::= <FORM:a> + <TERM:b>

will produce a meaning whose evaluation will generate a polish postfix phrase if *sum* is defined as follows:

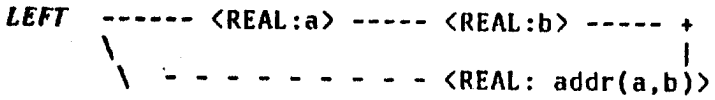
```
sum(a:FORM  b:TERM) = FORM:
 //[a;b;]      " LEFT and COLUMN are now input parameters "
       Let OLD_COLUMN be local
    OLD_COLUMN:= COLUMN          " Save COLUMN "
    COLUMN:= NIL

    EX(a)        " Let a generate its phrase "

    LEFT:= COLUMN       " Step Right "
    COLUMN:= NIL

    EX(b)        " b generates its phrase "

    LEFT:= COLUMN       " Step Right "
    COLUMN:= OLD_COLUMN   " Restore COLUMN "

    SEM:= NIL            " Generate + "
    POS:= "+"
    CALL  NEWNODE          \\
```

This phrase generation program is very nearly identical to that of a production's lefthand phrase. The only difference is that EX is used to generate some of the subphrases. Indeed, the evaluation of the program yielded by *sum* will generate a phrase whose leftmost node's LEFT references the value held by the input variable LEFT and whose rightmost node is placed on COLUMN. Just like a production's lefthand phrase, COLUMN will be modified only by appending more elements onto COLUMN.

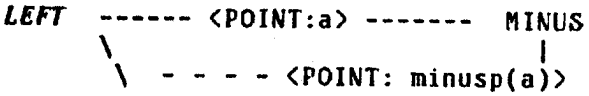
What should SEMOR do under these conventions? SEMOR needs merely to keep LEFT constant over the evaluation of both parameters, i.e.,

```
SEMOR(a,b):
   Save LEFT
   EX(a)      " Let one possibility generate its phrase as
                though it were the only possibility "
   Restore LEFT    " LEFT may very well have been damaged "
   EX(b)      " Let the other possibility generate its
                phrase over the same span "
```

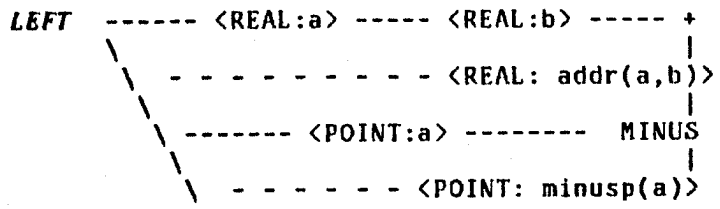
The phrases generated by *a* and *b* are placed on the same column and each shares the same span. The order in which *a* and *b* are evaluated is irrelevant. For example, if *a* generates the phrases



and if *b* generates the phrases



then SEMOR will leave COLUMN containing



Two Sources of Ambiguity

With this scheme, ambiguities, i.e., alternate phrases, are created by two distinct means. First of all, ambiguities in the first grammar of a multipass scheme generate derivations containing OR-nodes and the evaluation of these OR-nodes combine independently generated phrases to form alternate phrases. Secondly, even in the absence of OR-nodes, ambiguities in the second grammar will themselves generate alternate phrases. In the example given above, the first two parsing graphs consist of alternate phrases generated exclusively by the second grammar, the grammar with the rules

<REAL> ::= <REAL> <REAL> +

<POINT> ::= <POINT> MINUS

The third parsing graph, the parsing graph generated by SEMOR, contains alternate phrases brought together by SEMOR and not by the second grammar.

Both sources of ambiguity manifest themselves in exactly the same way. Each appends alternate phrases onto COLUMN. Thus, distinctions between the two sources of ambiguity disappear. This is appropriate when one considers that each of the alternate phrases offers a valid

interpretation of the evaluated meaning and that the alternate phrases do not interact with one another.

Locality of Ambiguity

The existence of OR-nodes represents not only ambiguity, but also locality of ambiguity. For example, the derivations

sum(OR(a,b) , OR(c,d)) *and*

OR(sum(a,c) , sum(b,c) , sum(a,d) , sum(b,d))

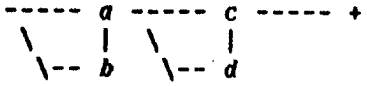
present the same alternatives but the first derivation represents a tighter locality.

Given a derivation containing OR-nodes, one can imagine expanding the derivation by bringing OR-nodes from the inside out. In fact, OR-nodes can be brought all the way out to the top level, thus yielding a *set* of derivations, each devoid of OR-nodes. This kind of expansion destroys locality of ambiguity.

The locality of ambiguity represented by OR-nodes is preserved by the evaluation of a meaning. For example, if a,b,c, and d generate the parsing graphs *a*,*b*,*c*, and *d* respectively, the derivation

sum(OR(a,b) , OR(c,d))

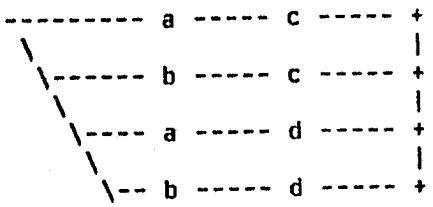
generates the parsing graph



whereas the less localized derivation

OR(sum(a,c) , sum(a,d) , sum(b,c) , sum(b,d))

generates the less localized parsing graph



Local ambiguity generated by the first grammar in a multipass scheme gives rise to localized OR-nodes. As shown here, the localized OR-nodes give rise to localized ambiguity in the generated parsing graphs for the second grammar. Thus, locality of ambiguity is preserved from one pass to the next.

In fact, during the generation of parsing graphs for the second grammar, NEWNODE still collapses parsing graph nodes of identical span. Hence, OR-nodes may come to exist in the derivations under phrases in the second grammar. For example, if *a* generates

<INTEGER:x>

<REAL:y>

<POINT:z>

and if *b* generates

<BOOLEAN:w>

<REAL:v>

then OR(a,b) generates

<INTEGER:x>

<BOOLEAN:u>

<POINT:z>

<REAL: OR(y,v)>

The meaning under the <REAL> contains a new OR-node which will manifest itself during the third pass. In this example, an OR-node in one pass gives rise to an OR-node in the next pass.

Efficient Treatment for Shared Derivations

We have just seen how ambiguous derivations can be tolerated when it is agreed that meanings generate parsing graphs. We will now consider a refinement of these conventions which will permit considerable computational savings. As mentioned in the section about the parser, the parser yields a derivation consisting of at most a number of nodes which is a polynomial function of the length of the input string. This relatively small number of nodes may none the less represent an exponential number of distinct derivations. This comes about because many subderivations are shared.

The EX operator takes no advantage of the fact that many subderivations may be shared. When a particular subderivation is referenced from two distinct points of view, EX applied from each point of view will cause the shared subderivation to generate its phrases twice. The computation incurred by EX is the same whether or not subderivations are shared. Thus, even if only a polynomial number of nodes represent an exponential number of derivations, EX will take an exponential amount of time.

A way to remedy this situation is to have each derivation node save its results, i.e., its generated phrases, so that all non-first accesses can simply fetch the previously computed values and hence avoid their recomputation. This guarantees that each node performs a computation only once and hence the time and memory taken to evaluate a derivation is proportional to the number of nodes making up the derivation. Thus, even though an exponential number of derivations might be represented, only polynomial space and time is needed to process all of the

represented derivations.

It is possible to have a derivation node store the results of its evaluation and to allow futher references to simply fetch the previously computed value *only if* the results are independent of the particular reference. That is, two distinct nodes which reference a given subderivation may set up different top-down contexts which will cause two evaluations of the given subderivation to yield different results. In such cases, the shared subderivation cannot simply yield the result of the first evaluation as the value for the second evaluation. Thus, the feasibility of sharing results of evaluations depends on conventions about top-down context.

It would appear that the variables LEFT and COLUMN are top-down context for the evaluation of derivations which generate parsing graphs. One evaluation of a given shared subderivation might occur where LEFT and COLUMN have one set of values and yet another evaluation of the same subderivation might occur with different values for LEFT and COLUMN. A way to resolve this dilemma is to imagine a representation for phrases which has the following two properties:

1) The representation is independent of the top-down context LEFT and COLUMN and

2) The representation may readily be converted to a value which incorporates the top-down context LEFT and COLUMN.

With such a representation, we can allow derivation nodes to store this imagined representation which is independent of top-down context. When a particular reference fetches this stored representation, it must convert a copy to incorporate the specific top-down context.

This technique of factoring out top-down context has worked with great success in other applications such as display graphics. The top-down context in display graphics is a transformation matrix. Many occurences of a given picture which differ only in orientation may be represented by a single instance of the picture where various references to the stored picture include individual transformation matrices. When a particular reference is made to the stored picture, the transformation matrix is applied to the picture in order to properly incorporate the top-down context.

The Semantic Operator PAW - Pruned Awakening

We will now consider a top-down context-free representation for parsing graphs. We will see both how easy it is to factor out the effects of the top-down variable COLUMN and yet how hard it is to do so for the variable LEFT. We will then consider a scheme of less generality where it is easy to factor out the effects of LEFT. We will also see how the loss of generality fits nicely with multipass language processing when one considers the problem of documenting a language.

To obtain a top-down context-free parsing graph from a meaning, the operator *PAW* sets both COLUMN and LEFT to NIL for the evaluation of the meaning. As such, the resulting parsing graph is certainly independent of the given values in COLUMN and LEFT. PAW then attaches the resulting parsing graph onto the given meaning. Thus, upon future references to the given meaning, PAW can simply pick up the previously computed value. A derivation node which includes the value of a previous evaluation is said to be *awake*.

Upon each call, PAW converts the stored top-down context-free parsing graph into one which incorporates the given values in COLUMN and LEFT. Since PAW is supposed to appear to act exactly as EX and because EX applied to a meaning is supposed to append new phrases onto COLUMN, PAW merely appends the stored parsing graph onto COLUMN. This properly incorporates the top-down context offered by COLUMN because in actuality, COLUMN is treated as an append-only variable and hence COLUMN in no way affects the generation of a parsing graph.

How might the variable LEFT be incorporated into one of these top-down context-free parsing graphs. One might suggest that a copy of the parsing graph be made where all LEFT fields which are found to be NIL be substituted with the value in the variable LEFT. Unfortunately, this technique does not yield the same parsing graph as would be yielded by actually evaluating the meaning with the given value in LEFT. For example, suppose LEFT references the parsing graph

<FORM:a>

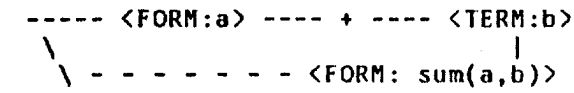
and suppose that evaluating a given meaning would generate the parsing graph

+ <TERM:b>

If LEFT is left unchanged for the evaluation of the given meaning, i.e., LEFT is not set to NIL, the phrase

<FORM:a> + <TERM:b>

will exist as <TERM> is generated and hence the grammar will add a new <FORM> node:



This exactly is what happens if EX is used instead of PAW. On the other hand, if LEFT is set to NIL for the evaluation of the given meaning, the evaluation will generate the phrase

+ <TERM:b>

in the absence of the neighboring <FORM:a> node and hence the grammar will not at this time add a <FORM:sum(a,b)> node. When PAW finally incorporates the variable LEFT, i.e., by changing the + node's LEFT field to reference the <FORM:b>, we indeed get the phrase

<FORM:a> + <TERM:b>

but we do not get a <FORM:sum(a,b)> node. The change to +'s LEFT is made too late; the <TERM> node's consideration by the grammar has already come and gone. Recall that a grammar is triggered upon the generation of the *rightmost* node in a matched phrase.

Thus, the incorporation of the top-down variable LEFT presents a problem. In order to alleviate the problem, we will consider some new conventions about generating phrases. Following are two observations. Referring to a previous example, even though *sum* might generate the parsing graph

<INTEGER:a> + <REAL:b>
<REAL:float(a)>
<REAL: addr (float(a) , b) >

the only phrase of real interest is the full spanning

<REAL: addr(float(a) , b) >

We are concerned only with the fact that the datum represented by *sum* is REAL. The fact that REAL came from this particular parsing graph is adequately represented in the meaning associated with the REAL. The second observation is that the non-first grammars in a multipass scheme can simply be reverse polish grammars. *Sum* can easily generate

<INTEGER:a> <REAL:b> +

instead of the infix phrase.

The new conventions are as follows:

1) PAW retains only full spanning phrases of length one.

2) The righthand phrase of each production in a non-first grammar must be either of length one or must have an operator as its rightmost part-of-speech.

In multipass processing, the first convention states that the relevance of a meaning is the generation of one part-of-speech and *not* a string of parts-of-speech. Looking at ICL's multipass scheme, this states that under a syntax part-of-speech, there must appear a well defined datatype and not a string of datatypes.

With this first convention, a language implemented by a multipass scheme can be simply documented by documenting each syntax production independently. Along with each syntax production, one can completely specify the relevant requirements imposed by the type-grammar solely in terms of a datatype relation which constrains the datatypes which may appear under each part-of-speech in the syntax production. This is

possible because only a well defined datatype, and not an abstract datatype phrase, will be associated with each syntax part-of-speech. The documentation need not mention the type-grammar nor the individual phrases generated by the meaning associated with each syntax production. Refer to the *ICL Reference Manual* for an abundance of this sort of documentation.

The second convention is necessary so that PAW can effectively generate phrases in isolation. Consider what would happen if *sum* were to use PAW in generating an infix phrase. *Sum* would first call on its left parameter to produce

<INTEGER:a>
<REAL: float(a)>

Sum would then generate a "+" to the right. Finally, *sum*'s righthand parameter would generate, in isolation, the phrase

<REAL:b>

When these phrases are put together, yielding

<INTEGER:a> + <REAL:b>
<REAL:float(a)>

the full spanning <REAL:addr(float(a),b)> will be missing. As in the previous example, the fact that <REAL:b> was generated in isolation means that the grammar never sees the phrase

<REAL:float(a)> + <REAL:b>

However, if *sum* generates a polish postfix phrase, *sum*'s two parameters may be evaluated in isolation and finally put together to yield

<INTEGER:a> <REAL:b>

<REAL:float(a)>

At this point, we suffer no loss in the fact that the grammar does not see this particular parsing graph as a whole. By convention (2), the grammar contains no productions which can match a phrase in this parsing graph except for productions whose righthand phrases have length one. However, these excepted productions have already applied during the individual generations of *sum*'s two parameters, e.g., the *float* rule has already generated the <REAL:float(a)>. When *sum* finally generates the + to the right, i.e., yielding

<INTEGER:a> <REAL:b> +

<REAL:float(a)>

the grammar's + production will fire, having access to the necessary phrases, and thus the grammar will indeed generate the desired <REAL:addr(float(a),b)> spanning the whole parsing graph.

It might appear that the conventions stated above remove so much generality from a non-first grammar that the grammar itself could be replaced by a set of functions. Since most productions will include a specific, well defined operator, it appears that each production could be replaced by a function whose name is the operator itself, e.g., the + production could be replaced by a + function which computes the possible resulting datatypes. However, the functions would have to take in not single datatypes, but lists of alternative datatypes. In addition, some functions might have to act as several productions, e.g., there are two + productions, one for integer and one for real arithmetic. In general, the functions would have to contain CASE and looping statements. It is

noteworthy that with the grammar implementation, the CASE and looping constructs are efficiently and generally handled by the parser's matching routine FIND, and in addition, the parser naturally yields lists of alternate parts-of-speech. Finally, a grammar implementation greatly surpasses a function implementation for two reasons: Some productions include no operators whatsoever, e.g., the *float* rule; they operate implicitly everywhere. Secondly, the grammar implementation facilitates a very modular definition, e.g., the two + productions may be expressed independently. This modular feature is extremely valuable in a compiler where the processing of declarations may spuriously add productions at different times.

Top-Down Context Besides LEFT and COLUMN - The Operator RESET

In addition to the variables LEFT and COLUMN, top-down context may be specified thru other variables. For example, a production which incorporates declarations for a local program block will modify the symbol table prior to the evaluation of the program block. In this example, the symbol table appears as top-down context for the evaluation of the program block. The operator RESET is provided for dealing with top-down context excluding LEFT and COLUMN.

RESET applied to a meaning removes all the stored results from previous evaluations. In this way, any record of previous top-down context is removed. This means, of course, that when PAW is applied to the reset meaning, all phrases will have to be regenerated. However, the recomputation will again be proportional to the number of nodes in the meaning because within the reset meaning, shared subderivations will

recompute only once.

Reluctant Derivations and Cycles - The Operator GOODNS

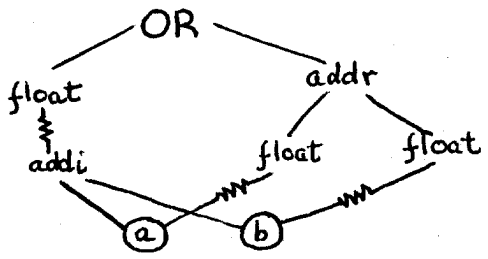
The reluctance associated with a production is stored in the derivation nodes produced by the production. For example, the phrase

<INTEGER:a> + <INTEGER:b>

may yield the ambiguous derivation

OR(float(addi(a,b)) , addr(float(a),float(b)))

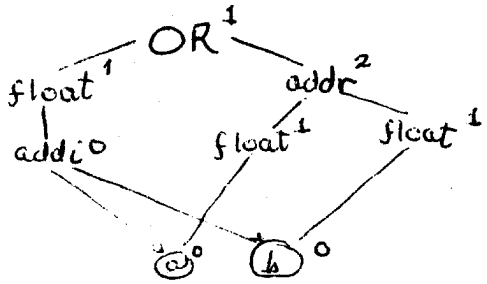
when viewed as a REAL. This derivation can be drawn as



where the resistor symbol represents the application of a reluctant production, e.g., the *float* production. From the point of view of the OR-node on the top, the lefthand alternative contains one resistor whereas the righthand alternative contains two resistors.

The operator GOODNS (short for *goodness*) climbs thru a derivation and removes subderivations of higher resistance. More specifically, GOODNS associates a number with each node in a derivation to record the number of resistors contained in the total subderivation whose top is the given node. GOODNS associates with an OR-node the minimum resistance of its two alternative subderivations. In the example given

above, the resistance numbers are



As GOODNS associates a number with an OR-node, if the numbers associated with the two alternative subderivations differ, GOODNS replaces the OR-node with a NO-OP node which references only the minimal subderivation. In this way, reluctance is manifested in a given derivation. The only OR-nodes which survive are those which reference subderivations of equal resistance.

It is intended that GOODNS will be applied to a derivation before any other semantic operator is applied.

There is another situation which GOODNS must handle. Derivations yielded by the parser may be cyclic; the parser does make destructive modifications when installing an OR-node. For example, consider the rules

<REAL: float(a)> ::= <INTEGER:a>

<INTEGER: fix(a)> ::= <REAL:a>

One can imagine that <INTEGER:a> will parse as

<INTEGER:a>

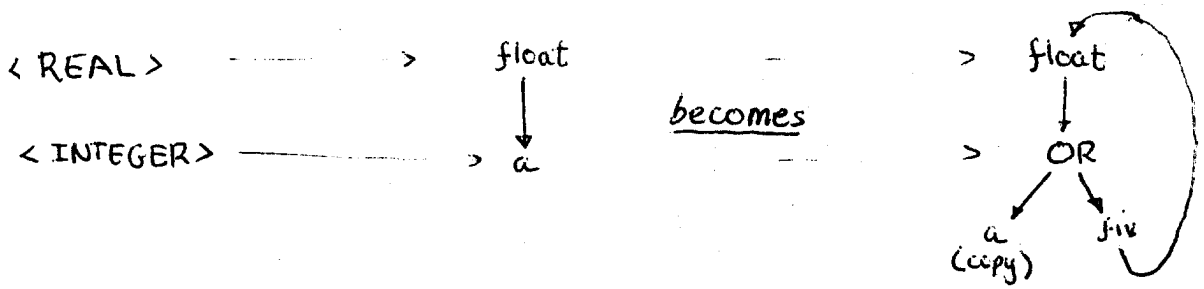
<REAL: float(a)>

<INTEGER: fix(float(a))>

<REAL: float(fix(float(a)))>

. . .

However, when the parser proposes the second <INTEGER> node, NEWNODE sees that there already exists an <INTEGER> node having the same span, and therefore NEWNODE does not generate a second <INTEGER> node, rather, NEWNODE modifies the derivation associated with the first <INTEGER> node. NEWNODE modifies the derivation node *a*, *in place*, so that in fact, the node at *a* is now the OR-node and one of the nodes referenced from the OR-node is a copy of *a*. From the points of view of the <INTEGER> and <REAL> nodes, what used to be



after NEWNODE incorporates the proposal of the second <INTEGER> node.

This cyclic derivation does indeed capture the infinite number of derivations implied by the two given productions. For example, from the point of view of the <INTEGER> node, the viewer has a choice of taking simply *a*, or, taking one ride around the cycle, *fix(float(a))*, or taking two rides around the cycle, *fix(float(fix(float(a))))*, etc. From the point of view of the <REAL> node, the same choices are available where each choice is embedded within one *float*.

The following arbitrary decision concerning cycles is implemented in GOODNS: At a given OR-node, GOODNS removes an alternative subderivation if that subderivation has no choice but to refer back to the given OR-node. In essence, GOODNS climbs down a derivation and upon encountering a previously encounted node which has been entered but not yet left, GOODNS associates infinity as the resistance number. Thus, at the closest OR-nodes which access this previously encountered node, GOODNS will naturally remove the cyclic alternative. Here is a rationale: Assuming that at least one of the productions involved in the cycle is reluctant, it would appear that travelling around the cycle even once will collect more resistance than avoiding the cycle altogether.

This algorithm is indeed arbitrary because it may depend on the order in which GOODNS climbs down thru a given derivation. However, the algorithm does remove all cycles and does so by removing only OR-nodes. In addition, the clipping occurs in some sense as close as possible to the re-encountered node.

ICL OVERVIEW

ICL is a general programming language implemented on the language processor presented in this thesis. Refer to the large appendix *The ICL Reference Manual* for a formal and complete definition of ICL. Refer to the appendix about the MACRO-10 implementation for a description of the files which make up ICL.

ICL is implemented in a multipass scheme as described in the section about languages. ICL consists of three passes. The first pass constrains syntax, the second pass constrains datatypes, and the third pass constrains the use of data *sources* and data *sinks*. The third pass ensures that only data *sinks* may appear on the lefthand sides of assignment statements. Furthermore, the third pass deals with a special kind of sink, a *looping-target*, which facilitates a uniform treatment for ICL's main loop-generator, the *selection FOR-quantifier*.

ICL was created with several goals in mind. The first goal was to show that the general language processor is indeed a very practical tool for implementing languages. The second goal was to provide a convenient language to aid in the design of integrated circuits. A third goal was to produce a language which includes constructs absent from other programming languages which have none the less proven indispensable in the field of language processing. A fourth goal was to incorporate into a conventional programming language as much of the extensive flexibility and generality offered by the rewrite parser as possible. Another goal was to remove the consideration of pointers from the user's domain; too much confusion arises from ad hoc use of pointers. Finally, ICL was to

be very modular in both implementation and documentation, and as such, ICL should be readily extensible.

Following are the results of each goal. This section concludes with ICL's compile time error reporting mechanisms.

Modularity

As the reader may note, the ICL reference manual documents ICL in terms of small groups of productions. Each group is independent of all the others. The ICL reference manual, like the PDP-10's reference manual, is both complete and hard to learn from. The ICL manual is a very straightforward translation from the implementation into English with additional comments concerning the relevance of each construct.

Because ICL is implemented on top of the language processor, one can easily extend ICL by adding more productions to any of the three grammars, or passes. One need not worry about interactions among productions as long as one follows the conventions for meanings under the various parts-of-speech.

The Use of the Language Processor

As just mentioned, one of the profits gained by using the language processor is modularity and extensibility. The majority of routines which make up the ICL compiler are each less than a half a page of nearly double spaced MACRO-10 source. The vast majority of routines correspond one-to-one with the productions in ICL's three grammars. The notable exceptions are the routine which processes declarations and the

routine which coordinates the three passes.

The use of the language processor facilitated easy modification during the creation of ICL. I didn't need to worry about syntax or datatype processing; I merely dealt with individual productions, adding, removing, and modifying each one on an independent basis.

I did go to some trouble to optimize the syntax grammar for speed. Some simple productions are broken into several smaller productions to facilitate a linear rather than quadratic parsing time. For example, strings (arrays) in ICL are created with the notation

{ element ; element ; element ; ... ; element }

A simple and straightforward syntax description is

```
<STRING_EXPR>  ::=  <EXPR>
<STRING_EXPR>  ::=  <EXPR> ; <STRING_EXPR>
<EXPR>         ::=  { <STRING_EXPR> }
```

With the parser presented in this thesis, this grammar will take quadratic time as a function of the length of a string expression; a <STRING_EXPR> will be found to span every substring, e.g.,

{ element ; element ; element ; element }

(--STRING_EXPR----)

On the other hand, I could take advantage of the fact that a <STRING_EXPR> is useful only between the brackets *{* and *}*. I constrained the creation of <STRING_EXPR>s to occure only in the context of a *}* with the grammar

```
<STRING_EXPR>  ::=  <EXPR> }
<STRING_EXPR>  ::=  <EXPR> ; <STRING_EXPR>
<EXPR>         ::=  { <STRING_EXPR>
```

This grammar takes only linear time to parse a string, e.g.,

```
{  element ; element ; element ;    element  }
                                  (-STRING_EXPR-)
                            (------ STRING_EXPR ----)
                      (------------ STRING_EXPR --------)
                (---------------- STRING_EXPR --------------)
           (------------------ EXPR ---------------------)
```

With this grammar, a <STRING_EXPR> is created only in the context of a } and hence substrings not including the last element will not parse as <STRING_EXPR>.

Because the language processor preserves locality of ambiguity, I was able to maintain a modularity in the syntax grammar, even at the expense of making the syntax grammar ambiguous. For example, there are two distinct applications of ICL's CASE statement; one is for SCALAR datatypes and the other is for VARIANT datatypes. The syntax production for the SCALAR-CASE is

```
<EXPR>  ::=  CASE <EXPR> OF   ...
```

and the syntax for the VARIANT-CASE is

```
<EXPR>  ::=  CASE <ID> OF   ...
```

The CASE value in the VARIANT form is constrained to be a single variable (<ID>) for semantic reasons. Rather than including only the former production and implementing the latter with a semantic

restriction, I simply included both productions. The meaning transformations under each production are independent and each considers only one of the VARIANT and SCALAR meanings.

Because ICL also includes the production

<EXPR> ::= <ID>

the form

CASE <ID> OF ...

will parse two ways, one for the SCALAR interpretation and one for the VARIANT interpretation. However, thru all three passes, this ambiguity is manifested only in the locality of the CASE construct. In fact, in the datatype pass when the <ID>'s datatype is known, the ambiguity will cease to exist.

ICL was started in June 1976, in both conception, design, and implementation. These three efforts occured in parallel with little trouble. Within one year, by June 1977, ICL was working, nearly free of bugs. Until now, April 1978, fewer than ten bugs have been found (and fixed), the last being resolved over three months ago. Since June 1977, ICL has been under extensive use by myself, and more recently, there have been several other users designing ICs.

ICL aimed at IC Masks

ICL includes three main features important for dealing with IC masks. First, the notation for creating two dimensional points is brief; there is a one character overhead. Operators like + and - are defined for points as well as for integers and reals. Secondly, the

selection-FOR quantifier provides for convenient access to polygons represented as strings of points. Several vertices may be taken at a time with the option for wrapping around back to the beginning of the string. Finally, automatic data sharing facilitates both safe and efficient representations for IC-masks, objects of a highly repetetive nature. Refer to a following section about pointers for more about data sharing.

Carryovers from Language Processing

ICL provides for the creation and invocation of processes as previously described in the section *Meaning as Programs* within the section about languages. Indeed, the //...\\ construct exists in ICL. This construct allows for programs along with specific context to be passed off as data and hence to be stored in datastructures as readily as any other kind of data. In fact, the //...\\ construct goes beyond that which has yet been described. In ICL, parameters may be passed to a process invocation just as they are passed to a function call. In addition, the user may specify that a process be allowed to change the values of its context variables so that later invocations can have a private memory of previous invocations. For numerous examples, please refer to the section on *processes* near the very end of the ***ICL Reference Manual***.

Ambiguity - A Manifestation of the Parser

The existence of the parser with its tolerance of ambiguities has made simple the implementation of both coercions and polymorphic function names. In addition, the parser offers a totally general and restriction-free implementation for datatypes. In essence, if the user can imagine a way by which his program will make sense in the space of datatypes, the parser will find it and implement it.

In fact, with the parser's upper bound for expense, it is guaranteed that any set of coercions and functions will be accepted and processed in finite time. Coercions may be defined between datatype without concern for cycles, e.g., the INTEGER-to-REAL and a REAL-to-INTEGER coercion may both exist simultaneously.

At early stages in ICL's developement, I considered making ICL's syntax dynamically extensible. Such a feature is nearly trivial to implement. However, syntactic extensibility has the disadvantage that programs written by different people might not be easily readable by others in the user's group. Besides, a reliance on syntax extension can easily divert people's attention from the more relevant, semantic issues involved in a given programming task.

ICL's extensibility is more of a semantic sort. The second grammar, the datatype grammar, is completely extensible via the use of type, coercion, and function declarations. Extensibility limited to the datatype grammar conforms to the kind of extensibility offered by a conventional programming language like PASCAL.

Following are examples of the various ways in which ambiguity crops up in datatypes.

Two representations for geometric lines are defined in ICL by

```
TYPE  SEGMENT =  [FROM:POINT  TO:POINT] ;
      EQUATION=  [A:REAL  B:REAL  C:REAL] ;
```

A SEGMENT consists of two points labled FROM and TO. An EQUATION consists of three numbers labled A, B, and C which define the line equation

$$Ax + By + C = 0$$

Suppose we provide coercions between the two representations, i.e., we declare

```
LET  SEGMENT  BECOME  EQUATION  BY  some program
LET  EQUATION  BECOME  SEGMENT  BY  some program
```

These two coercions make the types SEGMENT and EQUATION interchangeable, i.e., any SEGMENT may be viewed as an EQUATION and visa versa. Finally, suppose we define a routine for intersection:

```
DEFINE  INTERSECT( A:EQUATION  B:EQUATION ) = POINT: ...
```

INTERSECT takes in two EQUATIONs and yields a POINT. The two coercions and one function declarations affect the type-grammar by adding the rules

```
<EQUATION>  ::=  <SEGMENT>
<SEGMENT>   ::=  <EQUATION>
<POINT>     ::=  INTERSECT ( <EQUATION> , <EQUATION> )
```

The third rule is really entered in a reverse polish form, but we will ignore that fact for clarity. Now, if the user writes

P := INTERSECT(A,B);

where P is a POINT and where A and B are EQUATIONs, the type-pass will generate the parsing graph (again ignoring polish conventions)

POINT := INTERSECT (*EQUATION* , *EQUATION*) ;

(--------------- *POINT* ---------------) .

This assignment statement is legal because both sides of the assignment can be viewed as the same type of object, namely *POINT*. If A were a SEGMENT instead of an EQUATION, this assignment would parse as

POINT := INTERSECT (*SEGMENT* , *EQUATION*) ;

(-*EQUATION*-)

(---------- *POINT* ----------) .

The *SEGMENT-to-EQUATION* coercion is employed to maintain datatype consistency. In fact, each parameter in INTERSECT may independently be either of type SEGMENT or of type EQUATION. Each parameter which is not of type EQUATION will invoke one coercion.

An optimization is obtained by defining another INTERSECT especially for SEGMENTs, e.g.,

DEFINE INTERSECT(A:SEGMENT B:SEGMENT) = POINT: ... .

Consider that if the parameters to INTERSECT are each of type SEGMENT, the given assignment statement will parse either as

POINT := INTERSECT (*SEGMENT* , *SEGMENT*) ;

(--------------- *POINT* ---------------)

or as

```
POINT := INTERSECT ( SEGMENT , SEGMENT ) ;
                     (-EQUATION-) (-EQUATION-)
          (---------- POINT ----------)
```

This ambiguity reflects the fact that either INTERSECT routine can be employed. However, because productions entered by coercion declarations are entered as reluctant productions, the first parsing graph will dominate. Hence, no coercions will apply and the INTERSECT routine which directly deals with SEGMENTs will be chosen.

If the user defines all four INTERSECT routines, one for each possible type combination, no coercions need ever apply and hence the user has achieved an optimization. Note the flexibility offered here: The user is allowed to define anywhere from one to four different INTERSECT routines and in any case, the user's program will work. Without changing any program text, e.g., programs refering to INTERSECT, definitions for INTERSECT may be added or removed with the effect of varying only optimization and not correctness.

Another example of ambiguity arises in the following program. Suppose the user declares

```
TYPE  GQS = EITHER
              JUST_ONE = QS
              MANY     = { GQS }
            ENDOR ;
```

where QS denotes the type for quoted text strings. This declaration states that a GQS may be formed in one of two ways:

1) Any QS is a GQS *and*

2) Any *string* of GQSs is itself a GQS.

The notation { GQS } denotes a datatype which represents a *string* (array) of GQSs. This declaration essentially generates the rules

```
<GQS>  ::=  <QS>
<GQS>  ::=  { <GQS> ; <GQS> ; ... }
```

Thus, for example, the QS

```
'Hi'
```

is a GQS. The string of GQSs

```
{ 'Hi' ; 'There' ; 'You' }
```

is a GQS. In fact, the nested expression

```
{  { 'Hi' ; 'There' }  ;  { 'You' }  }
```

is a GQS. Now, suppose the user defines the type MESSAGES as follows:

```
TYPE  MESSAGES =  { GQS } ;
```

The expression

```
{   { 'Hi' ; 'There' }  ;  'You'   }
```

may be viewed either as a single GQS or as a MESSAGES. If viewed as a MESSAGES, this expression represents a string of length two whose elements are the GQSs

```
{ 'Hi' ; 'There' }    and    'You'
```

Thus, if the user declares the function

```
DEFINE  PROCESS( M:MESSAGES ):  ...
```

and if the user subsequently specifies

```
PROCESS(   {  { 'Hi' ; 'There' }  ;  'You'  }   );
```

the { { 'Hi' ; 'There' } ; 'You' } will be viewed as a MESSAGES and not a GQS so to be compatible with PROCESS.

Another example involves a datatype called RG which is meant to represent pictures. We wish that an RG be formed by any of the following

1) Any POLYGON is an RG

2) Any *union* of RGs is an RG, and

3) Any *displacement* upon an RG is an RG.

We can declare RG with

```
TYPE  RG =  EITHER
              SIMPLE =   POLYGON
              UNION  =   { RG }
              DISP   =   [DISPLACE:RG  BY:POINT]
            ENDOR ;
```

This declaration for RG essentially adds the rules

```
<RG>  ::=  <POLYGON>
<RG>  ::=  { <RG> ; <RG> ; ... }
<RG>  ::=  [ DISPLACE: <RG>   BY: <POINT> ]
```

Thus, if CURLY is an instance of type POLYGON, then

```
{ CURLY ; [DISPLACE: CURLY BY: point] }
```

represents two CURLYs, one of which is displaced by *point*. This particular expression is an RG via the parsing

```
{ CURLY ; [DISPLACE: CURLY BY: point] }
 (-POLYGON-)          (-POLYGON-)
 (---RG----)          (---RG----)
            (---------- RG ----------)
(-------------------- RG ----------------------)
```

Now, suppose the user wishes to associate a minimum bounding box (*mbb*) with each subpicture. For sure, he doesn't want to specify the *mbb* each time he specifies a subpicture; the user likes the current notation for specifying RGs. We can get *mbb*s installed automatically and implicitly by making the following declarations. First of all, we make up a new datatype called MRG which will represent an RG along with its *mbb*:

```
TYPE MRG = [BODY:RG MBB:BOX] ;
```

Even though we wish to specify pictures as RG's, we would like to access an RG as though it were an MRG. The coercion

```
LET RG BECOME MRG BY [BODY:RG MBB: f(RG)] ;
```

specifies that any RG may be viewed as an MRG. We assume, of course, that *f* maps an RG to its *mbb*. This coercion adds the rule

```
<MRG> ::= <RG>
```

Let us redeclare the type RG so that each reference to an RG is replaced by a reference to an MRG:

```
TYPE  RG =   EITHER
                SIMPLE =   POLYGON
                UNION  =   { MRG }
                DISP   =   [DISPLACE:MRG  BY:POINT]
             ENDOR ;
```

This new definition for RG guarantees that each subpicture will include its *mbb*, i.e., each subpicture in a *union* will include its *mbb* and each subpicture involved in a displacement will include its *mbb*. Note that any given expression which could be viewed as an RG under the old definition will still be viewable as an RG under the new definition because any subpicture, an RG, will automatically coerce to an MRG. For example, we will get the parsing

```
{  CURLY  ;  [DISPLACE:  CURLY  BY: point]  }
 (-POLYGON-)            (-POLYGON-)
 (---RG----)            (---RG----)
 (---MRG---)            (---MRG---)
            (---------- RG ----------)
            (---------- MRG ---------)
(---------- RG ------------------------)
(---------- MRG -----------------------)
```

Each place where an RG is rewritten to an MRG, code will be generated which will calculate the *mbb* and thus create a valid MRG.

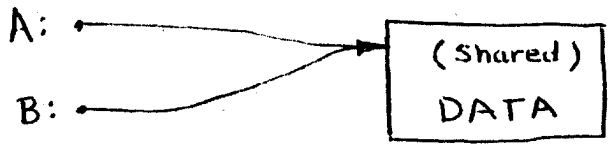
ICL's tolerance of ambiguity in datatypes very often makes it possible to modify declarations without having to modify the executable part of a program. Modifications to declarations can be made for optimization as well as for changes in concept. Coercions generally

come in handy to cover up a splitting of what used to be one datatype, e.g., RG, into several types, e.g., RG and MRG. More examples are contained in the *ICL Reference Manual*.

Pointers

In order to provide a safe and flexible system for serious use, pointers had to be used in the implementation but also had to be invisible for the majority of programs. For example, the languages PASCAL and SIMULA require an obsessive and inconvenient awareness of pointers. In these languages, the user must explicitly distinguish between a pointer to an object and the referenced object itself in both declarations and operations. The explicit use of pointers is required even for the very common purpose of defining recursive datastructures. Even worse, many subtle bugs arise with pointers from an inadvertent sharing of data, e.g., a modification to a datastructure may become apparent from unwanted points of view. The programmer is forced to do his own bookkeeping with respect to specifying copy operations in order to avoid unwanted data sharing. Pointers, like GOTOs, will often obscure simple constructs, and even worse than GOTOs, pointers may be abundantly created by the execution of programs.

In principle, the need to be aware of pointers is a rarity. Pointers are necessary in concept only when one wishes to share data for the single purpose of allowing modifications to the data to be simultaneously apparent from several points of view. Except for this purpose, it is conceptually easiest to imagine that no datum is shared and that pointers do not exist. For example, the data structure



may be thought of as being equivalent to

unless we wish a modification to the shared data made from the points-of-view of either A or B be apparent from both A and B. In practice, however, a programmer will often share data for efficiency even though he does not wish that modifications be apparent from all references to the shared data.

ICL does not require the programmer to be aware of pointers except in programming tasks where it is in principle necessary to be aware of pointers. That is, the ICL programmer may define and use recursive datastructures or do anything he wishes without having to know about pointers and data sharing. However, if the user wishes to implement shared data for the purpose of having modifications be apparent from several points of view, the user must obviously think in terms of pointers; hence ICL has provided a single operator, the @ operator, which allows the user to make modifications which will expose pointer structure. The @ operator corresponds to a combination of LISP's RPLACA and RPLACD operators.

Backstage, ICL does indeed make extensive use of pointers whether or not the user wishes to be aware of it. ICL automatically shares data as much as is possible without any overhead. Data transfers, e.g., assignment statements and parameter passage, are each implemented by transferring a one word entity which is often a pointer. Datastructures whose creations are specified with multiple references to a particular variable automatically wind up sharing the structure referenced by the variable.

However, ICL will never destructively modify an existing structure except via the @ operator. Excluding the @ operator, when the user specifies a modification, the modification will be carried out in such a way that the modification will become apparent only to the variable with which the user specifies the subject structure. ICL copies a minimal amount of the subject structure, just enough to implement the modification, and finally assigns this augmented structure to the variable. It will appear as though the variable has always referenced a private copy of the datastructure. If no variable or structure references the original structure, those parts from which copies were made will automatically be returned to free storage during the next garbage collection. If in fact some variable or structure does reference the original structure, both the modified and the original structure will exist sharing all that substructure which was not involved in the minimal copy. This "copy on write" technique allows data to be shared invisibly. Digital Equipment Corporation uses the same technique on the coarser scale of memory pages.

Refer to the section *ICL's Policy About Assignments, Copying, and Pointers* in the *ICL Reference Manual*. That section contains both examples and implementation details.

Error Reporting In ICL

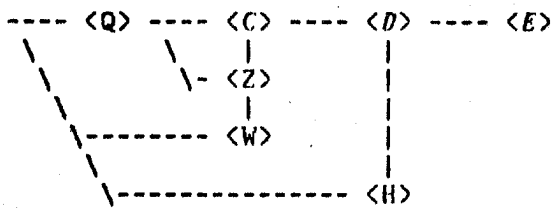
The reader may refer to the *ICL Reference Manual* to see how compile time user errors are reported. Basically, each pass has its own ways of reporting errors. What follows are the techniques used in generating the error messages. The mechanism by which errors are reported is in fact supplied with the language processor and not with ICL.

Syntax Errors

Some of the productions which make up ICL's syntax are *deterministic*. A deterministic production is one which destroys alternate phrases during the generation of its lefthand phrase. A deterministic production removes any phrase whose span intersects the span of the generated lefthand phrase. For example, the deterministic production

<A> <B> ::= <C> <D> <E>

will apply in the parsing graph



leaving the following parsing graph

---- <Q> ---- <A> ---- <B>

Any phrase which (partially) spanned the <C>-<D>-<E> phrase has been removed.

Any production can be made deterministic by modifying the generation of its lefthand phrase. As the reader may recall, when a lefthand phrase is about to be generated, the global variable COLUMN contains all alternate phrases which share the same righthand edge. Basically, a deterministic production sets COLUMN to NIL before it generates its lefthand phrase. In the example given above, the deterministic production is entered when COLUMN references the <E> node. The deterministic production removes the <E> node from COLUMN and hence kills any reference to the parsing graph accessible from <E>. However, when the deterministic production places the phrase <A><B> onto COLUMN, only the <Q> node from the original parsing graph remains accessible.

To support the application of several deterministic productions over a given span, a deterministic production actually sets COLUMN not to NIL, but to the contents of another variable which is initialized to NIL. After the production's lefthand phrase is generated, the production stores COLUMN back into this other variable. Thus, if another deterministic production applies, the phrases generated by the earlier applications of deterministic productions will not be lost; COLUMN will be set not to NIL, but to the previously generated phrases. In some sense, the lefthand phrases of deterministic productions form an elite set.

A deterministic production represents a certainty that the application of the production is unconditionally appropriate. The righthand phrase of a deterministic production should contain an abundance of unique keywords so that only the existence of such a keyword in a user's text can trigger the deterministic production.

Syntax errors are reported by presenting the user with a linearized version of the parsing graph. This linearized parsing graph is obtained by scanning the parsing graph from right to left, arbitrarily choosing the first node in each encountered column. This indeed presents a linearized parsing graph devoid of alternate parses. It so happens that the first node in most columns is an input character. This is so because columns are ordered by part-of-speech and input characters have the lowest parts-of-speech. (Refer to the appendix on *MACRO-10 Implementation*). Thus, except where deterministic productions have applied, the user will be presented with his original input text. Where deterministic productions have applied, the original input text will be replaced by a syntax part-of-speech. Thus, the linearized parsing graph indicates where certain productions have applied.

The usefulness of this sort of error message increases with the number of deterministic productions. However, with a greater number of deterministic productions, a modification to ICL's syntax requires a greater amount of care. If a modification isn't made carefully, the application of a deterministic production might wind up removing phrases which are necessary for a successful parsing.

At the present time, ICL includes few deterministic productions. The last index in the *ICL Reference Manual* lists ICL's deterministic productions.

This scheme is not well understood and hence it should either be replaced or better understood. A better scheme might have deterministic productions simply remove one element from COLUMN, namely a terminal part-of-speech, without destroying the other alternate phrases. Yet another method might have deterministic productions remove no phrases whatsoever and simply mark the generated lefthand phrase so that the error reporter can still see the "deterministic" application.

Datatype and PASS3 Errors

Each non-first pass, or process step, is precisely the evaluation of the derivation yielded by the previous pass. The evaluation of the derivation yielded by the previous pass generates phrases in the language of the current pass. The result of the current pass is the parsing graph, i.e., phrases, generated by the top node in the derivation.

Let us assume that the operator PAW is used to evaluate a derivation. PAW stores with each node in the derivation the parsing graph generated by that particular node. If a particular node generates the empty parsing graph, we will say that the node has no parsing graph. An error in a non-first pass is *detected* by noticing the lack of a parsing graph associated with the top node in the derivation.

For example, consider the derivation node created by the syntax production for "+":

<EXPR: //[a;b;] *f*(a,b)\\ > ::= <EXPR:a> + <EXPR:b>

The resulting derivation node references *f* along with the parameters *a* and *b*. Suppose that *f* generates a parsing graph which depends on the parsing graphs generated by *a* and *b*, e.g.,

EX(*a*) "+" EX(*b*) .

If EX(*a*) generates <INTEGER> and if EX(*b*) generates <BOOLEAN>, then *f* generates

<INTEGER> + <BOOLEAN> .

This phrase matches no production of the second grammar, i.e., INTEGERs and BOOLEANs cannot be combined with "+". Thus, the grammar will generate no full spanning phrase of length one over the given phrase. The parsing graph which PAW stores with the derivation node representing *f* is a pruned parsing graph; only full spanning phrases of length one are kept. Thus, PAW will store no parsing graph with the derivation node representing *f*.

An *erroneous* derivation node is any derivation node with the following properties:

1) It has no parsing graph *and*

2) Each of its sons, e.g., *a* and *b* in the example given above, does have a parsing graph.

The first property indicates that the derivation has no interpretation in the current pass. The second property indicates that the lack of interpretation is not the fault of a subderivation.

Each node in a derivation can be identified with the production which generated the node. In ICL, a production is identified by the name of the function which implements its meaning. For example, the "+" production given above is identified by the name *f*. In the *ICL Reference Manual*, the documentation for each syntax production includes the name of the routine which implements the production's meaning. This name is presented as the *name* of the production.

An error in a non-first pass is reported by

FOR each *erroneous* node in the derivation,

1) Identify that node for the user and
2) Provide a *backtrace* so that the user can see where in his program the error occurs.

A *backtrace* consists of the sequence of derivation nodes lying between the erroneous node and the top derivation node. Refer to the section *Datatype Errors* in the *ICL Reference Manual* for a convenient way to interpret a backtrace.

This error reporting scheme will identify an error in the second pass by identifying particular syntax productions. Because each syntax production is documented in the *ICL Reference Manual*, a user can successfully interpret the error message generated from a datatype error. However, an error in the third pass is reported by identifying productions belonging to the second pass. Because the productions belonging to the second pass are *not* documented, the user can make little sense of a third pass error message. As mentioned in the *ICL Reference Manual*, errors from the third pass rarely occur, and for the most part, they are the result of very obvious user errors.

None the less, it would be nice to report errors from the third pass in terms of syntax productions rather than in terms of datatype productions. The following scheme might achieve this: Associate with each third-pass derivation node a pointer to the second-pass derivation node which generated those phrases responsible for the creation of this third-pass derivation node. Given such a pointer, an *erroneous* third-pass derivation node can be reported in terms of a second-pass derivation node.

The creation of such pointers can be implemented by setting a global variable which references a given second-pass derivation node *during* the evaluation of the given second-pass derivation node. As the second-pass derivation node generates phrases for the third pass, each derivation node created by a third-pass production can be augmented to include the value currently in the global variable. Thus, each third-pass derivation node which is created by the evaluation of a given second-pass derivation node will reference the given second-pass derivation node. This is currently not done in ICL.

CONCLUSION

This thesis has presented a language processor and a sample language implemented with this language processor. The language processor supports ambiguity so that the specification for a language can be extremely modular. The language processor practically supports ambiguity by representing and processing only essential differences among multiple interpretations.

The sample language, ICL, is a rich, general-purpose programming language which takes special advantage of the language processor in support of user-defined datatype coercions and polymorphic operators. Both the language processor and ICL work reliably.

Several systems have already been implemented in ICL including an IC-mask processor, a graphics system which includes arbitrary, non-linear transformations, a text preprocessor for the program RUNOFF, and a graphics text processor which includes fonts and colors. Numerous IC-masks have been made with ICL and one user has defined a function which yields a PLA as a function of the number of AND terms, the number of OR terms, and the binary PLA code. This thesis itself was run through the text preprocessor for subsequent processing by RUNOFF.

ICL's datatype checking has repeatedly facilitated quick and sure creation or modification of programs. Many bugs are found and pinpointed immediately at compile time. A rich use of datatype coercions and polymorphic functions not only offers a convenient technique for program specification, but it also facilitates quick changes in datatype definitions. Without re-examining programs which

utilize the modified datatypes, a quick recompilation has often run without any problems; ICL chooses a different placement of coercions so that the program specification remains consistent with the new datatype definitions. If no placement of coercions can render consistency, then ICL cites exactly those places in the program specification which present no possible interpretation. Often by introducing a new coercion or a new definition for an existing function name, these remaining problems quietly cease to exist.

Although ICL generates rather sloppy machine code, e.g., there is no attempt to optimize the machine code *per se* and all temporaries are stored not in registers but on the stack, ICL code has run three times as fast as SIMULA compiled code. This relation was obtained by running a program which adds 10000 points in each of ICL and SIMULA. Because the type POINT is primitive in ICL and not in SIMULA, I reran the ICL program with the type POINT substituted with a user-defined record datatype which represents two REALs. Still, ICL ran 2.5 times as fast as the SIMULA program. I imagine that the difference between SIMULA's and ICL's runtimes rests on the fact that SIMULA leaves some datatype considerations for runtime, e.g., superclass searching, whereas ICL processes all datatype considerations, e.g., datatype coercions, at compile time.

One of the currently largest ICL systems includes the IC-mask processor, the general graphics system, and the graphics text processor. This system resides in 86.5K words of memory. This figure includes all of the ICL compiler including the symbol table and the three grammars. This system includes 325 user-defined functions. The 86.5K memory

includes 30K for list-space, 12K of which is free, 17K of machine code for the user-defined functions, and 7.9K for the datatype grammar.

The language processor's preservation of locality of ambiguity pays off not only in theory, but also in practice. To see the effect of this feature, I chose a one line ICL statement which would generate a maximal amount of syntactic ambiguity:

```
WRITE( 1 \A 1 \A 1 \A 1 \A 1 );
```

The \A is an infix notation for calling the function named A. ICL will consider all possible ways to apply parentheses around subexpressions. ICL will finally choose that placement of parentheses which tends to group from left to right while satisfying the datatype requirements imposed by the infix function calls. In this example, A was defined to map two integers to one integer, and thus ICL would ultimately choose the strictly left to right grouping. Theory says that an expression of this form which has *n* \As will give rise to at least an exponential number of groupings. However, theory also says that this language processor will process the exponential number of meanings in polynomial time.

With the standard ICL compiler, this statement with *6* \As compiles in about one second. With *15* \As, it takes about 16 seconds. To make ICL ignore locality of ambiguity during the second and third passes, I modified the semantic operator PAW so that it would *not* take advantage of shared subderivations. With this modification, ICL took *44* seconds to process a statement with *5* \As, and with *6* \As, it took a minute and a half before ICL fatally ran out of memory. It is impossible to make the first pass ignore locality of ambiguity without modifying the parser

itself. I have modified the parser's routine NEWNODE so that it would *not* collapse identical parsing graph nodes. This, however, led to infinite looping and memory consumption because of the existence of cyclic rules like

<REAL> ::= <INTEGER> *and*
<INTEGER> ::= <REAL> .

ICL compiles and executes an assignment statement which assigns a newly created box with a color to a variable whose type admits IC-masks. Compiled and executed one at a time, ICL processes about 300 of these assignment statements per minute.

For future work, this language processor needs a meta-language besides MACRO-10. Currently, all productions are expressed in MACRO-10 with the help of macros, as described in the first appendix. MACRO-10 was chosen as the meta-language because to specify semantics, it is often necessary to specify programs which implement meanings. However, now that ICL is working, it should be relatively easy to augment ICL to include new datatypes and syntax for specifying grammars. All semantics can be conveniently expressed in ICL.

The main problem with using MACRO-10 as the meta-language is that each change to a grammar requires a reassembly. This restriction forbids runtime creation of grammars. Another disadvantage follows immediately from the fact that MACRO-10 provides no type checking. Bugs in MACRO-10 programs can be much harder to find than bugs in ICL programs. ICL always generates machine code which obeys the conventions imposed by system components such as the garbage collector. A single

violation of system conventions can result in obscure behavior, e.g., illegal memory references. The bug will become apparent much too late, e.g., during a subsequent garbage collection.

Currently, I imagine that the parts-of-speech for a new grammar should be declared statically like variables and datatypes. A part-of-speech declaration might look like

POS *FORM = INT* ;

This would declare that FORM is a new part-of-speech and that INT will serve as the datatype for any meaning which can be associated with the part-of-speech FORM. In general, the declaration for each part-of-speech should include a datatype which will serve as the datatype for any meaning which can be associated with the new part-of-speech. With this information, ICL can verify that all meaning transformations preserve datatype integrity. The section *Languages* shows why it is absolutely necessary to associate a datatype with each part-of-speech.

A replacement rule can be specified with a notation like

RULE <FORM: *EXPR*> ::=
<FORM: *variable*> '+' <TERM: *variable*>.

An expression of this form can be thought of as an instance of a new primitive datatype called *RULE*. A *grammar* can be defined to be a *string* of *RULE*s. ICL can compile the meaning for the lefthand <FORM> by compiling the specified <EXPR> in the context where each of the variables specified in the righthand phrase becomes an implicit parameter to the <EXPR>. The type for each of these parameter variables

is known immediately from the part-of-speech declarations. For example, looking at the declaration given above for the part-of-speech FORM, we can tell that the *variable* specified in <FORM:*variable*> should be given the type INT. Finally, ICL can verify that the <EXPR> associated with the lefthand <FORM> is of type INT, the type associated with the part-of-speech FORM.

However, to provide the flexibilty offered by the MACRO-10 meta-language, it will be necessary to support more than *RULE*s and part-of-speech declarations. For multipass specification, the meta-language must include a notation for *generating* phrases under program control. This might be done by providing ICL functions which call routines in the language processor, e.g., the routine NEWNODE. A special notation will be needed to specify phrase *selection*, i.e., calls to the routine FIND. For example, the specification

WITH *x* -> *<FORM:a>* '+' *<TERM:b>* DO *action* END

can mean

Execute *action* for each occurence of the phrase

<FORM> + <TERM>

in the parsing graph *x*.

For each phrase match, *action* will be executed where the variables *a* and *b* are set to the meanings under the matched <FORM> and <TERM> respectively.

Finally, in order to support *production schema*, the meta-language should support *wild-card* part-of-speech specification. For example, the following rule schema specifies the datatype requirements of the

IF-THEN-ELSE construct:

```
RULE  < ?T: f(a,b,c) >  ::=
                    'IF'  <BOOL:a>  'THEN'  <?T:b>  'ELSE'  <?T:c>
```

The wild-card part-of-speech *?T* matches any part-of-speech. All occurences of the part-of-speech *?T* in this rule must match the same part-of-speech. There are many important uses for production schema as shown in the section *Languages*. Arbitrary constraints can be placed upon wild-card parts-of-speech in the MACRO-10 meta-language.

Bibliography

[1] Knuth, D.E. "On The Translation of Languages From Left to Right" Information and Control, 8,6 (1965), Pages 607-639

[2] Aho, A.V. and Ullman, J.D. Principles of Compiler Design Addison-Wesley, 1977

[3] Earley, J. "An Efficient Context-Free Parsing Algorithm" Comm ACM, 13,2 (Feb 1970), Pages 94-102

[4] Thompson, F. and Dostert, B. Practical Natural Language Processing: The REL System as Prototype *in* Advances In Computers, Academic Press, Vol 13, 1975

[5] Kay, M. "Experiments with a Powerful Parser" The RAND Corp., memorandum RM-8452-PR, Oct 1967

[6] Jensen, K. and Wirth, N. PASCAL User Manual and Report Springer-Verlag, New York, N.Y., 1975

[7] Teitelman, W. INTERLISP Reference Manual XEROX Palo Alto, Calif. 1975

Appendix 1

A Sketch of The Language Processor in MACRO-10

This section documents some of the macros defined in the assembly language implementation for the language processor. ICL was implemented thru the use of these macros. It will be assumed that the reader is familiar with MACRO-10. The latter part of this section documents refinements to the parser and grammar representation which further optimize the matching process, e.g., the routine FIND documented earlier. Finally, I will describe the set of source files for both the language processor and ICL.

Declaration of Parts-of-Speech

The macro

TYPES < *name* , *name* , ... >

declares each *name* to be a part-of-speech. TYPES assigns each name a unique number. The macro

RNGTYP < *number*,*name* , *number*,*name* , ... >

declares each *name* to represent an array of parts-of-speech of size *number*. This is in no way meant to allocate storage. The *number*s merely increment the unique number allocator. Examples are:

```
TYPES  <RANGE,SSV,VDECL,TYPEX>
RNGTYP  <32,EXPR, 32,BOP, 3,DECL, 3,QUANT>
```

The array parts-of-speech, e.g., EXPR, are useful for implementing precedence grammars. This declaration for EXPR makes the values EXPR+0, EXPR+2, EXPR+4, ... , EXPR+64 valid parts-of-speech. All parts-of-speech declared by these macros are assigned *odd* numbers in order to satisfy the conventions imposed by the garbage collector.

For terminal parts-of-speech, i.e., the ASCII characters, append a **.$** to the character to obtain the corresponding part-of-speech. Non-alphabetic characters have special names; see the file ICLSYN.MAC.

Rule Declarations - The Coarse Form

A rule of grammar is declared with the RULE macro:

RULE *righthand phrase* , *variables* , *action*

The *righthand phrase* must be a list of parts-of-speech and *variables* must be a list of variables and *action* must be machine code. For example,

RULE < LSET , TYPEX , RSET > , <,X> , *action*

specifies the production

?? ::= { <TYPEX> }

(LSET is the part-of-speech for "{" and RSET is "}"). This declaration also specifies that when the righthand phrase is matched, the variable X will be set to the meaning under the matched TYPEX. *Action*, having access to X, will be performed upon each match. The lefthand phrase for the production should be generated by *action*. *Action* will be entered where the register LEFT contains the LEFT field of the leftmost node in the matched phrase.

The complete production

<TYPEX: //[X;] *STRNGT*(X)\\> ::= { <TYPEX:X> }

is declared by

```
RULE  <LSET,TYPEX,RSET>,<,X>,<
          SUSPEND   STRNGT,<X>
          GIVEA(TYPEX)
          DEAD
      >
```

The macro SUSPEND implements the //..\\ notation and GIVEA implements a call to NEWNODE. SUSPEND defines NEWNODE's parameter SEM and GIVEA both defines NEWNODE's parameter POS and actually calls NEWNODE. DEAD signals the end of the action and assembles as a POPJ instruction.

The second parameter in the RULE macro, the list of variables, corresponds to the first parameter in almost a one-to-one manner. Basically, the first variable will be set to the meaning under the first part-of-speech in the matched righthand phrase, and so forth for the remaining parts-of-speech and variables. The one-to-one correspondence locally becomes a one-to-two correspondence when a specified part-of-speech is one declared by the RNGTYP macro. An array part-of-speech will match any part-of-speech between its bounds. The two variables corresponding to an array part-of-speech are set to hold the meaning and the specific matched part-of-speech respectively. Referring to the part-of-speech declarations given above,

```
RULE  <EXPR,BOP,EXPR>,< X,P1, Y,P2, W,P3> , Action
```

implements the production scheme

?? ::= <EXPR> <BOP> <EXPR> .

Action will find X, Y, and W containing the meanings under the matched phrase and *action* will find P1, P2, and P3 containing the specific parts-of-speech held by each of the matched nodes. For example, the precedence production scheme

<EXPR$_i$: //[x;y;w;] *EBOP*(x,y,w)\\>
::= <EXPR$_u$:x> <BOP$_i$:y> <EXPR$_v$:w>

where u is required to be less than or equal to i and where v is required to be strictly less than i is implemented by

```
RULE  <EXPR,BOP,EXPR>,< X,P1, Y,P2, W,P3>,<
    MOVE  POS,P2        ; Part-of-speech of BOP
    ADDI  POS,EXPR-BOP  ; Displace into range of EXPR, i.e,
                        ; BOP_i goes to EXPR_i
    CAMGE  POS,P1       ; i must be greater or equal to u
      DEAD              ; otherwise, abort this rule
    CAMG   POS,P3       ; i must be greater than v
      DEAD
        ; Precedence conditions are now satisfied.  Also,
        ; POS contains the part-of-speech for EXPR_i
    SUSPEND  EBOP,<X,Y,W>   ; SEM:= //[X;Y;W;] EBOP(X,Y,W)\\
    GIVEA                   ; Generate EXPR_i.  No parameter
              ; is specified because POS is already set
    DEAD    >                   .
```

The user who wishes to declare a general rewrite rule, e.g.,

```
<A: //[x;] f(x)\\> <B: //[x;y;] g(x,y)\\  ::=  <C:x> <D:y>
```

must specify the generation of the lefthand phrase as has been described in the section about the parser, e.g.,

```
RULE  <C,D>,<X,Y>,<
        PUSH.  COLUMN      ; OLD_COLUMN:= COLUMN
        SETZ   COLUMN,     ; COLUMN:=NIL

        SUSPEND  f,<X>     ; SEM:= //[X;] f(X)\\
        GIVEA(A)           ; Call NEWNODE

        MOVEI  LEFT,(COLUMN)    ; Step Right
        POP.   COLUMN

        SUSPEND g,<X,Y>    ; SEM:= //[X;Y;] g(X,Y) \\
        GIVEA(B)           ; Call NEWNODE
        DEAD     >
```

The //..\\ Notation: SUSPEND

The macro

```
SUSPEND  f,<X,Y,Z>
```

implements the statement

```
SEM:= //[X;Y;Z;]  f(X,Y,Z)\\
```

As the reader may recall, SEM is the *meaning* parameter to NEWNODE. In general, all meanings are represented by programs in this way. *F* must be the name of a procedure declared by the SUSFUNC macro (see below).

The SUSFUNC macro - Another Component of the //...

A function which implements a meaning, i.e., one whose name is used in the SUSPEND macro, must be declared with the SUSFUNC macro. SUSFUNC is a declarative statement:

SUSFUNC(*name* , *R* , *frozen parameters* , *local variables*)

Procedure Body

DEAD

The *frozen parameters* is a list of variables. Upon entrance to the procedure body, these variables will be set to the values that were contained in the variables specified in the SUSPEND macro. *Local variables* specify the names of variables which are to be local to the procedure body. *R* specifies a reluctance; the default is zero. The reluctance of a production is specified with the production's meaning routine.

For example, referring to the <EXPR> <BOP> <EXPR> rule given earlier, we can implement the routine EBOP with the following conventions:

1) EX(*an EXPR*) sets register 1 to a number, *and*

2) EX(*a BOP*) sets register 1 to a number where it is expected that the global variables ARG1 and ARG2 will first be set to two numbers.

EBOP is then defined by

```
SUSFUNC(EBOP,,<E1,B1,E2>)

    EX(E1)      ; ARG1:= value from lefthand EXPR
    MOVEM  1,ARG1

    EX(E2)      ; ARG2:= value from righthand EXPR
    MOVEM  1,ARG2

    EX(B1)      ; Give the BOP control.  Leaves
    DEAD        ;  register 1 containing result.
```

A Finer Control Over the Matching Process - WANT

The parser's matching routine, FIND, is actually implemented by a sequence of invocations of the WANT macro. WANT matches one element of a phrase. That is, WANT takes a part-of-speech and a single column and searches the column for nodes having the given part-of-speech. Upon each match, WANT "returns". Unlike with standard procedure protocol, WANT does not leave the stack level unchanged upon return. WANT returns having pushed some data onto the stack. The user specifies that WANT is to resume its searching by performing a POPJ.

For example, a call to FIND with RHS = <A><B><C> is implemented by the sequence

```
WANT(C)
WANT(B)
WANT(A)
```

That is, from the given parsing graph in register P, WANT looks down the column referenced by P and stops at each node whose part-of-speech is C. Upon each match, WANT(C) "returns" and WANT(B) executes. Whenever WANT "returns", WANT leaves P containing the LEFT field of the matched node. Thus, cascaded calls to WANT implement the routine FIND. When WANT can find no more matches, WANT itself executes a POPJ. In this example, when WANT(B) finds no more matches, WANT(B) executes a POPJ and thus gives control back to WANT(C) so that WANT(C) will try to find another C node.

WANT can take a second parameter which specifies a variable into which WANT will store the meaning associated with the matched node. Thus, the righthand phrase

```
..  ::=  <A:X>  <B:Y>  <C:W>
```

may be programmed as

```
WANT(C,Z)
WANT(B,Y)
WANT(A,X)
  Body
POPJ
```

Body will be executed upon each occurrence of the phrase <A><B><C> within the parsing graph referenced by P. *Body* will be executed in an environment where X, Y, and W have the meanings associated with the matched nodes and where P contains the LEFT field of the matched A node. The POPJ at the end of *body* will give control back to WANT(A) so that WANT(A) will resume searching for another A-node. When WANT(A) finds no more A-nodes, WANT(A) POPJ's and thus gives control back to WANT(B). If WANT(B) finds another <B> node, WANT(B) will again give control to WANT(A) with P containing the LEFT field of the newly matched B-node. In this way, alternate phrases represented in a parsing graph are transformed into backtracking program execution.

An Optimization - Factored Righthand Phrases and Ordered Columns

When a grammar is called, the grammar has to search for each production's righthand phrase within the given parsing graph. A certain saving will be achieved if some of the searching effort can be shared

among the various righthand phrases. Consider, for example, that the set of righthand phrases

```
<A> <B> <C>

<D> <B> <C>

<E> <Q> <C>
```

can be factored from the right to yield

```
<A> ---- <B> ---- <C>
 |        |
<D>       |
          |
<E> ---- <Q>
```

In trying to find instances of these three righthand phrases, the factored representation facilitates some sharing of the searching effort. That is, rather than searching for a <C> node three times, once for each righthand phrase, the search for a <C> node can simultaneously serve all three righthand phrases.

With the WANT macro, the unfactored set of righthand phrases is searched by

```
PUSHJ.   [  WANT(C)
            WANT(B)
            WANT(A)
             body1
            POPJ.      ]

PUSHJ.   [  WANT(C)
            WANT(B)
            WANT(D)
             body2
            POPJ.      ]

PUSHJ.   [  WANT(C)
            WANT(Q)
            WANT(E)
             body3
            POPJ.      ]
```

The factored righthand phrases are searched by

```
WANT(C)

PUSHJ.  [  WANT(B)
           PUSHJ.   [  WANT(A)
                       body1
                       POPJ.      ]

           PUSHJ.   [  WANT(D)
                       body2
                       POPJ.      ]
           POPJ.   ]

PUSHJ.  [  WANT(Q)
           WANT(E)
           body3
           POPJ.   ]
```

Each match of a C-node serves simultaneously for all three phrases.

In fact, WANT takes a third parameter which specifies the address of a program to which WANT will branch when WANT can find no more matching nodes. When the third parameter is specified, WANT performs the branch rather than performing a POPJ. Thus, the factored righthand phrases can be searched by

```
          WANT(C)
            WANT(B,,LABLE1)
              WANT(A,,LABLE2)
                body1
              POPJ.

LABLE2:       WANT(D)
                body2
              POPJ.

LABLE1:     WANT(Q)
              WANT(E)
                body3
              POPJ.
```

In general, the searching of alternate parts-of-speech from within the same column is efficiently implemented by a series of WANTs linked together by their third parameters, e.g., the phrases

```
... <A>
... <B>
... <C>
... <D>
```

implemented by

```
        WANT(A,,LABLE1)
          body1
        POPJ.

LABLE1: WANT(B,,LABLE2)
          body2
        POPJ.

LABLE2: WANT(C,,LABLE3)
          body3
        POPJ.

LABLE3: WANT(D)
          body4
        POPJ.
```

Given a column and a set of alternate parts-of-speech to be searched, we can achieve further optimization by requiring that both the column and the set of alternate parts-of-speech be ordered, e.g., in increasing order by part-of-speech. This constraint will facilitate a linear rather than quadratic search time. That is, rather than independently searching the column for each given part-of-speech, we can find all matches with exactly one scan thru both the column and the given set of parts-of-speech.

In fact, both the procedure NEWNODE and the macro WANT are written to create and examine ordered columns with ordered grammars. NEWNODE inserts a new node into COLUMN at an appropriate place so to preserve order in COLUMN. WANT ceases to search for a given part-of-speech in a given column as soon as WANT comes across a node whose part-of-speech is greater than the given part-of-speech. WANT branches to the address

specified in its third parameter *leaving P containing* the unsearched portion of the given column rather than setting P to the start of the given column. In this way, an ordered series of WANTs linked via their third parameters search a given column in a single scan.

A generalization of the macro WANT allows the specification of an array of parts-of-speech:

WANTR(LOW , HIGH , P1 , X , ALT)

specifies a search for any part-of-speech between LOW and HIGH. Upon each match, the variables P1 and X are set respectively to the matched part-of-speech and meaning. The final parameter, ALT, is identical to the third parameter in the WANT macro.

All productions specified via the declarative RULE macro are initially assembled as list structures. Upon system initialization, all of the righthand phrases specified in RULE macros are gathered. The set of righthand phrases is then factored from the right and ordered. Finally, the factored datastructure as a whole is compiled into machine code as though optimally specified with the use of the WANT macro. The resulting program becomes the grammar.

Several grammars, e.g., grammars for a multipass system, are compiled separately so that each grammar may be independently and dynamically engaged to the parser. Another declarative macro enables the user to specify that following productions are to belong to a specified grammar. There is another macro which engages a grammar to the parser. The file NEWBMT.MAC contains relatively complete documentation on these macros.

The Source Files Making Up the Language Processor

Universal Files

BIGMAC Top level universal file.

Contains register assignments and macros for memory management and other generally useful macros and opdefs.

NEWBMT (New Basic Metalanguage)

Contains macro definitions for the metalanguage. Contains relatively complete documentation for each macro.

NMETAL Extension of NEWMBT.

NEWMBT and NMETAL together define at least the macros presented in this section.

Source Code

BEGIN System Initialization and Local UUO handler.

CIRCUS (Circulatory System)

Memory Management. Includes list-space garbage collector.

NEWSCN (New Scanner)

Contains the parser's character input routine which, in addition to generating nodes representing the input characters, generates each of the alternative phrases <ID>, <NU>, and <QS> over the appropriate input strings of characters. <ID> stands for *identifier*, <NU> stands for *unsigned integer*, and <QS> stands for *quoted text string*. The reader may note that these three parts-of-speech are treated specially in the ICL reference manual. In addition, NEWSCN ignores comments and manages the symbol table for *identifiers*.

NEWPAR (New Parser)

Includes both the parser and the semantic evaluator presented in this thesis. Many of the macros defined in NEWBMT reference programs contained in NEWPAR. Note one major difference in naming: There is no single prodedure corresponding to the procedure named FIND in this thesis. As mentioned earlier, the effect

of this mythical FIND is implemented by uses of the WANT macro. The WANT macro references a routine which happens to be called FIND.

GCMPIL (Grammar Compiler)

Compiles the righthand phrases of a given grammar into an efficient, factored use of the WANT macro.

CODGEN (Code Generation)

Contains the machine-code generation procedures. Serves as the assembler language for automatic code generation, e.g., supports labels and foward references. Also interfaces to the memory manager and automatically fragments the generated machine code so as to optimally use segmented free storage.

UTILS (Utilities of general interest)

Supports file I/O, numberic output, and contains a little spill over from NEWPAR.

The Source Files Making Up The Specific Language ICL

Universal Files

ICLSYN (ICL Syntax)

Declares the parts-of-speech for ICL's syntax grammar.

ICLTYP (ICL Types)

Declares the parts-of-speech for ICL's type and pass3 grammars.

ICLSEM (ICL Semantics)

Defines the datastructure which represents the user's declared non-primitive types.

ICLRUN (ICL Runtime Support)

Defines registers and fields for ICL's runtime support. Also includes the macros for code-generation. These macros reference the file CODGEN.

Syntax Files

The syntax files nearly correspond to the major parts-of-speech in ICL's syntax. Following is a list of parts-of-speech, syntax files, and semantic files. For each part-of-speech in the first column, the second column names the file declaring productions whose lefthand phrases consist of the named part-of-speech, and the third column names the file containing the programs which implement the meanings for .these productions.

<TYPE>	TYPEX	TYPEX8
<EXPR>	EXPR and EXPR1	EXPR8 and EXPR9
<DECL>	DECL	DECL8 and DDECL8
<BOP>	BOP	BOP8
<UOP>	UOP	UOP8
<RANGE>	RANGE	RANGE8
<SS>	SS	SS8
<QUANT>	QUANT	QUANT8
Processes	QUOTE	QUOTE8
Metalanguage	META	META8
Miscellaneous	MISC	(MISC itself)
Top Level	FUN	(FUN itself)

The files named in the second column contain invocations of the SUSPEND macro and the procedures named within the SUSPEND macro are defined in the corresponding file in the third column.

The Type and Third Pass Files

PASS2 RULE declarations for permanent rules of the type-pass.
PASS2B More of the same.

Both PASS2 and PASS2B together include the meaning routines under these rules. These files also include the access functions for the datastructures which represent the parts-of-speech for non-primitive types.

PASS3 Rules of the third pass.
PASS38 Meaning routines for third pass.

The routines in PASS38 generate machine code.

Miscellaneous Compiler Files

ERRORS The compile time error reporting mechanism.

Also includes the ↑C-handler.

KEYIDS Sets up correspondence between symbols used in the

MACRO-10 source to name datatypes and the identifiers used by the ICL user. KEYIDS also sets up correspondence between some of the keywords found in the syntax productions and symbols used in the MACRO-10 source as parts-of-speech for these keywords.

ICL's Runtime Support

ICLRTS and ICLRT1 Runtime support

TOPS20 A little more runtime support.

This runtime supports requires the TOPS-20 monitor.

ICLDDT The debugging package

Appendix 2

REFERENCE MANUAL FOR ICL

REFERENCE MANUAL FOR ICL

Reference Manual for ICL

Introduction

ICL was initially intended to be an upgraded PAL (precision artwork language) to further ease the design and realization of integrated circuit masks. PAL, as it turns out, is hardly programmable except that it supports assignment statements for numbers and parameterless subroutines for pictures. There is no block structure, no recursion, and no associative data structure. ICL, however, is a full blown programming language with some features especially designed for dealing with geometry.

This manual describes ICL in its full generality as a programming language. The ICL tailored for IC implementation is described in the manual titled *The IC manual for ICL*. The basic programming language is kept separate from its specialization in order to provide flexibility in keeping with evolving styles of IC design. The special functions and datatypes which define the IC-specific ICL are all implemented in ICL and thus are subject to relatively easy modification. Throughout this manual, ICL refers to the general programming language.

ICL includes many features present in both LISP and PASCAL. Like LISP, ICL encourages generative and embedded expression. A record structure, for example, may be generated in ICL without the use of assignment statements, like LISP's LIST function, whereas in PASCAL, one must assign each component separately. Unlike LISP, but like PASCAL, ICL is a completely typed language. That is to say, any computed entity must be associated with some declared datatype. ICL is completely type

safe, whereas PASCAL leaves a few areas inadequately type checked: For example, PASCAL gives the user completely independent access to the case key and to the body of a variant data structure.

ICL represents its data in terms of pointers. The user, however, may ignore the existence of pointers altogether. Except for one, optional operator, the existence of pointers is invisible. The use of pointers in the implementation allows for efficient and automatic data sharing. Besides, the user may define recursive data structures without thinking about pointers.

ICL supports *process* expressions and in fact, has *process* types. That is, a program may be packaged along with some current context and passed off as datum. At some later time, this datum may be evaluated, causing the program to execute then and there. The evaluation occurs in the current context combined with the old context which was saved at the time of the packaging.

ICL supports user-defined type coercions. A type coercion is a declaration specifying that one datatype may implicitly be transformed into a second datatype via a given program. Even the common integer-to-real coercion, which is implemented in almost every language including FORTRAN, is user-defined in ICL. A coercion is a function which has no name and whose invocation occurs without any specification whatsoever. The compiler will apply coercions throughout the user's program in the effort to maintain datatype consistency.

Type coercions are essential to support the notion of equivalence classes of representations. For example, a geometric line may be represented either by a pair of points or by three numbers ,which represent the coefficients of a linear equation. After one has defined the two coercions relating these representations, an instance of a line may be generated in either of the two forms and independently accessed in either form. Thus, a routine which requires, say, the equation representation for a line can work even if given a line in the pair-of-points representation.

Independent of type coercions, a single procedure name may be shared by several different procedures. One example of this is found in the programming language PASCAL: The procedure-name WRITE is the name of the procedure which prints integers and is simultaneously the name of the procedure which prints booleans. The operation, WRITE, is defined for more than one datatype. In ICL, the user may define many different procedures using the same name so long as they are distinguishable by their input or output datatypes. Throughout the languages of science, there are many operators whose definitions depend on the types of their parameters. For example, ABSolute-value is defined on integers, reals, and points. The operator DISPLACE can be defined to mean "displace a point by a point", or "displace a mask by a point", or even, "displace a linear transform by a point".

The space of datatypes may be extended to include many distinct types whose representations are identical. For example, a list of points is a suitable representation for both a wire and a convex polygon. However, the set of convex polygons is clearly a subset of the

set of all lists of points. In ICL the construct "PRIVATE" enables the user to specify a new type which is a restricted form of an existing type. He can then specify coercions between the restricted and unrestricted types. For example, a list of points could be coerced into the restricted type, a convex polygon, via a program which verifies convexity and which reorders the list of points to trace the polygon in the clockwise direction. A convex polygon could be coerced back into a list of points via the identity. Thus, the user can define procedures which take convex polygons as input and which access the input as lists of points. The user can be certain that the input is indeed clockwise and convex.

Datatypes in ICL provide more utility than do datatypes in PASCAL. PASCAL's datatypes serve mainly to aid the compiler in detecting program inconsistencies. ICL's datatypes not only check program integrity, but also play an active role of choosing which functions to call and which coercions to invoke where.

The type pass in ICL operates as a parser trying to come up with a successful parse in a language whose parts-of-speech are datatypes. The rules of grammar come from the coercion, function, and datatype definitions. The compiler generates machine code. All decisions about when to apply coercions or what functions to use are made at compile-time. Thus, the free use of datatypes has no runtime overhead *per se*.

Datatypes are to programming languages as units are to physics. A meaningful equation describing a physical principle must not only make sense syntactically but must also make sense in terms of units. It

often happens that one can complete an equation very easily guided only by the units requirements. My experience is that much of programming is very automatic once one knows the type of object to produce where.

Overview

The ICL system is composed of four major sections: a meta-language, a compiler, a debugging package, and a ↑C (control-C) handler. The meta-language is used to specify input source files for the compiler to read, output files on which to keep a complete record of the session's activity, and files which are to be closed or forgotten. The compiler is the main body of ICL. The debugging package permits the user to trace the execution of functions and to set break points at functions' entrances and exits. It also gives him the ability to look at and set a function's input and output parameters. The debugging package can be called from a running ICL program. The ↑C handler responds to ↑C's and will accept several one-character commands.

I will proceed by describing the meta-language and the ↑C-handler first. These components are applicable nearly everywhere. Then I will describe the ICL language itself and finally, the debugging package. However, I must first define some basic terms and conventions used throughout ICL.

Basic Conventions

Throughout this manual, the term "letter" refers only to capital letters.

An identifier in ICL is a letter followed by a sequence of either a letter, a digit, or an underscore (_). An identifier is terminated only by some character other than a letter, digit, or underscore. From here

on out, <ID> will mean identifier.

A comment is text which is completely ignored by the compiler. Comments begin and end with a double quote (").

Text strings, also known as Quoted strings, are specified by beginning and ending with a single quote ('). A single quote may be entered into the quoted string by placing two single quotes with no intervening characters. The symbol <QS> will be used to denote a quoted string.

An uninterrupted string of digits without leading zeros will be denoted by <NU>. An unsigned integer number is an instance of <NU>.

The term *blank*, or *blanks*, will be used to denote any non-empty sequence of spaces, tabs, carriage-returns, line-feeds, or form-feeds. Blanks are ignored except in the following places: Blanks in a quoted string are preserved, and as noted above, blanks cannot occur within an <ID> or <NU>.

We shall adopt a slightly extended BNF notation for specifying the syntax of ICL. A BNF rule has the format

lefthand phrase ::= *righthand phrase*

where each *phrase* is a sequence of parts-of-speech. A part-of-speech is either an identifier enclosed in angle brackets, e.g., <ID>,, a literal identifier, e.g., IF, or a character. A rule which is written as

lefthand phrase :::= *righthand phrase*

is equivalent to the first form in all respects except for a tiny matter relevant only to the interpretation of ICL's syntax error messages.

Thus far, we have introduced the parts-of-speech <ID>, <NU>, and <QS>. I have refrained from using BNF to describe <ID>, <NU>, or <QS> because unlike other parts-of-speech in ICL, blanks are not ignored in these parts-of-speech. Blanks in all other ICL forms are optional. Hence, the righthand phrases of BNF rules implicitly invite blanks between their elements.

There is one other commonly used part-of-speech, <IDLIST>, which we can describe by the rules:

```
<IDLIST>  ::=  <ID>
<IDLIST>  ::=  <IDLIST> , <ID>
```

This states that an <IDLIST> is a sequence of <ID>s separated by commas. For example, the following is an instance of <IDLIST>:

```
OBI_WAN_KENOBI,DARTH_VADER ,THE_FORCE , LUKE
(----IDLIST--)
(-----------IDLIST--------)
(----------------------IDLIST--------)
(--------------------------IDLIST----------)
```

Some computer terminals cannot accept the characters "{" or "}". These characters are used extensively in specifying strings, or lists of objects. For these poor terminals, ICL has the rules

```
{   ::=   [ )
}   ::=   ( ]
```

so that a [) will pass as a { and a (] will pass as a }.

The ICL system receives teletype input from one of two ports: the compiler port and the general port. Unless otherwise mentioned, all input goes into the compiler port. The compiler port tries to interpret its input as ICL source language text. The general port is used by ICL's error handlers and all running ICL programs. The general port is merely a character by character port. It follows none of the conventions described above and it does not understand the meta-language or the ICL language.

The compiler port takes in characters a line at a time. This means that the compiler does not see any input until a *break* character is typed. Included in the set of break characters are ↑G (bell), carriage-return, and ↑Z.

MACRO Hackers

> The input TTCALLs comprise the general port. An "XCT SCANIN" is the compiler port. It sets AC 1 to the character. The "XCT SCANIN" does not itself follow any of the above conventions.
>
> Do not use any I/O channels except via the mechanisms provided in UTILS.MAC

Meta-language

The meta-language is entered by typing a /* and is left by typing a */. Any text *produced* by the enclosed meta-statements appears to substitute for the /* ... */ string. *Producing* text means feeding the text to the compiler port. Any sequence of the following meta-statements may appear between the /* and the */. The part-of-speech <file> will be described after the meta-statements are

described.

Input

READ <file> ;

produces the text contained in <file>. The default extension is ICL. The compiler port takes in characters from <file> but the general port remains unaffected. Hence <file> should contain only ICL source language text and meta-language text. Any input requested by ICL's error handlers or by any running user's program will *not* be taken from <file>.

EREAD <file> ;

(echo read) is equivalent to READ except that the text is also echoed to the terminal.

COPY <file> ;

produces the text contained in <file> like READ, but, in addition, any input requested thru the general port is also taken from <file>. Both input ports take characters from <file>. Hence it is conceivable that <file> may contain source language text, meta-language text, user program input, and responses to questions posed by ICL's error handlers. The default extension is ICG (ICl loG).

ECOPY <file> ;

is COPY with echo to the TTY.

Output

IN_LOG <file> ;

produces nothing. However, *all* characters input from the TTY, starting after the terminating */, will go to <file>. Default extension is ICG. Note that since all your keystrokes are recorded, you can completely replay your session by restarting ICL and then typing "/*ECOPY <file>;*/ <carriage-return>". IN_LOG records all TTY input from both TTY ports.

OUT_LOG <file> ;

produces nothing. However, all characters typed out to the TTY go into <file>.

FULL_LOG <file> ;

produces nothing. All TTY characters input or output go to <file>.

MACRO Hackers

If you take BIGMAC.MAC as a universal file, all TTCALL's will be intercepted for the LOG files. "TTCAL." has been OPDEFed to the real TTCALL.

Ending

CLOSE <file> ;

produces nothing. Closes <file>. This is necessary to insure the existence of the output files. For input files, CLOSE is equivalent to FORGET.

FORGET <file> ;

produces nothing. For input files, FORGET cuts short the input by simulating an early EOF. For output files, FORGET undoes all writing that has occurred to the file. The old version, if any, remains untouched.

CLOSE and FORGET work for any files, even if they are being used by a running ICL program. CLOSE and FORGET may occur asynchronously. Input files are cut short, and further output to the output file is ignored.

Meta-language Filenames

A <file> is described by the following BNF rules:

1) <file> ::= <ID>

takes the default extension

2) <file> ::= <ID> .

blank extension

3) <file> ::= <ID> . <ID>

extension specified

4) <file> ::= <file> - <file>

The concatenation of the two files; may not be used

for specifying output files.

5) <file> ::= <ID> : <file> ;

<ID> specifies a device for all of <file>. Note that even if the specified device is TTY, none of the TTY characters taken in thru this mechanism will appear on any of the LOG files.

6) <file> ::= <file> [<NU> , <ID>]

Project Programmer Number (PPN) specification. <File> may not be one directly from (4).

Note that <file> represents only a subset of the PDP-10's possible filenames.

Examples:

1) If the file A.ICL contains the text "+2*K", then

I:= JOHN /*READ A;*/; is equivalent to

I:= JOHN +2*K;

2) If the file B.WHO contains the text "+3/*READ A;*/;", then

I:=JOHN/*READ B.WHO;*/ is equivalent to

I:=JOHN+3+2*K;

3) The following are equivalent:

```
/* IN_LOG X; READ A-B-C;*/

/* IN_LOG X; READ A; READ B-C;*/

/* READ A-B; IN_LOG X; READ C;*/

/* READ A;*/ /*READ B-C;*/ /*IN_LOG X;*/
```

However, the following is different:

```
/*IN_LOG X; READ A-B;*/ /* READ C;*/
```

The file X.ICG will begin with the characters /* READ C;*/. Remember that the IN_LOG takes effect immediately following the closing */.

↑C-Handler

While running ICL, typing one or two ↑C's will get you into the ↑C-handler. The ↑C-handler prompts with a "<>". Typing twenty or thirty ↑C's should get you to the monitor level in case of an ICL bug. Sometimes, the "<>" will not appear at first; I don't know why. However, in either case, typing one of the following letters will do ...

H

(Help) Type out a reminder of these letter commands.

C

(Continue) Ignore the ↑C and resume what was being done.

B

(Bye) Get out. Go to monitor level. You may CONT from the monitor level and be back in the ↑C-handler.

E

(Exit) Prepares to make a save file. You are then asked for an initial message. Type anything and terminate by a ↑G (bell). ICL then exits. If you do a SAVE, you can later run the saved file and be right back where you were just before the ↑C. You will first be greeted by your initial message.

The E command will not exit if any I/O channels are currently open. If any I/O channels are open, the user will be notified and the E will proceed like the C command does.

A

(Abort) Abort a running ICL program. Acts like a "C" if an ICL program is not currently running. The debugging package will be entered as soon as some function is entered or left.

D

(DDT) Enter DDT. Return from DDT by DDT's <altmode>G.

I

(Intercept) Intercept the compiler port so that when it requests another character, the compiler port will take

characters directly from the TTY. Intercept is meant to enable the user to override the current input source for the compiler port. This is useful if a long file is currently being read by the compiler port.

"I" leaves you in the ↑C handler. Do a "C" so that the system will continue processing. When the compiler next asks for input, it will be waiting for TTY input.

The compiler port will resume taking in characters from the original source immediately after you type a ↑Z.

You may not intercept an intercept; intercepts may not be nested.

Example:

You have done /*READ A;*/ and the file A is the wrong file; you would like to put an early end to A. Do an intercept ("↑C I C") and then type "/*FORGET A;*/↑Z". The first part of A will have been read in, but nothing since the ↑C.

NOTE: ↑C's are not recorded on any LOG file, nor are any of these single-character commands. These letter commands come thru neither input port.

THE COMPILER

The compiler is the main body of ICL. The compiler responds to user input by attempting to view it as a valid ICL program. If successful, the compiler then generates machine code and transfers control to it. When the user program terminates, ICL is ready to respond to more user input.

ICL is documented entirely in terms of the individual rules of grammar which define ICL's syntax. Each syntax rule is independent from all the rest and in fact plays the role of an individual, predefined function. So, for example, where LISP defines the function "(COND ...)", ICL defines the construct "IF ... THEN ... ELSE ... FI". Associated with each syntax rule is additional, non-syntactic information. This additional information expresses requirements imposed by further compiler passes. For the user to understand ICL's error messages, he must be aware of the overall structure of the compiler.

Compiler Structure and Error Reporting

ICL is implemented as a three pass compiler. The first pass enforces *syntactic* requirements, the second pass enforces *datatype* consistency, and the third pass enforces consistent use of data *sources* and data *sinks*. In the event that a user's program is ill-formed, he will be informed as to which pass failed and will be given a set of possible reasons for failure. Each pass has a different way of reporting error conditions.

Syntax Errors

Failures from the first pass, the *syntax pass*, are reported by typing back the user's input in a partially compressed form. *Incorrect* sections of input are not compressed and hence appear unmodified. However, some *correct* sections of text *are compressed* in the sense that, in place of the correct section, ICL gives the appropriate syntax part-of-speech enclosed in angle brackets. For example, the following syntactically incorrect text:

```
HAPPINESS := IF TODAY=SATURDAY THEN 100 ELSE 0 FI * %
                    YESTERDAYS_HAPPINESS + K*20 ;
```

yields the syntax error message:

```
HAPPINESS := <EXPR> * %
                    YESTERDAYS_HAPPINESS + K*20 ;
```

The text between the IF and FI is correct and has been compressed.

Unfortunately, the error reporter's notion of syntactic correctness is more restricted than ICL's. Some correct sections will not be compressed. The compression of correct sections occurs on a rule by rule basis and not every rule participates in compression. Compression occurs only with *deterministic* rules. The documentation for each syntax rule specificies whether or not the rule is deterministic. The "::=" of the BNF notation is replaced by ":::=" in deterministic rules.

Consider the example above. The IF-THEN-ELSE-FI rule *is* deterministic, and because its use in the example has no errors, the IF-THEN-ELSE-FI rule has been compressed. However, the "K*20" *is correct* but it is not compressed. The rule which implements infix operators ,e.g., the "*", is *not* deterministic. The syntax error

message would be more informative if every rule were deterministic. However, a rule can be deterministic only if its applicability can be determined without reference to surrounding text. The infix-operator rule cannot be deterministic because, given the text "1+2", its applicability depends on whether the "1+2" is contained in "A:=1+2;" or "A:=1+2*N".

The only certain information the user can derive from a syntax error message is that

1) Compressed sections are syntactically correct, and
2) Non-compressed sections may or may not be correct, except that
3) A non-compressed section involving a *deterministic* rule is definitely not correct.

Datatype Errors

Failures of the second pass, the *type-pass*, are reported in terms of ICL's syntax rules. The user, when informed of a type-pass failure, will be told which syntax-rule failed the type-pass. The user should then look up the syntax rule and understand that he violated the type requirements associated with that rule. The user will also be given a *backtrace* of grammar rules, so that he can see *where* in his program the faulty syntax rule was applied.

PASS3 Errors

Errors emanating from the third pass are not well reported by ICL. Fortunately, PASS3 errors are relatively rare and may be characterized rather simply. Since PASS3 enforces consistent use of data sinks and

sources, a PASS3 error indicates that the user has put a non-sink on the lefthand side of an assignment statement, e.g., "1:=A+B;".

The one subtlety of a PASS3 error is that a datatype coercion can leave a valid sink as a non-sink. That is, there is one kind of error which might be characterized as a type-error, but which ICL detects only in PASS3. This occurs when the type-pass, in order to satisfy type requirements, coerces something which will later turn out to be a data-sink. The canonic example is this: The user has defined the coercion from integer to real (FLOAT) but has not defined the coercion from real to integer (FIX). The error occurs when he assigns a real to an integer, e.g.,

```
I:=R; .
```

The type-pass will be forced to *coerce the lefthand side* into a real in order to have matching types across the assignment. Thus, the type-pass has effectively put a function call on the lefthand side, yielding

```
FLOAT(I):=R; .
```

The lefthand side is no longer a data sink. If, on the other hand, the user wishes such an assignment to be valid, he must supply a coercion from real to integer (FIX), so that the type-pass can be satisfied by coercing only the righthand side of the assignment, yielding

```
I:=FIX(R); .
```

I will return to the matter of error reporting with examples after some of ICL is formally defined.

ICL's Rules of Grammar

ICL is documented entirely in terms of its syntax rules. An earlier section, *Basic Conventions*, outlines the form of a syntax rule.

ICL's rules of grammar will be grouped together by the part-of-speech appearing on the lefthand side of a rule. Each group of rules defines a distinct component of ICL. The components of ICL are named by ICL's major parts-of-speech. There is, however, one group of rules which has no lefthand side. This group makes up what is called the ICL process.

ICL's Major Syntax Parts-of-Speech

ICL's linguistic constructs fall into one of two categories: declarations and algorithms. Algorithms, or sentences, are executable forms which perform actions. Declarations, on the other hand, are linguistic specifications which augment the type-grammar, the language of the second pass. Declarations consist of function definitions, datatype definitions, coercion definitions, and the declaration of variables. Declarations and algorithms may be embedded within one another. Declarations, being linguistic augmentation, have their effects manifested *implicitly* within algorithms.

Declarative statements fall under the part-of-speech <DECL>. Algorithms take on the part-of-speech <SS>, read as *sentence*. Within algorithms, computed *values* take on the part-of-speech <EXPR>. Within <EXPR>s, infix binary operators, e.g., +,-,*, and /, take on the part-of-speech <BOP>, read as *binary operator*. Loop-generating

statements, quantifiers, take on the part-of-speech <QUANT>. Within declarations, a *datatype expression* is called a <TYPE>. More parts-of-speech will be introduced to implement sub-sections of the major parts-of-speech.

The ICL Process

The compiler is an infinite loop which repeatedly waits for the user to type a sequence of characters which can be parsed as a possibly null sequence of <DECL> and <SS> terminated by a ↑G (bell). The compiler *always* responds to a ↑G except within comments or quoted strings. If the compiler does not respond to a ↑G and is indeed waiting for TTY input, then the user has forgotten to close a comment or quoted string. The user should then type a double quote (") followed by a ↑G. If there is still no response, he should type a single quote (') followed by a ↑G. The compiler will definitely have responded by this time.

In the event that the input text, if any, has not parsed into a sequence of <DECL> and <SS> prior to the ↑G, the user is notified of a syntax error. He is given the choice of seeing the syntax error message which contains the partially compressed form or skipping it. In either case, the compiler finally responds with a "*" and is ready for another go around.

If, on the other hand, there are no syntax errors, a carriage-return is typed out and the compiler proceeds as follows. All of the <DECL>s are processed. This includes compiling any function definitions or coercions. Whenever a function or coercion is compiled,

the header of the function or coercion is typed out. Finally, if the declarations compile successfully, all the <SS>s are compiled and executed. The compiler ultimately responds with a "*" and is ready for another go around.

Declarations

Declarations are represented by the part-of-speech <DECL>. Declarations play the role of providing implicit information for algorithms. The four kinds of declarations are: the definition of new *datatypes*, the declaration of *variables*, the definition of *functions*, and the definition of *coercions*. The definition of a new datatype associates an identifier to a new datatype expression. The declaration of a variable associates an identifier to a datatype by creating a variable which is capable of representing instances of that type. The definition of a function associates an identifier, a set of input parameter datatypes, and an output datatype to an algorithm. The definition of a coercion associates two datatypes to an algorithm which translates an instance of the first datatype into an instance of the second datatype.

Basic to all declarations is the notion of *datatype*. We shall begin by describing the datatypes which ICL supports.

ICL's Datatypes - Part 1

The part-of-speech <TYPE> covers all datatype expressions.

Primitive Datatypes

The primitive datatypes of ICL are integer (INT), REAL, POINT, boolean (BOOL), character (CHAR), quoted text string (QS), and strings of bits (LOGICAL). A POINT is represented by a pair of REALs. We have the rules:

```
<TYPE>    ::=    INT
<TYPE>    ::=    REAL
<TYPE>    ::=    POINT
<TYPE>    ::=    BOOL
<TYPE>    ::=    CHAR
<TYPE>    ::=    QS
<TYPE>    ::=    LOGICAL ( <NU> )
```

<NU> is a decimal number which specifies the *maximum* number of bits, or word length. <NU> may be at most 36.

Instances of INT and REAL are formed just as they are in FORTRAN except that ICL will not accept the "E" notation. Note that an instance of INT will automatically pass as an instance of REAL if the user has included the INTeger-to-REAL type coercion. Instances of POINT are formed by infixing two REALs with a "#". The instances of BOOL are TRUE and FALSE. Instances of CHAR are formed by enclosing a single character between single quotes. Instances of QS are formed by enclosing any string of characters between single quotes. An instance of LOGICAL(<NU>) is formed by enclosing one or two octal numbers, separated by a space, within "L(...)". Each octal number may consist of no more than 6 octal digits. If you write two octal numbers, then the left-hand number is automatically positioned 6 octal digits to the left

in significance. The total word length implied by the octal number(s) must not exceed the <NU> in LOGICAL(<NU>).

So

256	is an INT
256.1 or .1 or 5.	are REALs
TRUE and FALSE	are BOOLs
3.1#6.5	is a POINT
3#5	is a POINT when we have the INTeger-to-REAL coercion
'C'	is a CHAR
'C' or '↑&*hi'	are QSs
L(5)	is an instance of LOGICAL(k) where k is between 3 and 36.
L(200000 451)	is an instance of LOGICAL(k) where k = 35 or 36.

The formation of instances of these types are covered formally in the section for <EXPR>s. The operations performable on the various types are also described under <EXPR> and <BOP>.

Non-primitive Datatypes

The non-primitive datatype constructs are described by the following BNF rules. Subscripts are used to distinguish instances of the same part-of-speech for later reference.

Strings

$\langle TYPE_0 \rangle$ ::= { $\langle TYPE_1 \rangle$ }

The resulting type, $\langle TYPE_0 \rangle$, is called a *STRING* of $\langle TYPE_1 \rangle$. An instance of $\langle TYPE_0 \rangle$ is an ordered sequence of instances of $\langle TYPE_1 \rangle$. Curly brackets "{}" are generally used in conjunction with strings.

Records

$\langle TYPE_0 \rangle$ ::= [<CTYPE>] where

<CTYPE> ::= $\langle IDLIST_k \rangle$: $\langle TYPE_k \rangle$

<CTYPE> ::= <CTYPE> <CTYPE>

The resulting type, $\langle TYPE_0 \rangle$, is called a *RECORD*. An instance of $\langle TYPE_0 \rangle$ is a *composite* of components where each component consists of an <ID> in $\langle IDLIST_k \rangle$ along with an instance of $\langle TYPE_k \rangle$. The subscript $_k$ is used to remind the reader that the form <IDLIST>:<TYPE> may appear more than once in a <CTYPE>. The multiple appearances are allowed because of the final syntax rule. Square brackets "[]" are generally used in conjunction with records.

Variants

<TYPE$_0$> ::= EITHER <VTYPE> ENDOR where

<VTYPE> ::= <IDLIST$_k$> = <TYPE$_k$>

<VTYPE> ::= <VTYPE> <VTYPE>

The resulting type, <TYPE$_0$>, is called a *VARIANT*. An instance of <TYPE$_0$> is an instance of *one* of the <TYPE$_k$> along with a case key, an ID from <IDLIST$_k$>. Given an instance of <TYPE$_0$>, the associated case key indicates which one of the <TYPE$_k$>s is used to represent this instance. We will use the terms *state* and *case key* interchangeably. Automatically, ICL supplies a coercion from <TYPE$_k$> to <TYPE$_0$> with this <TYPE> construct; the effect is that an instance of <TYPE$_k$> will pass as an instance of <TYPE$_0$>.

Scalars

<TYPE$_0$> ::= SCALAR (<IDLIST>)

The resulting type, <TYPE$_0$>, is called a SCALAR. An instance of <TYPE$_0$> is any one of the <ID>s in <IDLIST>.

Referencing a Previously Declared Type

<TYPE> ::= <ID>

The resulting type is precisely the user-defined type <ID>, whatever <ID> was declared to be.

There are a few more <TYPE> constructs, but their presentation is delayed until the reader become familiar with more of ICL.

Examples:

```
{ INT }
```

denotes a new datatype called a *string of* INTegers. Strings in ICL play the role of arrays in other programming languages. Strings may be indexed to select a particular element, or may have a tail selected to yield a substring of the original. However, the most common and efficient use of strings is in the program loop generator which iterates for each element in the string.

```
[ LENGTH,WIDTH:INT  CENTER:POINT  ANGLE:REAL ]
```

denotes a new datatype, called a *record* whose components are named by LENGTH, WIDTH, CENTER, and ANGLE. The LENGTH and WIDTH components are INTegers, CENTER is a POINT, and ANGLE is a REAL. This datatype might be a good representation for rectangles. Records differ from strings in that record components are named and may have differing datatypes. All elements in a string, on the other hand, are of the same datatype and their number is unbounded.

```
EITHER
   BOX =  [ LENGTH,WIDTH: INT  CENTER: POINT ]
   CIRCLE= [ CENTER: POINT  RADIUS: INT ]
ENDOR
```

is a *variant* whose possible states are named by BOX and CIRCLE. An instance of this datatype is an instance of either of these two record datatypes depending on the state.

```
SCALAR(RED,BLUE,GREEN,YELLOW,BLACK)
```

denotes a new datatype, called a SCALAR. Instances of this datatype can take on precisely five values, namely RED, BLUE, GREEN, YELLOW, and BLACK.

How instances of these various datatypes are created and accessed is presented formally under the rules for <EXPR>. However, here are a few examples of creation:

```
{ 1 ; 20 ; 70 ; 100 }
```

is an instance of { INT } which has four elements. The elements in a string are separated by semicolons.

```
[LENGTH: 5  WIDTH: 5  CENTER: .2#.1  ANGLE: 90]
```

is an instance of the record datatype defined above.

```
BOX:: [LENGTH: 1  WIDTH: 2  CENTER: .1#.5]
```

is an instance of the variant datatype described above. "BOX::" denotes the state and "[LENGTH: 1 ... " denotes the value. However, the "BOX::" may be omitted because the value's type unambiguously implies the BOX state.

Defining New *Datatypes* and Declaring New *Variables*

The declaration of new datatypes and the declaration of program variables are short and simple. One merely needs to associate an <ID>, the name for a new datatype or variable, to a <TYPE>.

Declaring Datatypes

Type declarations are characterized formally by the rules:

<DECL> ::= <TDECL> where

<TDECL> ::= TYPE $\langle ID_1 \rangle$ = $\langle TYPE_1 \rangle$;

<TDECL> ::= <TDECL> $\langle ID_k \rangle$ = $\langle TYPE_k \rangle$;

This specifies that $\langle ID_k \rangle$ is a new datatype whose representation is $\langle TYPE_k \rangle$. The latter two rules are *deterministic*.

Examples:

```
TYPE  STACK_OF_INTEGER = { INT } ;
```

specifies that STACK_OF_INTEGER is a string of INTegers. The datatype STACK_OF_INTEGER now understands all the operations which a string understands and in addition, each element of STACK_OF_INTEGER is known to be an INTeger.

```
TYPE  COMPLEX_NUMBER = [ REAL_PART, IMAGINARY_PART:  REAL ];
```

specifies that the type COMPLEX_NUMBER is a record having two components, both of which are REALs.

```
TYPE  SET_OF_POLES = { COMPLEX_NUMBER } ;
```

specifies that SET_OF_POLES is a string of COMPLEX_NUMBERs.

```
TYPE  LISP_ELEMENT = EITHER
                         ATOM = QS
                         CONS_PAIR =  [CAR,CDR:
                                        LISP_ELEMENT]
                         INTEGER_NUMBER = INT
                         FLOATING_NUMBER = REAL
                     ENDOR;
```

specifies that a LISP_ELEMENT is either an ATOM which is a quoted string, or a CONS_PAIR which is a record having a CAR and a CDR field - each of which is again a LISP_ELEMENT, or an INTEGER_NUMBER which is an INTeger, or a FLOATING_NUMBER which is a REAL. In other words, a LISP_ELEMENT residing in the ATOM state is represented by a QS, a LISP_ELEMENT residing in the CONS_PAIR state is represented by an instance of the record [CAR,CDR:LISP_ELEMENT], a LISP_ELEMENT found in the INTEGER_NUMBER state is represented by an INTeger, and a LISP_ELEMENT found in the FLOATING_NUMBER state is represented by a REAL. Note that the coercions supplied by the variant construct imply that instances of QS, the record [CAR,CDR:LISP_ELEMENT], INTeger, and REAL, all pass as instances of LISP_ELEMENT. Note also that a LISP_ELEMENT may be examined only after its current state is determined because the *representation* is dependent upon that state. The only construct ICL provides for examining a variant type, like LISP_ELEMENT,

is the variant-CASE form. The user is always required to consider each possible state when examining a variant object.

```
TYPE  COLOR = SCALAR(RED,BLUE,GREEN,YELLOW,BLACK) ;
```

specifies that a COLOR is RED, BLUE, GREEN, YELLOW, or BLACK.

Declaring Variables

The declaration of variables is characterized by

<DECL> ::= <VDECL> where

<VDECL> :::= VAR <IDLIST$_1$> = <TYPE$_1$> ;

<VDECL> :::= <VDECL> <IDLIST$_k$> = <TYPE$_k$> ;

This specifies that each <ID> in <IDLIST$_k$> is a program variable whose type is <TYPE$_k$>. The latter two rules are *deterministic*.

Examples:

```
VAR  I,J=INT; R=REAL;
```

declares I and J to be variables which contain instances of INTegers and R to be a variable which contains instances of REAL. Writing

```
VAR  C = COMPLEX_NUMBER ;
```

enables one to write the assignment statement:

```
C := [ REAL_PART: 1.2  IMAGINARY_PART: R ] ;
```

C is assigned the COMPLEX_NUMBER whose components are 1.2 and the contents of R. Also, we can now write

```
R :=  C.REAL_PART ;
```

R is assigned the REAL_PART of C. However,

```
C := R;     or   R := C ;
```

though syntactically correct, both fail the type-pass because the <EXPR>s on each side of the ":=" are not of equal types.

When are Types Equal?

There are many syntax rules which state their datatype requirments in terms of two datatypes being *equal*. The assignment statement (:=) is one such rule, because for an assignment statement to pass the type-pass, the <EXPR>s on either side must be of *equal* types. *Two types are equal only if the names of the types are identical, or if the name of one of the datatypes was defined directly from the name of the other,* or in one other case involving the <EXPR> "NIL" which is described later. In other words, two datatypes which have identical structure are not necessarily equal. Thus, the types A and B are *not equal* if they were declared by

```
TYPE  A = { INT };  B = { INT };
```

but A and B are *equal* if declared by

```
TYPE  A = { INT };  B = A;
```

The types A, B, C, and D are all equal to one another if declared by

```
TYPE  A = { INT };  B=A; C=B; D=C;
```

The declaration

```
TYPE  PATH = { POINT } ;
      WIRE = [ THICKNESS: INT  DIRECTION: { POINT } ] ;
```

not only specifies that a PATH and a WIRE's DIRECTION are indeed strings of POINTs but it also specifies that a WIRE's DIRECTION is *not necessarily* a PATH and that a PATH is not necessarily a WIRE's

DIRECTION. In contrast, the declaration

```
TYPE  PATH = { POINT } ;
      WIRE = [ THICKNESS: INT  DIRECTION: PATH ] ;
```

specifies the same *representation* implied in the previous declaration but it also specifies that a PATH and a WIRE's DIRECTION are identical types. The latter declaration specifies that an instance of PATH may be assigned into the DIRECTION component of a WIRE and visa versa, whereas the former declaration forbids such an assignment.

This rather restricted notion of type equality imposes a style of declaration which is characterized by the following conventions:

1) Do not nest <TYPE> expressions and
2) Use previously declared types in VAR statements.

For example, the declaration

```
TYPE  TWO_DIMENSIONAL_ARRAY = { { INT } } ;
```

involves nested <TYPE> expression, whereas

```
TYPE  TWO_DIMENSIONAL_ARRAY = { ONE_DIMENSION } ;
      ONE_DIMENSION = { INT } ;
```

involves no nesting of <TYPE> expressions. Also, the variable declaration

```
VAR  MAIN_JOB_QUEUE = { JOB } ;
```

involves a <TYPE> expression, whereas

```
TYPE  JOB_QUEUE = { JOB } ;
VAR   MAIN_JOB_QUEUE = JOB_QUEUE ;
```

declares MAIN_JOB_QUEUE without involving a <TYPE> expression in the VAR statement.

Holding to this style has the effect of requiring the user to give a name to each indivisible type construct. These names invariably become useful when the user wishes to declare variables or define functions which refer to sub-structures without reference to the whole structure. For instance, we can define many functions which operate on instances of ONE_DIMENSION and which have no knowledge of their role within instances of TWO_DIMENSIONAL_ARRAY. Also, if we wish to iterate thru the *vectors* of a TWO_DIMENSIONAL_ARRAY, we will need a variable of type ONE_DIMENSION.

Defining Functions and Coercions

Thus far, we've covered the declaration of types and variables. This section documents the rest of <DECL> by introducing function and coercion definitions.

Functions

A function is defined by any of the four following *deterministic* rules:

<DECL> ::= DEFINE <ID> : <SS> ENDDEFN

<DECL> ::= DEFINE <ID> = <TYPE> : <EXPR> ENDDEFN

<DECL> ::= DEFINE <ID> (<CTYPE>) : <SS> ENDDEFN

<DECL> ::= DEFINE <ID> (<CTYPE>) = <TYPE> : <EXPR> ENDDEFN

The first form defines a procedure which has no input or output parameters. The second form defines a function which has no input parameters but which does produce a value. The third and fourth forms correspond to the first two forms with the addition of input parameters. In each case, the part following the colon, the <EXPR> or <SS>, is called the *body* of the function. The <ID> is called the function *name*, and all that which precedes the colon is called the function *header*.

Remember that <SS> stands for an algorithm which produces no value and that <EXPR> stands for a computed value. Note that an "=<TYPE>" is present only in those two forms whose bodies are <EXPR>s. The <EXPR> must be of type <TYPE>. The "=<TYPE>" is absent from the other two forms, whose bodies are <SS>s.

<CTYPE> was introduced earlier when records were introduced:

<CTYPE> ::= <IDLIST> : <TYPE>

<CTYPE> ::= <CTYPE> <CTYPE>

A <CTYPE> is used here to represent input parameters. The names of the input parameters are in the <IDLIST>s and the type of each input parameter is the corresponding <TYPE>. All <TYPE>s involved in the function header must be <ID>s, the *names* of previously declared datatypes.

Input parameters are passed *by value*, not by reference as in FORTRAN. That is to say, modifications made to the input parameters from within the function body are not felt by the caller. These parameter names are to the function merely local variables whose values have been initialized to the given input values. Refer to ICL's assignment statement in the section for <SS>s to gain a complete understanding of how ICL assigns new values to variables.

Examples:

```
DEFINE  CLEAR_JOB_QUEUE:   MAIN_JOB_QUEUE:=NIL;  ENDDEFN
DEFINE  NUMBER_OF_JOBS=INT:   some INTeger <EXPR>  ENDDEFN
DEFINE  DRAW(P:PICTURE  AT:POINT):   some <SS>  ENDDEFN
DEFINE  FACTORIAL(N:INT)=INT:
          IF  N =< 1 THEN 1 ELSE N*FACTORIAL(N-1) FI
  ENDDEFN
```

Note the absence of commas in the input parameter specification for DRAW. Also note that

```
DEFINE  COPY(FROM,TO:INT):  FROM:=TO;  ENDDEFN
```

effectively defines COPY to be a no-op. A call like

```
COPY(I,5);
```

does not set I to 5; I is left untouched.

The format for calling a function depends on which of the four kinds you're calling. The first kind is called by "<ID>;" and is syntactically a <SS>. The second kind is called by "<ID>" and is syntactically an <EXPR>. The third kind is called by "<ID>(<EXPR>, ... ,<EXPR>);" and is a <SS>. The fourth kind is called by "<ID>(<EXPR>, ... ,<EXPR>)" and is an <EXPR>. This is equivalent to FORTRAN's calling syntax except that the keyword "CALL" is replaced by a terminating semicolon.

The user may declare local variables and embed <SS>s within <EXPR>s by using the ever useful rules:

```
<EXPR> ::= DO  <SS>  GIVE  <EXPR>
<EXPR> :::= BEGIN  <DECL>  <EXPR>  END
<SS>   :::= BEGIN  <DECL>  <SS>  END
```

These rules will later be documented in full. I could define FACTORIAL by

```
DEFINE  FACTORIAL(N:INT)=INT:
     BEGIN   VAR I=INT;
        DO  IF N =< 1 THEN I:=1; ELSE I:= N*FACTORIAL(N-1); FI
        GIVE  I
     END
 ENDDEFN
```

The body is put together employing our syntax rules as follows:

BEGIN <DECL> DO <SS> GIVE <EXPR> END

The ICL user may link to MACRO-10 routines by using the following function definition forms:

<DECL> ::= DEFINE <ID>: MACRO-10(<QS>)

<DECL> ::= DEFINE <ID> = <TYPE> : MACRO-10(<QS>)

<DECL> ::= DEFINE <ID> (<CTYPE>) : MACRO-10(<QS>)

<DECL> ::= DEFINE <ID> (<CTYPE>) = <TYPE> : MACRO-10(<QS>)

<QS> is a quoted-string which names the global symbol representing the address of a MACRO-10 routine. The compiler obviously does not try to verify that the MACRO-10 routine does indeed expect the designated number of arguments and produce the right type of data. This is taken on faith.

MACRO Hackers

> The routine may damage AC's 1 thru 6, TX (13), and RET (16). Each argument, a one word entity, is pushed onto the stack and then a PUSHJ is executed. The stack register is STK (17). The routine, upon returning, must decrement the stack pointer by (the number of arguments + 1). The output value, if any, is to be returned in AC DATA (1). The files ICLRTS.MAC and ICLRT1.MAC contain ICL's runtime support and the user is free to call upon them. Atop your MACRO-10 file, copy the text residing on the first page of ICLRT1.MAC. Further details are not yet available.

Coercions

The user may define *coercions* via:

<DECL> ::= LET <ID_1> BECOME <ID_2> BY <EXPR> ;

<ID$_1$> and <ID$_2$> must be names of previously declared datatypes. <EXPR> must be of the type designated by <ID$_2$> and must be a data *source*. Within <EXPR>, <ID$_1$> is automatically declared to be a variable of type <ID$_1$>, and is initially set to the input argument.

Examples:

```
LET BOOL BECOME INT BY IF BOOL THEN 1 ELSE 0 FI ;
```

declares that any BOOLean value may be viewed as an INTeger value via the transformation which takes TRUE to 1 and FALSE to 0.

```
LET REAL BECOME COMPLEX_NUMBER BY [REAL_PART: REAL
                                   IMAGINARY_PART: 0 ] ;
```

declares that any REAL may become a COMPLEX_NUMBER by generating a COMPLEX_NUMBER whose REAL_PART is the given REAL, and whose IMAGINARY_PART is 0.

Coercions apply only to data sources, not to data sinks. A coercion is not a macro, rather, it is a function. There is a note in the introduction for *The Compiler* which mentions how a coercion may participate in masking a type-pass error, only to have it show up as a PASS3 error.

The ICL user may define a coercion in terms of a routine written in MACRO-10 by:

```
<DECL> :::= LET <ID> BECOME <ID> BY MACRO-10( <QS> )
```

The MACRO-10 routine should act like a function with one input parameter.

Example:

```
LET  INT  BECOME  REAL  BY  MACRO-10('FLT$')
```

The file BEGIN.ICL is usually the first file read into a freshly loaded ICL system. BEGIN.ICL includes the above INT to REAL coercion plus the definitions for WRITE (on INTs, REALs, CHARs, QSs, POINTs, BOOLs) and COS, SIN, TAN, and other such utilities.

Multiple Coercions

With the three coercions

```
LET  BOOL  BECOME  INT  BY  . . .
LET  INT  BECOME  REAL  BY  . . .
LET  REAL  BECOME  COMPLEX_NUMBER  BY  . . .
```

any BOOL can be viewed as a COMPLEX_NUMBER because ICL will apply coercions upon coercions if necessary. However, of all the possible ways that coercions can be applied, ICL will always choose a way which *minimizes* the total number of coercions. For example, if in addition to these three coercions, the user declares

```
LET  BOOL  BECOME  COMPLEX_NUMBER  BY  . . .
```

ICL will apply this fourth coercion instead of applying the other three coercions when a BOOL must be viewed as a COMPLEX_NUMBER.

Miscellaneous <DECL>s

Any sequence of <DECL> is also a <DECL>:

<DECL> ::= <DECL> <DECL>

The order of <DECL>s is irrelevant.

EXECUTABLE FORMS

Executable forms consist of all of ICL's linguistic constructs except declarations. Executable forms are represented by the parts-of-speech <EXPR> and <SS>. An <EXPR> represents a computed value whereas <SS> represents a sentence form, or action.

Each functional form of <EXPR> or <SS> will be described by a generalized rule of grammar. A generalized rule consists of a set of BNF rules each having a *name* and numbered righthand phrase elements, a set of type requirements including specification of a resulting type, a set of requirements for PASS3, and a description of meaning with examples. Any reference to a rule made by the ICL compiler will be by the rule's *name*. An explanation of this rule format follows the presentation of the first rule.

ICL is an expression oriented language. That is to say, the majority of syntax rules define <EXPR>s and only a few rules describe <SS>s. We shall begin by describing <EXPR>s.

Computed Values: <EXPR>s - Part 1

The part-of-speech <EXPR> represents a computed value. Because ICL is a typed language, each computed value is an instance of some type. It is relatively easy to partition off certain sets of <EXPR> forms: some <EXPR> forms deal with *strings*, some deal with *records*, and some deal with *variants*. Independently, there are some <EXPR> forms which deal with all kinds of types such as the IF-THEN-ELSE, function calling, and various other forms.

Each non-primitive type construct, e.g., string, record, and variant, has a special set of <EXPR> forms which perform *generation* and a special set of forms which perform *selection*. Generation refers to the creation of new objects and selection refers to the examination or analysis of existing objects. For example, LISP has the generation forms CONS and LIST and the selection forms CAR and CDR. Languages like PASCAL and FORTRAN have no generating forms: The effect of generation is achieved only by putting individual selection forms on the lefthand sides of individual assignment statements.

The part-of-speech <EXPR> actually stands for an array of parts-of-speech. The common notion of *operator precedence* (e.g., multiplication before addition) splits the part-of-speech <EXPR> into <EXPR>s of various precedences. We denote a particular element in this array of parts-of-speech by writing *<EXPR> of precedence i*. The precedence of an <EXPR> is precisely the precedence of that binary operator (<BOP>) most recently used in constructing the <EXPR>. Operator precedence will be described in full with the set of rules which integrate <BOP>s and <EXPR>s. Let it suffice for the time being

that unless otherwise specified, the <EXPR> appearing on the lefthand side of a BNF rule always has precedence zero and that the <EXPR>s appearing on the righthand side always invite any precedence. Thus, the user can ignore all considerations of precedence except in those rules having some specified precedence condition.

Many of the <EXPR>-rules presented in this section have counterparts for <SS>s. For example, the IF-THEN-ELSE construct is defined both for <EXPR>s and for <SS>s. The <EXPR> IF-THEN-ELSE chooses one value among many values and the <SS> IF-THEN-ELSE chooses one action among many actions. Refer to the section *Sentence Forms: <SS>* to see which <EXPR> forms carry over to <SS> forms.

In any description for PASS3 requirements, the terms *sink* and *target* will be used interchangeably.

The IF-THEN-ELSE

EBIF: <EXPR$_0$> ::= <BIFE$_1$> <EXPR$_2$> THEN <EXPR$_3$> ELSE <EXPR$_4$> FI

where

BIF1: <BIFE> ::= IF

BIF2: <BIFE> ::= <BIFE$_{k1}$> <EXPR$_{k2}$> THEN <EXPR$_{k3}$> EF

Type Requirements:

<EXPR$_2$> = BOOL = <EXPR$_{k2}$>

result = <EXPR$_3$> = <EXPR$_4$> = <EXPR$_{k3}$>

PASS3:

<EXPR$_2$> = SOURCE = <EXPR$_{k2}$>

result = <EXPR$_3$> = <EXPR$_4$> = <EXPR$_{k3}$> = (SOURCE or TARGET)

Meaning:

The <EXPR$_2$> and <EXPR$_{k2}$>s are evaluated in sequence until one of them evaluates to TRUE. Then the corresponding THEN <EXPR>, either <EXPR$_3$> or one of the <EXPR$_{k3}$>s, is evaluated and that is all. However, if niether <EXPR$_2$> nor any of the <EXPR$_{k2}$>s yield TRUE, then the ELSE <EXPR>, shown as <EXPR$_4$>, is evaluated.

Examples:

```
K:=  IF  A=B  THEN  1  ELSE  2  FI  ;
```

The meaning is nearly obvious: K is assigned 1 if A=B, otherwise, K is assigned 2.

```
K:=  IF  A<1  THEN  1
```

```
EF  A<2  THEN  2
    ELSE  3  FI    ;
```

K is assigned 1 if A<1, otherwise, K is assigned 2 if A<2, otherwise, K is assigned 3. "EF" is short for "ELSE IF".

```
IF  A<1  THEN  I  ELSE  J  FI    := 5 ;
```

If A<1, then I is assigned 5, otherwise J is assigned 5.

An Explanation of the Generalized Rule Format

The IF-THEN-ELSE construct was described by the three rules whose names are EBIF, BIF1, and BIF2. These syntax rules are easily understood by using "IF" in place of <BIFE>. Thus the first rule, EBIF, is simply "IF <EXPR$_2$> THEN <EXPR$_3$> ELSE <EXPR$_4$> FI". Substituting "IF" for <BIFE> is legitimate by the second rule, BIF1. In general, to comprehend a set of rules like these, use the simplest rules as direct substitutions. Now, since <BIFE> is essentially "IF", we can view the third rule, BIF2, as "IF <EXPR> THEN <EXPR> EF", and furthermore, we could substitute the <BIFE> in the first rule by "IF <EXPR> THEN <EXPR> EF" and come up with "IF <EXPR> THEN <EXPR> EF <EXPR> THEN <EXPR> ELSE <EXPR> FI", etc.

The use of "*k*" in some of the subscripting conforms to the fact that there may be many occurences, due to the recursive structure of the BNF rules. For example, the <EXPR$_{k3}$> in the rule BIF2 refers to all <EXPR>s occupying that slot between "THEN" and "EF" in *each* application of the rule BIF2. The "*k*"*s* are merely reminders about the possible multiplicity of the subscripted entity.

The *type requirements* state that the <EXPR>s immediately following <BIFE>s must be of type BOOL, and futhermore, that all of the other <EXPR>s, those which follow "THEN" or "ELSE", may be of any type so long as they are all of *equal* types. The resulting type, the type of <EXPR$_0$>, is given this common type. Note that ICL will apply coercions in order to satisfy type requirements. Thus, for example, the <EXPR>s following the THEN and the ELSE may be of different types as long as there exists some common type to which ICL can coerce each of the these given <EXPR>s. Similarly, <EXPR>s following <BIFE>s can be of any type as long as that type can be coerced to the type BOOL.

The *PASS3 requirements* state that the <EXPR>s immediately following the <BIFE>s must be data *sources*. The other <EXPR>s may either all be sources or all be targets. Allowing the target case means that an IF-THEN-ELSE may appear on the lefthand side of an assignment statement. Note that the term "evaluate" applies not only to sources, but also to targets. Evaluating a source means producing a value, and evaluating a target means consuming some given value.

The naming of rules and the numbering of their righthand elements facilitates a concise identification scheme. For instance, "EBIF 3" identifies the THEN-clause in the rule EBIF. The "*" in

```
IF...THEN  IF...THEN...ELSE * FI  ELSE...FI
                 (-------EXPR--------)
           (---------------EXPR----------------------)
```

is identified by the *backtrace*:

EBIF 4

EBIF 3

which says that the "*" is in the ELSE clause, <$EXPR_4$>, of an IF-THEN-ELSE, and that furthermore, this IF-THEN-ELSE is itself in the THEN clause, <$EXPR_3$>, of an enclosing IF-THEN-ELSE. The "*" in

IF...THEN * EF...THEN...EF...THEN...ELSE...FI

(---BIFE-----)

(--------------BIFE------)

(----------------------------EXPR-----------)

is identified by

BIF2 3

BIF2 1

EBIF 1

which says that the "*" is in the THEN clause, <$EXPR_{k3}$>, of the rule BIF2. Furthermore, the resulting <BIFE> is the <$BIFE_{k1}$> in the rule BIF2, and finally, this resulting <BIFE> is the <$BIFE_1$> in the rule EBIF. Each line in the *backtrace* specifies where, and in which rule, the previous line resides.

This identification scheme is used to specify where a type-error is found.

Terminal <EXPR>s

The following <EXPR> forms are terminal in the sense that the part-of-speech <EXPR> is absent from each rule's righthand side. *Any* <EXPR> will be expressed in terms of these basic <EXPR>s.

ENU: <EXPR> ::= <NU>

Type requirements The resulting type is INTeger.
PASS3 requirements The result is a SOURCE.
Meaning: The value is the integer <NU> itself.

Examples: 1 or 5 or 139

EQS: <EXPR> ::= <QS>

Type requirements

The resulting type is a QS (quoted text string) and is also a CHARacter whenever <QS> is one character long.

PASS3 requirements The result is a SOURCE.
Meaning The value is <QS> itself.

Examples: 'this is a QS' or 'c'

ELOG: <EXPR> ::= L (<NU>)
<EXPR> ::= L (<NU> <NU>)

Type requirements

The resulting type is LOGICAL(k) where k is greater than or equal to the total word length implied by the <NU>(s). Each <NU> must be no more than six digits long and must contain no 8's or 9's. The <NU>s are interpreted in octal.

PASS3 requirements The result is a SOURCE.

Meaning

If there is only one <NU>, then the value is the bit pattern implicit from the octal notation. If there are two <NU>s, then the bit pattern is:

$$\langle NU_1 \rangle * 2 \uparrow 18 + \langle NU_2 \rangle .$$

Examples:

L(5)	is an instance of LOGICAL(k) where k is between 3 and 36. The bit pattern is ... 101.
L(1 1)	is an instance of LOGICAL(k) where k is between 19 and 36. The bit pattern is ... 001 000 000 000 000 000 001

EFNU: <EXPR> ::= *a floating number*

Type requirements The result is a REAL.

PASS3 requirements The result is a SOURCE.

Meaning The REAL value itself.

Examples: 1.1 or 1.09 or .1 or .09 or 50.

There is no "E" notation.

ETRU: <EXPR> ::= TRUE

Type requirements The result is a BOOL.

PASS3 requirements The result is a SOURCE.

Meaning TRUE

EFALS: <EXPR> ::= FALSE

Same as TRUE, but the meaning is inverted.

ENIL: <EXPR> ::= NIL

Type requirements

The result is a pseudo-type called NIL. An explanation follows.

PASS3 requirements The result is a SOURCE.

Meaning "Undefined"

The NIL pseudo-type is a one of a kind entity in ICL. NIL is not a type. Variables may not be declared to be of NIL type. However, NIL is operational in that it is *equal* to any type excepting INT, REAL, BOOL, CHAR, and any LOGICAL or SCALAR type.

The <EXPR> construct "DEFINED(<EXPR>)", which is documented later, is the only way to sense a NIL value.

EID: <EXPR> ::= <ID>

Type requirements

The resulting type is either the type of the *variable* <ID> if <ID> is a declared variable, or any *scalar* type which includes <ID> in its <IDLIST>, or the output type of a parameterless function whose name is <ID>.

PASS3 requirements

In the *variable* case, the resulting PASS3 status is SOURCE and TARGET. The other cases yield SOURCE only.

Meaning

In the *variable* case, the SOURCE state means that the contents of <ID> are fetched, and the TARGET state means that <ID> is set to hold a given value. In the *scalar* case, the value is <ID> itself. In the function case, the value is the result of calling the function, <ID>.

Examples:

Refering to the examples presented in the section *Declarations* ...

I	represents the contents of the INTeger variable I
BLUE	represents an instance of the scalar type COLOR
NUMBER_OF_JOBS	represents the value obtained by calling the function NUMBER_OF_JOBS.

Two more terminal <EXPR> forms will be introduced in *Part 2*.

String <EXPR>s

String Generation

Strings are generated by

STRGEN: <EXPR> ::= { <REXPR>

where

SEXP: <REXPR> ::= $\langle EXPR_1 \rangle$ }

SEMNOP: <REXPR> ::= $\langle RANGE_1 \rangle$ }

SCRNG: <REXPR> ::= $\langle RANGE_{k1} \rangle$; $\langle REXPR_{k2} \rangle$

SCEXP: <REXPR> ::= $\langle EXPR_{k1} \rangle$; $\langle REXPR_{k2} \rangle$

SCCONX: <REXPR> ::= $\langle EXPR_{k1} \rangle$;* $\langle REXPR_{k2} \rangle$

Informally, this states that an <EXPR> may be formed by writing a "{" followed by a sequence of either <EXPR> or <RANGE>, followed by a "}". The elements in the sequence are separated by either ";" or ";*". A <RANGE> is a form which yields many values:

RFUNC: <RANGE> ::= $ $\langle EXPR_1 \rangle$ $\langle QUANT_2 \rangle$

RFUNC: <RANGE> ::= COLLECT $\langle EXPR_1 \rangle$ $\langle QUANT_2 \rangle$

RFUNC: <RANGE> ::= $\langle QUANT_2 \rangle$ $ $\langle EXPR_1 \rangle$

RFUNC: <RANGE> ::= $\langle QUANT_2 \rangle$ COLLECT $\langle EXPR_1 \rangle$

A <QUANT> is a loop generator. Refer to the section on *quantifiers*.

Type requirements

All the <EXPR>s must be of *equal* types, including the <EXPR>s in the RFUNC rules. The resulting type is any type which has been declared to be a *STRING* of this common type.

PASS3 requirments

All the <EXPR>s must be SOURCEs, and the result is a SOURCE. Later on, we shall see where a TARGET form of the above is useful. The TARGET case will be covered under the FOR-quantifier.

Meaning

The value is the ordered sequence consisting of the values of the <EXPR>s and the multiple values of any <RANGE>s. The separators ";" and ";*" are equivalent; the distinction between ";" and ";*" occurs only in the TARGET case. Any <EXPR>s which evaluate to NIL are ignored. The user *cannot depend* on having the <EXPR>s and <RANGE>s *evaluated* in order of specification.

All four of the <RANGE> rules (RFUNC) are semantically equivalent. The <RANGE> produces a sequence of values by evaluating $\langle EXPR_1 \rangle$ once for each iteration caused by $\langle QUANT_2 \rangle$. Remember that a <QUANT> is a program loop generator.

Examples:

```
{ 1 ; 2 ; 3 ; 4 }
```

is the ordered sequence 1,2,3,4, and is an instance of type { INT }, e.g., STACK_OF_INTEGER declared earlier. Note, however, if { INT } had never been mentioned in a declaration, this <EXPR>, or any <EXPR> having this as a sub-<EXPR>, would fail the type pass.

```
{ 3.1#2.0 ; 1.3#6.2 ; 7.0#8.0 }
```

is a sequence of three points and is an instance of type { POINT },
e.g., SET_OF_POLES declared earlier.

```
{ 3.1#2.0 ; NIL ; 1.3#6.2 ; 7.0#8.0 }
```

is equivalent to the previous example.

```
{ 1 ; 2 ;  $ I  FOR I FROM 3 TO 7;  ; 8 }
                    (----QUANT--------)
               (--------RANGE---------)
```

is equivalent to

```
{ 1 ; 2 ; 3 ; 4 ; 5 ; 6 ; 7 ; 8 }          and
{ 1 ; 2 ;  FOR I FROM 3 TO 7;  COLLECT I  ; 8 }
           (-----QUANT------)
           (--------------RANGE---------)
```

String Selection

STRSEL: <EXPR> ::= <EXPR$_1$> [<EXPR$_2$>]

<EXPR$_1$> must be of precedence zero.

Type requirements

<EXPR$_1$> is a *string* of some type and

<EXPR$_2$> = INT.

result = that type of which <EXPR$_1$> is a *string*.

PASS3 requirements

<EXPR$_2$> = SOURCE

result = <EXPR$_1$> = SOURCE. Also

result = TARGET when <EXPR$_1$> passes as both a SOURCE and a TARGET.

Meaning

Indexing: The resulting value, in the SOURCE case, is the <EXPR$_2$>'th element in the string <EXPR$_1$>. In the TARGET case, <EXPR$_1$> is modified so that its <EXPR$_2$>'th element appears to have the new value *unless that new value is* NIL. Assigning NIL to an element of a string has the effect of *deleting* that element from the string, preserving the string's order. This is in keeping with the fact that NIL is never an element of any string. Note that in the string generation rules, STRGEN, all NIL values are ignored.

The debugging package will be entered under the following conditions: In the SOURCE case, the index is non-positive. In the TARGET case, the index is non-positive or the index is larger than the length of the string. Note, however, if the index is larger than the string length in the SOURCE case, a value of NIL, 0, or FALSE is returned.

Examples:

```
{ 2 ; 4 ; 6 } [2]     is 4.
X[3]                  is the third element in X.
X[I+1]                is the I+1'th element in X.
X[I]   := 2;          modifies X so that its I'th element is 2.
```

The statements

```
V :=  { 1.0#1.0 ; 2.0#2.0 ; 3.0#3.0 } ;
V[2] := NIL ;
```

leave V being { 1.0#1.0 ; 3.0#3.0 }.

ETAIL: <EXPR> ::= <EXPR$_1$> [<EXPR$_2$> -]

<EXPR$_1$> must be of precedence zero.

Type requirements

<EXPR$_1$> is a *string* of some type and

<EXPR$_2$> = INT.

result = <EXPR$_1$>

PASS3 requirements

<EXPR$_2$> = SOURCE.

result = <EXPR$_1$> = SOURCE. Also

result = TARGET when <EXPR$_1$> passes as both a SOURCE and a TARGET.

Meaning

Tail extraction: The resulting value, in the SOURCE case, is the substring of <EXPR$_1$> which begins at the <EXPR$_2$>'th element and continues until the end. In the target case, <EXPR$_1$> is modified so that its tail starting at the <EXPR$_2$>'th position appears to be the new value.

The debugging package will be entered under the following conditions: In the SOURCE case, the index is non-positive. In the TARGET case, the index is non-positive or the index is larger than the length of the string. Note, however, if the index is larger than the string length in the SOURCE case, the NIL string is returned.

Examples:

```
{ 2 ; 4 ; 6 } [2-]      is  { 4 ; 6 }
X[I-]                   is the tail of X, starting
                        at position I.
X[I-][1]                is equal to X[I].
X[1-]                   is equal to X.
X[I-]    :=  { 20 ; 30 ; 40 } ;
```

modifies X so that X[I] is 20, X[I+1] is 30, and X[I+2] is 40. The last element in X is now X[I+2].

Miscellaneous String Forms

ERFRSH: <EXPR> ::= REFRESH (<EXPR$_1$>) *or*

ERFRSH: <EXPR> ::= REFRESH ! <EXPR$_1$>)

Type requirements

<EXPR$_1$> is a *string* of something

result = <EXPR$_1$>

PASS3 requirements result = <EXPR$_1$> = SOURCE

Meaning

An identity: <EXPR$_1$> appears unchanged, but a possibly more efficient internal representation is chosen for <EXPR$_1$>. REFRESH is purely an optimizing consideration. The <BOP>s "$>", "$$", and "<$" leave strings in a slightly inefficient form for accessibility. REFRESH straightens out the wrinkles, so to speak. Both the resulting <EXPR> and <EXPR$_1$> are refreshed.

If the user defines a unary function named REFRESH which takes a string as a parameter, his definition overrides the first ERFRSH rule. However, the equivalent form

REFRESH ! <EXPR>)

cannot be overridden.

Examples:

Y := REFRESH(X) ;

Y gets a refreshed X.

Z := REFRESH(X) ;

is equivalent to "Z:=X;" if performed immediately after the previous example, because X is already refreshed.

EREVRS: <EXPR> ::= REVERSE (<EXPR$_1$>) *or*

EREVRS: <EXPR> ::= REVERSE ! <EXPR$_1$>)

have the same requirements as ERFRSH.

Meaning

The resulting value is the string <EXPR$_1$> in reverse order. The result is automatically refreshed. REVERSE can be overridden just as REFRESH can be overridden.

Examples:

REVERSE({ 2 ; 4 ; 6 }) is equal to { 6 ; 4 ; 2 }

REVERSE(REVERSE(X)) is equal to X.

Record <EXPR>s

Record Generation

RGENF: <EXPR> ::= <RECX>

where

SEMNOP: <RECX> ::= [<RECXT>

RGENQ: <RECXT> ::= $<ID_1>$: $<EXPR_2>$]

RGEN1: <RECXT> ::= $<ID_{k1}>$: $<EXPR_{k2}>$ $<RECXT_{k3}>$

Informally, a new record is specified by a "[" followed by a sequence of "<ID> : <EXPR>" and is terminated by a "]". The elements in the sequence are separated by blanks.

Type requirements

The resulting type is any declared *record* type which contains the <CTYPE> elements "$<ID_1>$: the-type-of-$<EXPR_1>$" and each of "$<ID_{k1}>$: the-type-of-$<EXPR_{k2}>$".

PASS3 requirements

result = $<EXPR_2>$ = $<EXPR_{k2}>$ = (SOURCE or TARGET).

Meaning

In the SOURCE case, create a new record whose component *names* are $<ID_1>$ and each $<ID_{k1}>$ and whose corresponding values are $<EXPR_2>$ and each $<EXPR_{k2}>$. Unspecified components are automatically assigned the values: 0 for INT, REAL, and LOGICAL, FALSE for BOOL, the NULL character (code 0) for CHARs, and NIL for all other types. In the TARGET case, assign into each $<EXPR_2>$ and $<EXPR_{k2}>$ the value of the corresponding component from a given structure. The user

cannot depend on having the <EXPR>s *evaluated* in their specified order.

Examples:

[REAL_PART: 5.6 IMAGINARY_PART: 3.0]

is a new instance of COMPLEX_NUMBER, a type which was declared by an earlier example in the section *Declarations*.

[THICKNESS: 1 DIRECTION: { 0#0 ; 1#1 ; 1#0 }]

is a new instance of WIRE (also declared earlier).

[THICKNESS: 1]

is a new instance of WIRE. The DIRECTION component is NIL. Note, however, that

[THICKNESS: 1 DIRECTION: NIL]

fails the type-pass because nowhere is NIL required to be *equal* to some type. In general, an isolated "NIL" may not be specified in a record.

[THICKNESS: I DIRECTION: A_PATH] := A_WIRE ;

assigns the THICKNESS of A_WIRE into I and simultaneously assigns the DIRECTION of A_WIRE into A_PATH.

Record Selection

RSELQ: <EXPR> ::= $\langle EXPR_1 \rangle$. $\langle ID_2 \rangle$

$\langle EXPR_1 \rangle$ must be of precedence zero.

Type requirements

<EXPR$_1$> must be a *record* type which includes a component whose name is <ID$_2$>. The resulting type is the type of the component named <ID$_2$>.

PASS3 requirements

result = SOURCE = <EXPR$_1$>. Also,

result = TARGET when <EXPR$_1$> passes as both a SOURCE and a TARGET.

Meaning

In the SOURCE case, the resulting value is the <ID$_2$> component of <EXPR$_1$>. In the TARGET case, <EXPR$_1$> is modified so that its <ID$_2$> component appears to have a new value.

Examples:

```
[REAL_PART:1.3  IMAGINARY_PART:2.6]  . IMAGINARY_PART
```

is the REAL, 2.6.

```
X.REAL_PART
```

is the REAL_PART of X.

Point <EXPR>s

Point Generation

Points are generated by the form

<EXPR> # <EXPR>

where each <EXPR> is a REAL. Because ICL treates "#" as an infix binary operator, please refer to the operator "#" in the section for <BOP>s.

Point Selection

PTSELX: <EXPR> ::= <EXPR$_1$> . X

PTSELY: <EXPR> ::= <EXPR$_1$> . Y

<EXPR$_1$> must be of precedence zero.

Type requirements <EXPR$_1$> = POINT. result = REAL

PASS3 requirements are like those of the rule RSELQ.

Meaning

Select the X or Y coordinate of a point. A POINT is essentially the record [X,Y: REAL]. However, a POINT is generated by REAL # REAL rather than by the record generating form [X:REAL Y:REAL].

Examples:

```
(3.0#5.6).X        is 3.0
P.X                is the x-coordinate of
                   the point P
Q.X    := 5.0 ;
```

modifies Q so that its x-coordinate appears to be 5.0.

SCALAR Selection - The SCALAR CASE Form

This section covers the CASE-form when used in conjunction with SCALAR types.

ECASEE: <EXPR> ::= CASE <$EXPR_1$> OF <$EXPRV_2$>

where

EVCASE: <EXPRV> ::= <ID_1> : <$EXPR_2$> ENDCASE

EVCASB: <EXPRV> ::= <ID_{k1}> : <$EXPR_{k2}$> <$EXPRV_{k3}$>

Informally, this states that an <EXPR> may be of the form

```
CASE <EXPR> OF
        <ID> : <EXPR>
        <ID> : <EXPR>
            ...
        <ID> : <EXPR>     ENDCASE
```

Type requirements

<$EXPR_1$> must be of some declared *scalar* type. <ID_1> and each <ID_{k1}> must either be elements of this scalar type's <IDLIST>, or must literally be the <ID> ELSE.

Result = <$EXPR_2$> = <$EXPR_{k2}$>

PASS3 requirements

<$EXPR_1$> = SOURCE

Result = <$EXPR_2$> = <$EXPR_{k2}$> = (SOURCE or TARGET).

Meaning

Evaluate <$EXPR_1$>, thus yielding an <ID> in the scalar's <IDLIST>. Look down the list of <ID_1> and <ID_{k1}>s in <$EXPRV_2$> until you find a match. Then evaluate the corresponding <$EXPR_2$> or <$EXPR_{k2}$>. If no match is found, i.e. the user hasn't specified all the <ID>s in

the scalar's <IDLIST>, then evaluate the <EXPR> corresponding to the <ID> ELSE, if there is one. Otherwise, enter the debugging package.

This form of the CASE statement is slightly more concise than a corresponding use of the IF-THEN-ELSE form.

Example:

```
DEFINE   WRITE(X:COLOR):
    WRITE(   CASE  X  OF
                        RED:  'Red'
                        BLUE: 'Blue'
                        GREEN: 'Green'
                        YELLOW: 'Yellow'
                        BLACK:  'Black'       ENDCASE   ) ;
   ENDDEFN
```

defines the function WRITE for COLORs, a type which was declared earlier, using the function WRITE defined for quoted text strings (QS). The CASE form results in the type QS because all of its clauses result in the type QS.

Variant <EXPR>s

Variant Generation

The following rule has two independent meanings. This section documents only one of its meanings. The other meaning is covered under *Type Disambiguation*. The form presented here is refered to as *explicit* variant generation.

TYPDIS: <EXPR> ::= $\langle ID_1 \rangle$ :: $\langle EXPR_2 \rangle$

$\langle EXPR_2 \rangle$ must be of precedence zero.

Type requirments

result = a *variant* type where

$\langle ID_1 \rangle$ is the name of some state in that variant type and

$\langle EXPR_2 \rangle$ is of the type corresponding to this state.

PASS3 requirments

result = SOURCE = $\langle EXPR_2 \rangle$

Meaning

The resulting value is the variant object which resides in state $\langle ID_1 \rangle$ and which has value $\langle EXPR_2 \rangle$.

Examples:

The datatype declaration

```
TYPE  EDGE =  EITHER
                  STATE1 = LINE
                  STATE2 = ARC
              ENDOR;
```

specifies that an EDGE may reside in one of two states. The names of the states are STATE1 and STATE2. An EDGE found in STATE1 is represented by an instance of the type LINE and an EDGE found in STATE2 is represented by an instance of the type ARC. If LINE5 is a variable of type LINE, then the <EXPR>

STATE1 :: LINE5

is an instance of EDGE. This EDGE is in the state STATE1 and its value is LINE5. Similarly,

STATE2 :: *an <EXPR> of type ARC*

is an EDGE residing in the state STATE2.

The following <EXPR> is *not* an EDGE:

STATE2 :: LINE5

An EDGE cannot both be in STATE2 and be represented by a LINE. An EDGE in STATE2 can only be represented with an ARC.

As described earlier with the variant <TYPE> construct, the variant <TYPE> construct provides coercions from each of its constituent types to the variant type. Thus, the type EDGE, a variant type whose constituent types are LINE and ARC, comes with the coercions

LINE -> EDGE and

ARC -> EDGE

This means that an instance of LINE and an instance of ARC each implicitly can be viewed as an instance of type EDGE. The user actually does not need to write

STATE1 :: LINE5

to have LINE5 pass as an instance of EDGE.

LINE5

by itself passes as an instance of EDGE, thanks to the coercions. Similarly,

an <EXPR> of type ARC

passes as an instance of EDGE.

The reader might wonder if it is ever necessary to explicitly specify the state for a variant object. It would seem that the coercions supplied with the variant type declaration make it unnecessary. There are two reasons why the user will want to explicitly specify the state. The state may be specified solely for clarity or style. However, there are cases where it is absolutely necessary to specify the state. Consider the following variant datatype.

```
TYPE  PICTURE = EITHER
                    POLYGON = { POINT }
                    WIRE =    { POINT }
                ENDOR;
```

The type PICTURE has two constituents whose types are identical. The <EXPR>

{ point ; point ; point }

can be viewed as a PICTURE in two ways. Is it in the POLYGON state or is it in the WIRE state? If the user does not specify the state, the string of points is ambiguous when viewed as a PICTURE.

Each of the following is unambiguous because the state is explicitly stated:

POLYGON :: { *point* ; *point* ; *point* }

WIRE :: { *point* ; *point* ; *point* }

Variant Selection - The Variant CASE form

This section covers the CASE form when used in conjunction with a variant object.

A *variant* value can be examined *only* by using the following form.

ECASE: <EXPR> ::= CASE <ID_1> OF <$EXPRV_2$>

where <EXPRV> is as defined for the scalar CASE form:

EVCASE: <EXPRV> ::= <ID_1> : <$EXPR_2$> ENDCASE

EVCASB: <EXPRV> ::= <ID_{k1}> : <$EXPR_{k2}$> <$EXPRV_{k3}$>

Type requirements

<ID_1> of ECASE, the case variable, must be a variable of some *variant* datatype. Each of the <ID_1> and <ID_{k1}>s of the <EXPRV> must either be the name of one of the states in the variant datatype or literally ELSE.

The case-variable, <ID_1> of ECASE, is automatically modified within each case-clause, the <$EXPR_2$> and each <$EXPR_{k2}$>. The *type* of the case-variable within the case-clause labeled <ID_{k1}> becomes precisely that type which is associated with the state <ID_{k1}> in the variant datatype's definition. The case-variable assumes the state's particular type because the state of the case-variable is known within each case clause. However, within the ELSE clause, if there is any, the case-variable is not modified and it still retains its original, variant type.

Within each case-clause, the user is free to assign new values into the case-variable. However, the data he assigns must be of the specific type which the case-variable assumes in the

case-clause. Once the CASE form is terminated, the case-variable appears unmodified regardless of the new values it might have been assigned from within any of the case-clauses excepting the ELSE clause.

PASS3 requirements are like those for the scalar CASE.

Meaning:

Fetch the value from the variable <ID_1>. Look down the list of <ID_1> and <ID_{k1}>s within <$EXPRV_2$> until one matches the state in which the variant value currently resides. Then evaluate the corresponding <$EXPR_2$> or <$EXPR_{k2}$> and that is all. However, if no match is found, evaluate the ELSE clause if there is one, otherwise enter the debugging package.

As noted above, the meaning of the case-variable's *name* is different within each non-ELSE case-clause. Thus, if the user wishes to refer to the original *variant* value from within a specific case-clause, he must have previously assigned the original value to another, independent variable.

Examples:

Assume L is a variable of the variant type LISP_ELEMENT declared earlier.

```
WRITE(    CASE  L  OF
                   ATOM:  'L is an ATOM'
                   CONS_PAIR: 'L is a CONS_PAIR'
                   INTEGER_NUMBER:  'L is an integer'
                   FLOATING_NUMBER: 'L is a real number'  ENDCASE
 );
```

uses WRITE of a QS to report in which state L resides. If L is in the ATOM state, then "L is an ATOM" is typed out. If L is in the CONS_PAIR state, then "L is a CONS_PAIR" is typed out, etc.

```
DEFINE   IS_ATOM(L:LISP_ELEMENT)=BOOL:
        CASE L OF  CONS_PAIR: FALSE  ELSE: TRUE    ENDCASE
  ENDDEFN
```

defines the function "IS_ATOM" to be like LISP's predicate "ATOM".

```
DEFINE   IS_LITATOM(L:LISP_ELEMENT)=BOOL:
        CASE L OF  ATOM: TRUE   ELSE: FALSE    ENDCASE
  ENDDEFN
```

defines "IS_LITATOM" to match LISP's LITATOM predicate.

```
DEFINE  CDR(L:LISP_ELEMENT)=LISP_ELEMENT:
        CASE L OF
             CONS_PAIR:    L.CDR
             ELSE:         DO  report an error  GIVE  NIL
        ENDCASE
    ENDDEFN
```

defines CDR extraction. CDR doesn't succeed unconditionally. CDR of an INTEGER_NUMBER is an erroneous request, for example. Note that within the CONS_PAIR case-clause, the variable L may have its CDR field fetched because L has been implicitly declared to be of the record type "[CAR,CDR:LISP_ELEMENT]" for the duration of the CONS_PAIR case-clause.

```
DEFINE  EQUAL(X,Y:LISP_ELEMENT)=BOOL:
    CASE X OF
```

```
        ATOM:     CASE Y OF
                      ATOM:     X=Y      ELSE: FALSE   ENDCASE
        INTEGER_NUMBER: CASE Y OF
                      INTEGER_NUMBER: X=Y  ELSE: FALSE  ENDCASE
        FLOATING_NUMBER: CASE Y OF
                      FLOATING_NUMBER: X=Y ELSE: FALSE  ENDCASE
        CONS_PAIR:    CASE Y OF
                      CONS_PAIR: EQUAL(X.CAR,Y.CAR) &
                                          EQUAL(X.CDR,Y.CDR)
                      ELSE:     FALSE     ENDCASE
      ENDCASE
    ENDDEFN
```

defines the predicate EQUAL to tell if two given LISP_ELEMENTs are identical in structure. This states that if the first LISP_ELEMENT is in the ATOM state, then equality is achieved only if the second LISP_ELEMENT is also in the ATOM state and if it has the same value. Similar requirements are used to complete the definition for EQUAL by accounting for the other states in which a LISP_ELEMENT may reside. In the CONS_PAIR state, recursion on the CAR and the CDR is used.. The binary operators "=" and "&" are defined later in the section for <BOP>s. In ICL, "=" is defined only for the primitive datatypes. Note that in the ATOM cases for X and Y, the "=" compares two QS's and that in the INTEGER_NUMBER cases, the "=" compares two INTegers, and that in the FLOATING_NUMBER cases, the "=" compares two REALs. The CONS_PAIR cases, however, are not written using "=" because "=" will not compare two LISP_ELEMENTs. LISP_ELEMENT is a non-primitive type. Note also that *only* within the CONS_PAIR clauses can we select the CAR or CDR

field of X and Y.

Type Disambiguation

The following rule has two independent meanings. One of the meanings has already been covered in the section *Variant Generation*. The meaning presented here is referred to as *explicit* type specification.

TYPDIS: <EXPR> ::= <ID_1> :: <$EXPR_2$>

<$EXPR_2$> must be of precedence zero.

Type requirments

<ID_1> is the name of a declared datatype and

result = <$EXPR_2$> = the type <ID_1>

PASS3 requirments

result = <$EXPR_2$> = (SOURCE or TARGET)

Meaning

<$EXPR_2$> is *explicitly* required to be of type <ID_1>. This construct is useful for disambiguation. The existence of type coercions and polymorphic function names may lead to ambiguities in datatypes. For example, suppose we have twice defined the function name WRITE, once for INTeger input and once for REAL input. Suppose further that we have an INTeger-to-REAL type coercion. If the user types

```
WRITE(K);
```

where K is an INTeger variable, two scenarios appear possible. One scenario is that the INTeger-WRITE will be invoked. The other is that the REAL-WRITE will be invoked after coercing K to a REAL. ICL will choose the simpler of the two scenarios because ICL

applies coercions with reluctance. In short, the *preferred* type of K is INTeger, not REAL. However, the user may force the coercion to apply by writing:

```
WRITE( REAL::K );
```

The <EXPR> "REAL::K" passes the type-pass *only* by viewing K as a REAL. The preferred type of "REAL::K" is REAL, not INTeger.

Examples:

```
INT:: 1
```

is equivalent to 1.

```
INT:: (1.0+2.3)
```

is equal to 3 if the user has supplied a REAL-to-INT coercion, so that the REAL, (1.0+2.3), may be viewed as an INT.

```
LISP_ELEMENT:: NIL
```

is a NIL LISP_ELEMENT. NIL can be made an instance of any type to which NIL is *equal* by prefixing NIL with that type's name. Thus

```
[CAR: 5   CDR: LISP_ELEMENT::NIL ]
```

passes the type pass and is equivalent to

```
[CAR: 5]
```

Function Calling

Functions with parameters are called by

ECALLP: <EXPR> ::= $<ID_1>$ $<ARGS_2>$

where

ARGS1: <ARGSX> ::= (<EXPR>

ARGS2: <ARGSX> ::= $<ARGSX_1>$, $<EXPR_2>$

ARGS3: <ARGS> ::= <ARGSX>)

Informally, this states that an <EXPR> may be formed by an <ID> followed by a "(" followed by a sequence of <EXPR>s separated by commas followed by a ")". This has the appearence of FORTRAN.

Type requirements

There must exist a declared function whose name is $<ID_1>$ and whose input parameter types sequentially match the types of the <EXPR>s in <ARGS>. An isolated "NIL" may not be passed as a parameter because NIL has no type and a test for type *equality* is not used here. However, a NIL value may be passed as a parameter by using the rule TYPDIS.

PASS3 requirements

Each <EXPR> in <ARGS> = SOURCE and

result = SOURCE.

Meaning

Evaluate, in order, each <EXPR> of <ARGS> and then call the appropriate function, $<ID_1>$. The resulting value is that of the function.

Example

EQUAL(X,Y)

is a BOOLean if X and Y are LISP_ELEMENTs, referring to an earlier example.

Functions without input parameters are called by just naming the function name. This was described in the rule EID.

<EXPR>s Involving Binary and Unary Operators

Binary operators are denoted by <BOP> and are things like "+", "-", "*", etc. Unary operators are denoted by <UOP> and <RHUOP> and are things like unary minus and boolean NOT, "-". Associated to <BOP>s and <UOP>s is an attribute called *precedence*. Precedence is precisely that syntactic notion which specifies which operators are to be performed before others, or in other words, which operators have tighter bonds to their operands. For example, "* before +", meaning that 1+2*3 is 7 and not 9, is specified by having the precedence of "*" be lower than the precedence of "+". By an internal convention in ICL, lower precedence means a tighter bond.

<EXPR>s also have an associated precedence. The precedence of an <EXPR> is precisely the precedence of the most recent <BOP> or <UOP> which was used in forming that <EXPR>. For example, the precedence of the <EXPR> "1+2" is that of the <BOP> "+", and the precedence of "1+2*3" is also that of "+" because "+" is the last operator used in forming "1+2*3". The precedence of "(1+2)*3" is that of "*".

SEMNOP: $\langle EXPR_0 \rangle$ ::= ($\langle EXPR_1 \rangle$)

The resulting <EXPR>, $\langle EXPR_0 \rangle$, has precedence zero, as do all rules for <EXPR>s which don't specify otherwise. This rule has no type nor PASS3 requirements per se, and the meaning is nothing. The only effect of the parentheses is to overide any default grouping caused by precedence.

EBOP or EBOPG: $\langle EXPR \rangle$ ::= $\langle EXPR_1 \rangle$ $\langle BOP_2 \rangle$ $\langle EXPR_3 \rangle$

Syntax requirements

Precedence: The precedence of <EXPR$_1$> must be less than or equal to the precedence of <BOP$_2$> and the precedence of <EXPR$_3$> must be strictly less than the precedence of <BOP$_2$>. This guarantees that <BOP>s with lower precedence will be combined first and that <BOP>s of equal precedence will be associated left-to-right. This is just like FORTRAN.

If <BOP> has *no precedence*, then the rule applies, but after all type and PASS3 requirements are checked, a preference is made for left-to-right association. This feature will be explained in the section about <BOP>s.

This rule does not apply if <EXPR$_1$> has the special precedence called *EMAX*. A rule of grammar producing an <EXPR> of precedence *EMAX* specifies that its result has higher precedence than any <BOP>s and therefore applies only after all <BOP>s have been bound. For an example, see the rule EGIVE, which produces an <EXPR> of precedence *EMAX*.

Type and PASS3 requirements depend on <BOP$_2$>.

Meaning

Evaluate <EXPR$_1$> and <EXPR$_3$> and then apply <BOP$_2$> to the resulting values. There is no guarantee as to which of <EXPR$_1$> and <EXPR$_3$> is evaluated first.

Examples:

```
1 + 2
1 + 2 * 3
( 1 + 2 ) * 3
```

There are more examples in the section for <BOP>s.

EUOP: <EXPR> ::= <UOP$_1$> <EXPR$_2$>

Syntax requirements

Precedence: <EXPR$_2$> must be of precedence less than or equal to 2 and the resulting <EXPR> is of precedence 2. That is to say, <UOP>s may prefix <EXPR>s of precedence zero or <EXPR>s which are themselves prefixed by <UOP>s.

Type and PASS3 requirements depend on <UOP$_1$>.

Meaning

Evaluate <EXPR$_2$> and then apply <UOP$_1$>.

Example:

```
-1      or   --1
-1+2                    equals 1, not -3
-X[I]
```

groups as -(X[I]) and not as (-X)[I] because the string indexing construct, STRSEL, requires the string-<EXPR> to have precedence zero. The string-<EXPR> X has precedence zero but the string-<EXPR> -X has precedence 2. Hence, the string-<EXPR> -X cannot be used in the string indexing construct. The unary minus is therefore attached *after* X and [I] are attached because the <EXPR> X[I] has precedence zero.

EUOP or EUOPG: <EXPR> ::= <EXPR$_2$> <RHUOP$_1$>

This is like the rule EUOP above, but it's for unary ops which must appear on the righthand side. The precedence conditions and examples are delayed until <RHUOP>s are described.

Looping with <BOP>s

The mapping from "+" to *summation* is defined for all operators. The rules:

EBOPQ: <EXPR> ::= <BOP$_1$> <EXPR$_2$> <QUANT$_3$>

EBOPQ: <EXPR> ::= <QUANT$_3$> GIVE <BOP$_1$> <EXPR$_2$> END

EBOPQ: <EXPR> ::= <QUANT$_3$> <BOP$_1$> <EXPR$_2$>

all have the same meaning: each yields the cumulative value

<EXPR$_2$> <BOP$_1$> <EXPR$_2$> <BOP$_1$> ... <EXPR$_2$>

where the number of terms is determined by the program loop generator <QUANT$_3$>. The third rule results in an <EXPR> of precedence *EMAX*.

Type requirements

There must exist some type, T, which can act as a temporary for holding the cumulating value. Thus, T is characterized by

T = <EXPR$_2$>

T = the resulting type from (T <BOP$_1$> <EXPR$_2$>)

These accommodate assigning an initial value to the temporary and then assigning cumulative values for each iteration. These two equations, in the space of datatypes, are not singular; "T=<EXPR$_2$>" doesn't bind T exclusively to one datatype. If necessary, the <EXPR$_2$> in these two equations will differ by having one be the result of applying coercion(s) to the other. Typically, however, this type constraint means that both operands of the <BOP> and the resulting type are all equal.

PASS3 requirements result = <EXPR$_2$> = SOURCE

Meaning

Apply <BOP$_1$> to the successive values of <EXPR$_2$> generated by evaluating <EXPR$_2$> once for each iteration caused by <QUANT$_3$>.

Examples:

```
        +       1       FOR I FROM 1 TO 5;
    (-BOP-) (-EXPR-)  (--------QUANT-----)
```

sums up 1s as I marches from 1 to 5.

```
        +     I     FOR I FROM 1 TO N;
```

equals 1+2+3+...+N.

```
        *       2       REPEAT N;
    (-BOP-) (-EXPR-)  (--QUANT--)
```

equals 2 to the Nth power.

```
        +     F(I)     FOR I FROM 1 TO N;
    (-BOP-) (-EXPR-)  (------QUANT-------)
```

is equivalent to

```
        FOR I FROM 1 TO N;   GIVE   +      F(I)   END
       (-------QUANT------)      (-BOP-) (-EXPR-)
```

and to

```
        FOR I FROM 1 TO N;     +      F(I)
       (-------QUANT------)  (-BOP-) (-EXPR-).
```

Note that the precedence of the third rule specifies that

```
        REPEAT 10;   + 2 * 3 + 1
```

groups as

```
        REPEAT 10;   + (2 * 3 + 1)
```

and not as

```
(REPEAT  10;  + 2)  * 3 + 1
```

or

```
(REPEAT  10;  + 2 * 3) + 1
```

The resulting <EXPR> from the third EBOPQ rule has precedence *EMAX* and hence that <EXPR> cannot be the lefthand operand for a binary operator. Refer to the precedence requirements of the rule EBOP.

Existential and Universal <EXPR>s

The following <EXPR> forms correspond to mathematical logic's existential and universal quantification. The reader might note the similarity between these <EXPR> forms and those of the previous section, *Looping with <BOP>s*. The following four QBOOL1 rules are equivalent in meaning.

QBOOL1: <EXPR> ::= <QUANT$_1$> <EXPR$_2$> <QBOOL$_3$>

QBOOL1: <EXPR> ::= <QUANT$_1$> <QBOOL$_3$> <EXPR$_2$>

QBOOL1: <EXPR> ::= <QUANT$_1$> GIVE <QBOOL$_3$> <EXPR$_2$> END

QBOOL1: <EXPR> ::= <QBOOL$_3$> <EXPR$_2$> <QUANT$_1$>

where

QBALW: <QBOOL> ::= ALWAYS

QBNVR: <QBOOL> ::= NEVER

QBEXS: <QBOOL> ::= EXISTS

QBEXS: <QBOOL> ::= THERE_IS

Syntax Requirements

The second QBOOL1 yields an <EXPR> of precedence *EMAX*.

The first rule's <QBOOL$_3$> will not admit THERE_IS and the other rules' <QBOOL$_3$> will not admit EXISTS. EXISTS and THERE_IS have identical meanings; this restriction merely enhances program readability.

Type Requirements result = BOOL = <EXPR$_2$>

PASS3 Requirements result = SOURCE = <EXPR$_2$>

Meaning

Evaluate <EXPR$_2$> once for each iteration caused by <QUANT$_1$> and stop as soon as the condition specified by <QBOOL$_3$> becomes known. If the condition becomes known before the <QUANT> is exhausted, the user will find the variables of <EXPR$_2$> holding those values which were used in the final evaluation of <EXPR$_2$>.

The <QBOOL> ALWAYS yields TRUE only when all values of <EXPR$_2$> yield TRUE. NEVER yields TRUE only when all values of <EXPR$_2$> yield FALSE. EXISTS and THERE_IS yield TRUE as soon as <EXPR$_2$> yields its first TRUE.

Examples:

```
ALWAYS   X<5   FOR  X $E S;
                (--QUANT-----)
FOR  X $E S;  ALWAYS  X<5
(---QUANT---)
FOR  X $E S;  X<5   ALWAYS
(---QUANT---)
```

are equivalent and each yields TRUE if X<5 *for all* X in the string S.

```
THERE_IS  X<5  FOR X $E S;
FOR X $E S;  THERE_IS  X<5
FOR X $E S;  X<5   EXISTS
```

are equivalent and each yields TRUE if *there exists* at least one X in S with X<5.

```
NEVER  X<5  FOR X $E S;
FOR X $E S; NEVER  X<5
```

```
FOR X $E S;  X<5  NEVER
```

are equivalent and each yields TRUE if each element in S is *not* less than 5. The following are equivalent:

```
ALWAYS  <EXPR>  <QUANT>
NEVER  - <EXPR>  <QUANT>
- THERE_IS  - <EXPR>  <QUANT>
```

The "-" is logical NOT.

Note that if

```
ALWAYS  X<5  FOR X $E S;
```

yields FALSE then X is left containing a number such that X<5 is FALSE. The form

```
IF  THERE_IS X<5 FOR X $E S;  THEN then-clause
                              ELSE else-clause FI
```

guarantees that X contains the first value in S less than 5 upon entering the THEN-clause.

```
IF  ALWAYS X<5 FOR X $E S;  THEN then-clause
                            ELSE else-clause FI
```

guarantees that X contains the first value in S which is *not* less than 5 upon entering the ELSE-clause.

```
IF  NEVER X<5 FOR X $E S;  THEN then-clause
                           ELSE else-clause FI
```

guarantees that X contains the first value in S which *is* less than 5 upon entering the ELSE-clause.

```
FOR X $E S; ALWAYS THERE_IS Y=X FOR Y $E S1;
```

or

FOR X $E S; ALWAYS FOR Y $E S1; THERE_IS X=Y

states the condition that the string S is a subset of the string S1.

IF FOR X $E S; ALWAYS THERE_IS Y=X FOR Y $E S1;

THEN then-clause ELSE else-clause FI

guarantees that upon entrance to the ELSE-clause, X holds the first value in S which is not in S1.

Embedding <SS>s within <EXPR>s

The following three rules enable the insertion of a <SS> within an <EXPR>. This section concludes with a warning about side effects and order of evaluation.

EGIVE: <EXPR> ::= DO $\langle SS_1 \rangle$ GIVE $\langle EXPR_2 \rangle$

EGRAB: <EXPR> ::= GIVING $\langle EXPR_2 \rangle$ DO $\langle SS_1 \rangle$ END

EGRAB: <EXPR> ::= DO $\langle SS_1 \rangle$ GRABBING $\langle EXPR_2 \rangle$

The first and third rules yield <EXPR>s of precedence *EMAX*.

Type requirements result = $\langle EXPR_2 \rangle$

PASS3 requirements result = $\langle EXPR_2 \rangle$ = (SOURCE or TARGET)

Meaning

The first rule specifies that $\langle SS_1 \rangle$ is executed *before* evaluating $\langle EXPR_2 \rangle$. The resulting value is that of $\langle EXPR_2 \rangle$. This is LISP's PROGN function. The second and third rules are equivalent and each specifies that $\langle SS_1 \rangle$ is executed *after* $\langle EXPR_2 \rangle$. This is LISP's PROG1 function.

Examples:

```
DO  I:=K*N-V/5;  GIVE  I+I
```

yields the value of "I+I" after I has been assigned "K*N-V/5". Note that this groups as

```
DO  I:=K*N-V/5;  GIVE  (I+I)
```

and not as

```
(DO  I:=K*N-V/5;  GIVE  I) + I
```

because the DO...GIVE rule yields an <EXPR> of precedence *EMAX*. A DO...GIVE form cannot be the lefthand operand of a <BOP>, unless, of course, it is enclosed in parentheses. Hence the "I+I" has to bind first.

```
GIVING  I+I  DO  I:=K*N-V/5  END
```

yields the value of "I+I" and *then* resets I to the value of "K*N-V/5".

```
DO  WRITE(I*I);  GRABBING  I     := 5;
```

types out a 25. The evaluation of the lefthand side of this assignment statement sends the number 5 to the DO...GRABBING form. The DO...GRABBING form first evaluates its <EXPR$_2$>, I, and thus sets I to 5. It then executes <SS$_1$>, typing out a 25.

The following rule offers a more concise notation for one of the more common DO...GIVE usages.

SETQX: <EXPR> ::= (<EXPR$_1$> <SSRHS$_2$>)

This represents an assignment statement enclosed in parentheses. The part-of-speech <SSRHS> will be documented completely in the section called *Assignment Statements*. For the time being, let us assume the definition:

```
<SSRHS>  ::=    := <EXPR> ;
```

<SSRHS> stands for "<SS>'s righthand side" and comprises the righthand side of the assignment statement *including* the :=. Thus, the form

```
<EXPR>  <SSRHS>
```

represents the assignment statement

<EXPR> := <EXPR> ;

and the form

(<EXPR> <SSRHS>)

takes on the appearance

(<EXPR> := <EXPR> ;)

The parentheses around the assignment statement transform it into an <EXPR> whose value is that value which passes thru the :=.

Formally speaking, the *type* and *PASS3* requirements and the ***meaning*** for the rule SETQX are all derived by transforming

($\langle EXPR_1 \rangle$ $\langle SSRHS_2 \rangle$) into

(DO $\langle EXPR_1 \rangle$ $\langle SSRHS_2 \rangle$ GIVE $\langle EXPR_1 \rangle$)

(-------SS-------)

Refer to the assignment statement rule, SSASS, in the section for <SS>s.

Examples:

(I:=I+1;)	increments I and yields the resulting value of I.
I:=(J:=3;);	sets both I and J to the value 3.

Notice that the semicolon is required *as part of the assignment statement*!

IF (J:=N*N;) < 4 THEN J ELSE J+1 FI

yields the value N*N if N*N is less than 4, otherwise it yields N*N+1. This is equivalent to

```
IF DO J:=N*N; GIVE J < 4  THEN J ELSE J+1 FI
```

WARNING:

Embedding <SS>s within <EXPR>s expands the notion of *computed value* to include side effects. The evaluation of an <EXPR> containing an embedded <SS> not only yields a value but it also performs some actions, typically modifying variables. Because of such side effects, the order of <EXPR> evaluation becomes a relevant issue. For example,

```
(J:=2;) * J
```

yields either a 4 or 2*(the old value of J), depending on which of the two parameters to the "*" evaluates first. The rule incorporating the "*", EBOP, clearly states that the user *cannot depend* on the order of evaluation. Thus, the above <EXPR> yields an uncertain value. Similarly, the string

```
{  DO J:=2; GIVE J   ;   J }
```

yields either {2;2} or {2;the_old_value_of_J}. The STRGEN rules clearly state that the order of evaluation in strings is uncertain.

Embedding Declarations within <EXPR>s

EDECL: <EXPR> ::= <BEXPR>

where

DCOUGH: <BEXPR> :::= BEGIN <$DECL_1$> <$EXPR_2$> END

DCOUGH: <BEXPR> :::= BEGIN <$EXPR_2$> <$DECL_1$> END

Type requirements result = <$EXPR_2$>

PASS3 requirements result = <$EXPR_2$> = (SOURCE or TARGET)

Meaning

Evaluate <$EXPR_2$>. However, the declarations, <$DECL_1$>, are incorporated for the duration of <$EXPR_2$>. Thus, the user may declare new variables to be local to <$EXPR_2$>. He may also declare new types, functions, or coercions which are accessable only within <$EXPR_2$>. Outside of the BEGIN-END block, the effects of <$DECL_1$> are absent.

Any variable or type declaration which defines a previously used name automatically overrides the name's previous definition. However, the same is not true for function and coercion declarations. Unfortunately, an attempt to override a previous function or coercion definition results in ambiguity when the function or coercion is used within <$EXPR_2$>.

Examples:

```
BEGIN   VAR I,J=INT;
        DO   I:=20; J:=30;
        GIVE  I*J
END
```

declares I and J to be local INTegers for the duration of this <EXPR>. Its value is 600. Any external meanings for I and J are unchanged.

```
BEGIN   VAR I,J=INT;
        DO  I:=10; J:=30;
        GIVE   I+J* BEGIN  VAR I=INT; J=REAL;
                           DO  I:=3;
                           GIVE  I
                    END  * I
END
```

yields the value 910: The inner BEGIN-END <EXPR> yields the value 3 and the "I+J*...*I" therefore reduces to "I+J*3*I". Even though I is redeclared inside the inner BEGIN-END block, within the outer block but not within the inner block, I has its assigned value of 10.

```
BEGIN   LET  COMPLEX_NUMBER  BECOME REAL  BY
                             COMPLEX_NUMBER.REAL_PART ;
        some <EXPR>
END
```

specifies that for the duration of the <EXPR>, COMPLEX_NUMBERs may implicitly be viewed as REALs by considering only their REAL_PARTs. This might be useful if within this block, all COMPLEX_NUMBERs were scrutinized only for their relation to the imaginary axis.

```
DEFINE  IOTA(N:INT)=STACK_OF_INT:
    BEGIN    VAR I=INT;
             { COLLECT  I  FOR I FROM 1 TO N; }
```

```
        END
    ENDDEFN
```

defines APL's iota function, which returns the string of integers from 1 to N. The variable I is local to this function and hence does not interfere with any other use of the name I.

```
    DEFINE   SUBSCRIPT(SUBJECT,INDICES:STACK_OF_INTEGER)
                                          = STACK_OF_INTEGER:
        BEGIN   VAR I=INT;
              { COLLECT  SUBJECT[I]  FOR I $E INDICES; }
        END
    ENDDEFN
```

defines APL's vector-on-vector indexing operation. The "$E" within the FOR-quantifier reads "an element of".

Global Communications - The HOLDING form and <ASN>

This section covers a structured management for global variables: the HOLDING form. The HOLDING form works for any kind of variables but it is primarily useful for managing global variables. We will also introduce the part-of-speech <ASN>.

The HOLDING form

HOLDIT: <EXPR> ::= HOLDING $\langle ASN_1 \rangle$ GIVE $\langle EXPR_2 \rangle$ ENDHOLD

Type Requirements result = $\langle EXPR_2 \rangle$

PASS3 Requirements result = $\langle EXPR_2 \rangle$

Meaning

The resulting value is that of $\langle EXPR_2 \rangle$. However, preceding $\langle EXPR_2 \rangle$'s evaluation, the *specified variables* in $\langle ASN_1 \rangle$ are saved and $\langle ASN_1 \rangle$'s *implied assignments* are carried out. After the evaluation of $\langle EXPR_2 \rangle$, the *specified variables* in $\langle ASN_1 \rangle$ are restored.

This is like LISP's PROG function except that variables without implied assignment are left unchanged, i.e., they are not set to NIL. The <ASN> in the HOLDING form corresponds to the PROG's first parameter.

Examples will follow.

The <ASN>

ASN1: <ASN> ::= $\langle ID_{k1} \rangle$;

ASNRHS: <ASN> ::= $\langle ID_{k1} \rangle$ $\langle SSRHS_{k2} \rangle$

ASNX: <ASN> ::= <ASN> <ASN>

Informally, an <ASN> is a sequence of either "<ID>;" or "<ID>:=<EXPR>;". The part-of-speech <SSRHS> is covered in the section *Assignment Statements*. <SSRHS> is basically the form

:= <EXPR> ;

Thus, the form in the rule ASNRHS

$<ID_{k1}>$ $<SSRHS_{k2}>$

appears as

$<ID_{k1}>$:= <EXPR> ;

and represents an assignment statement where the lefthand side is the variable $<ID_{k1}>$.

Type Requirements

Each $<ID_{k1}>$ must be some declared *variable* and each

$<ID_{k1}>$ $<SSRHS_{k2}>$

must satisfy the type requirements implied by the assignment. Refer to the assignment statement rule, SSASS, in the section for <SS>s.

PASS3 Requirements

Those implied by the assignments

Meaning

An <ASN> has an abstract meaning in ICL. An <ASN> represents both a set of <ID>s, all the $<ID_{k1}>$s, and a set of assignment statements, all the $<ID_{k1}><SSRHS_{k2}>$s of the rule ASNRHS. The set of <ID>s is called the *specified variables* and the set of assignment statements is called the *implied assignments*.

Examples of <ASN>s:

```
I ; J ; K ;
```

has the *specified variables* I, J, and K, and has no *implied assignments*. Notice that there is a terminating semicolon.

```
   I    ;    J   := 3 ;     K ;
(--ASN--)      (-SSRHS-) (-ASN-)
               (----ASN---)
```

has the *specified variables* I, J, and K, and has the *implied assignments* J:=3;.

```
I ;  J:= (V:=3;);  K:=2;
```

has the *specified variables* I, J, and K, and has the *implied assignments*

```
J:=(V:=3;);   and  K:=2;.
```

Note that V is not in the set of *specified variables*.

Examples of the HOLDING form:

```
HOLDING   I;J;. GIVE  <EXPR> ENDHOLD
          (-ASN-)
```

specifies that I and J are to appear unchanged after this <EXPR> is evaluated.

```
HOLDING  I;J;  GIVE
      DO  I:=1;  GIVE <EXPR>   ENDHOLD
```

specifies the same as above except that I is set to 1 before <EXPR>'s evaluation.

HOLDING I:=1; J; GIVE <EXPR> ENDHOLD

specifies the same as the previous example.

HOLDING EPSILON:=EPSILON/2; GIVE
SOLUTION_TO_EQUATION ENDHOLD

specifies that while finding SOLUTION_TO_EQUATION, EPSILON is to be cut in half.

In general, when you want to reuse a global variable, use the HOLDING form to assign it its new value, lest the global variable's previous value be lost. This kind of treatment for global variables is essential in many recursive enviornments.

HOLDING INPUT_DEVICE:=DISK; GIVE
INPUT_TEXT ENDHOLD

specifies that the function INPUT_TEXT will operate in the context where INPUT_DEVICE=DISK. As implied by its use above, INPUT_TEXT is a parameterless function. However, if INPUT_TEXT were defined to be a function of one parameter, the input device, its call would look like

INPUT_TEXT(DISK)

The use of the HOLDING form is equivalent in the sense that an input parameter is being specified. However, the latter form requires the input device to be specified upon each call to INPUT_TEXT whereas the former form sets that parameter for all calls to INPUT_TEXT, thus making the input device an implicit

parameter.

Anchoring Pointers - @ and COPY

The introduction to this manual mentions that the ICL user need not be aware of *pointers*. This section presents the anchoring operator, the only operator in ICL which requires the user to be aware of ICL's pointer implementation. The anchoring operator is useful *only* if the user wants to take further advantage of ICL's pointer implementation.

EAT: <EXPR> ::= @ (<EXPR$_1$>)

Type Requirements

result = <EXPR$_1$> = any type to which NIL is equal *except* POINT.

PASS3 Requirements

<EXPR$_1$> = SOURCE

result = (SOURCE or TARGET)

Meaning

In the SOURCE case, @(...) is a no-op; the resulting value is simply the value of <EXPR$_1$>. However, in the TARGET case, ICL stores the given value into the *memory location* occupied by the *value* of <EXPR$_1$>.

The debugging package will be entered under the following conditions: In the TARGET case, either <EXPR$_1$> is NIL or the given value is NIL, i.e., either

@(NIL):= <EXPR> ; *or* @(<EXPR>):= NIL ;

Examples:

```
A :=[REAL_PART:1 IMAGINARY_PART:2];
B := A ;
```

leave A and B referencing the same memory location. The assignment

```
A.REAL_PART := 700;
```

modifies A so that A's REAL_PART becomes 700. However, B's REAL_PART is untouched and still contains the value 1. A and B now reference different memory locations. On the other hand, if we instead were to write

```
@(A).REAL_PART := 700;
```

B.REAL_PART would *also* become 700. That is, the *memory location* referenced by A is modified, *not* the variable A itself. B feels the change because B references the same location referenced by A.

When do computed values reference the same memory location? This question cannot be answered without some knowledge of ICL's implementation. The reader is referred to the section *ICL's Policy about Assignments, Pointers, and Copying* for a complete explanation. The LISP user, however, can come to a reasonable understanding by knowing that in ICL the generation of records, strings, and variants operates like LISP's LIST function; the results occupy newly allocated memory locations. For example,

```
[A:X  B:Y]   acts like  (LIST X Y)
{P;Q;R}      acts like  (LIST P Q R)
P::Q         acts like  (CONS P Q)
```

and with ICL's TARGET selection forms,

```
W.A:=X;   acts like  W:=(CONS X (CDR W))
                if A happens to be the first
                component in W
```

W[2]:=X; acts like W:=(CONS (CAR W)
(CONS X (CDDR W]

W[2-]:=X; acts like W:=(CONS (CAR W) X).

Thus, the ICL nested record

W:= [A: [P:X] B: [P:X]];

specifies that W.A and W.B do not reference the same memory location, but that W.A.P and W.B.P do reference the same location. The operation

@(X):=Y; or @(W.A.P):=Y;

makes a change which is apparent from both of W's A and B components. However, the assignment

@(W.A).P := Y;

appears to modify W's A component without modifying W's B component. W's A and B components reference different memory locations, each containing a different copy of [P:X].

Refer to the section *ICL's Policy about Assignments, Pointers, and Copying* for a complete explanation.

ECOPY: <EXPR> ::= COPY (<EXPR$_1$>)

ECOPY: <EXPR> ::= COPY ! <EXPR$_1$>)

Type Requirements are like the rule EAT, above.

PASS3 Requirements result = SOURCE = <EXPR$_1$>.

Meaning

The resulting value is a copy of the value of <EXPR$_1$>. *The copy and the value of <EXPR$_1$> occupy different memory locations.* COPY is a very fast operator, only two PDP-10 words are moved. However, if the value of <EXPR$_1$> is NIL, COPY acts as a no-op and simply returns NIL.

If the user defines a unary function named COPY, his definition will override the first ECOPY rule. However, the equivalent form

```
COPY ! <EXPR> )
```

cannot be overridden.

Example:

Referring to the above example with A and B, if we substitute the

```
B := A ;
```

with

```
B := COPY(A) ;
```

then the assignment

```
@(A).REAL_PART := 700;
```

does *not* affect B. The structure referenced by B is not at the location referenced by A, thanks to the COPY.

Detecting NIL

EDEF: <EXPR> ::= DEFINED ($\langle EXPR_1 \rangle$)

Type requirements

$\langle EXPR_1 \rangle$ must be of a type to which NIL is equal. Refer to the rule ENIL.

result = BOOL

PASS3 requirements result = $\langle EXPR_1 \rangle$ = SOURCE

Meaning

TRUE if $\langle EXPR_1 \rangle$ is not NIL, FALSE otherwise.

Binary and Unary Operators:
<BOP>s, <UOP>s, and <RHUOP>s

<BOP>s

The part-of-speech <BOP> refers to all infix binary operators. We will denote a <BOP>'s type requirements via the notation

"<TYPE$_1$> <TYPE$_2$> -> <TYPE$_3$>".

This states that the <BOP>'s lefthand parameter must be of type <TYPE$_1$>, its righthand parameter must be of type <TYPE$_2$>, and the resulting value is of type <TYPE$_3$>. The PASS3 requirements for <BOP>s are simply that both input parameters and the output parameter are SOURCEs unless otherwise specified.

```
BOPADD:   <BOP>  ::=  +
BOPSUB:   <BOP>  ::=  -
BOPMUL:   <BOP>  ::=  *
BOPDIV:   <BOP>  ::=  /
BOPEXP:   <BOP>  ::=  ↑
```

have the type requirements:

```
INT     INT     ->  INT
REAL    REAL    ->  REAL
POINT   POINT   ->  POINT
```

+, -, *, and / are the usual FORTRAN arithmetic operators where a POINT is treated as if it were a complex number. "↑" is exponentiation and does not yet work for POINTs. + and - have precedence 6, * and / have precedence 4, and ↑ has precedence 2. Thus, * and / are preformed before + and -, and ↑ is performed before * and /. Remember that in ICL, the lower the precedence, the tighter the operator binds to its operands. In general, the actual precedence numbers are unimportant; the only importance is

their relation to one another.

The <BOP>s * and / can also be used to combine a POINT and a REAL by yielding scalar multiplication or division. The REAL and POINT may appear on either side of the *,

```
POINT  REAL   -> POINT
REAL   POINT  -> POINT
```

but division admits the REAL only on the righthand side:

```
POINT  REAL   -> POINT
```

```
BOPAND:   <BOP>  ::=  &
BOPOR:    <BOP>  ::=  !
BOPXOR:   <BOP>  ::=  XOR
```

have the type requirements:

```
BOOL     BOOL     -> BOOL
LOGICAL  LOGICAL  -> LOGICAL
```

"&" stands for boolean AND, "!" stands for boolean OR, and "XOR" stands for exclusive OR. For LOGICALs, these <BOP>s procede bitwise. The LOGICALs must all be *equal* types. For example, a LOGICAL(7) and a LOGICAL(10) cannot be combined. The precedence of "&" is 10, "!" is 12, and "XOR" is 14. Thus &'s are done before !'s and !'s are done before XOR's.

```
BOPBIT:   <BOP>  ::=  BIT
```

has the type requirments:

```
LOGICAL  INT  -> BOOL
```

BIT tests a bit in the LOGICAL and tells whether it is a one or a zero, yielding TRUE or FALSE, respectively. The INT specifies which bit is to be examined. Zero is the rightmost bit, one is the second to the rightmost bit, etc. FALSE is returned if INT isn't less than the word size of the LOGICAL. BIT has precedence 20.

BOPLSL: <BOP> ::= SHIFTL

BOPLSR: <BOP> ::= SHIFTR

have the type requirements:

LOGICAL INT -> LOGICAL

SHIFTL means *shift left* and SHIFTR means *shift right*. The INT specifies the number of bits to shift. A negative INT cause that the shift occurs in the opposite direction. SHIFTL and SHIFTR each has precedence 20.

BOPMIN: <BOP> ::= MIN

BOPMAX: <BOP> ::= MAX

have the type requirements:

INT INT -> INT

REAL REAL -> REAL

POINT POINT -> POINT

MIN and MAX yield the minimum and maximum, respectively. MIN and MAX are defined on points by proceeding coordinate-wise. Thus, "P1 MIN P2", where P1 and P2 are POINTs, yields the lower left corner of the box determined by P1 and P2. Please note that the resulting point might not equal either P1 or P2. MIN and MAX have precedence 26.

The remaining <BOP>s are said to have *no precedence*. This means that they impose no precedence conditions. However, <BOP>s with *no precedence* will tend to associate in the usual left-to-right manner when possible. <BOP>s with *no precedence* also tend to apply after the other <BOP>s have applied; they tend to have a higher precedence. What distinguishes <BOP>s of *no precedence* from <BOP>s all having equal precedence is that their grouping is flexible enough to allow a non-left-to-right grouping when *datatype* inconsistency forbids the usual left-to-right grouping.

BSHARP: <BOP> ::= #

has the type requirements

REAL REAL -> POINT

and forms the POINT whose x and y coordinates are the two REALs respectively. The "#" operator works in the TARGET case if both parameters are TARGETs. Thus, the two coordinates of a point may be unloaded into two REAL variables simultaneously.

Examples:

```
1+2*3↑2+5                 equals 24
3.0#4.0 + 1.0#6.0         equals 4.0#10.0

L(4) BIT 0                is FALSE
L(4) BIT 1                is FALSE
L(4) BIT 2                is TRUE
```

```
L(4)  SHIFTL  2          is L(20)
L(4)  SHIFTR  -2         is L(20)
L(4)  XOR   L(5)         is L(1)

(1.0#2.0)*3.0            is 3.0#6.0
1.0#2.0 * 3.0            is 1.0#6.0
```

```
L(4) BIT 3 & TRUE
```

fails the type-pass because it groups as

```
L(4) BIT (3 & TRUE)
```

because of precedence, and "3 & TRUE" fails the type-pass. However,

```
(L(4) BIT 3) & TRUE
```

passes the type-pass. ("L(4) BIT 3" yields a BOOL and so the "&" operates on BOOLeans). Note that in the form without parentheses, precedence chooses a fatal grouping.

```
X#Y := P;
```

sets the REAL variables X and Y to the coordinates of the point P.

More <BOP>s of *no precedence*:

```
COMPEQ:   <BOP>  ::=  =
COMPNE:   <BOP>  ::=  <>
COMPGT:   <BOP>  ::=  >
COMPGE:   <BOP>  ::=  >=
COMPLT:   <BOP>  ::=  <
COMPLE:   <BOP>  ::=  =<
```

These have the type requirements

```
INT      INT      -> BOOL
REAL     REAL     -> BOOL
POINT    POINT    -> BOOL
CHAR     CHAR     -> BOOL
LOGICAL  LOGICAL  -> BOOL
```

=, <>, >, >=, <, =< are the compare operators. They are, in order: equal, not equal, greater, greater or equal, less, less or equal. Note that "less or equal" and "greater or equal" place the "=" relative to the "<" or ">" so to avoid forming an arrow. "=" and "<>" also allow

```
BOOL     BOOL     -> BOOL
QS       QS       -> BOOL
SCALAR   SCALAR   -> BOOL
```

A partial ordering is assigned to POINTs and LOGICALs by comparing each of the coordinates of a POINT (or bits in a LOGICAL) and requiring that both (all) of the comparisons succeed for a successful overall comparison. CHARs are ordered by their ASCII codes. These compare operators will also work on one other datatype which is yet to be introduced.

The operators "$>", "$$", and "<$" are for appending elements or strings onto strings.

Let A be any type and SA be the type "{A}", string of A.

```
BOPSTR:   <BOP>  ::=  $>
BOPSTC:   <BOP>  ::=  $$
BOPSTL:   <BOP>  ::=  <$
```

These have the type requirements (respectively):

SA A -> SA

SA SA -> SA

A SA -> SA

"$>" appends an element onto the righthand end of a string. "<$" appends an element onto the lefthand side of a string, like LISP's CONS. "$$" appends two strings, like LISP's APPEND. Thus,

1 <$ {2;3;4} equals {1;2;3;4}

{1;2} $> 3 $> 4 equals {1;2;3;4}

{1} $$ {2;3} $$ {4;5} equals {1;2;3;4;5}

A note about efficiency might be of interest. All three of these operators are equally fast (not slow). However, *accessing* a string built with many "$>"s or "$$"s is relatively slow. The REFRESH operator, the rule ERFRSH, rebuilds a string using "<$" so that it may be accessed efficiently from thereafter.

There is one more <BOP> form, which enables calling functions in an infix manner.

BOPBID: <BOP> ::= \ <ID>

<ID> must be the name of a function which takes in two parameters and yields a value. Therefore, the types of data that this <BOP> takes in and yields are determined by the particular function, <ID>.

Examples:

Referring to the function EQUAL defined earlier for comparing two LISP_ELEMENTs,

```
X \EQUAL Y
```

is equivalent to EQUAL(X,Y). X and Y must be of type LISP_ELEMENT and "X \EQUAL Y" is a BOOLean.

```
X \EQUAL Y & Z \EQUAL W
```

automatically groups as

```
(X \EQUAL Y) & (Z \EQUAL W)
```

because the "\<ID>" <BOP> has *no precedence*, and hence it groups to suit datatype compatibility.

Now, suppose we wish to define 3-D points:

```
TYPE THREE_POINT = [X,Y,Z: REAL];
```

Suppose further that we define addition and multiplication by

```
DEFINE PLUS(A,B: THREE_POINT)=THREE_POINT:
  [X: A.X+B.X  Y: A.Y+B.Y  Z: A.Z+B.Z]  ENDDEFN
DEFINE TIMES(A:THREE_POINT  R:REAL)=THREE_POINT
  [X: A.X*R  Y:A.Y*R  Z: A.Z*R]  ENDDEFN
```

Then, if A,B, and C are variables of type THREE_POINT,

```
A \PLUS B \PLUS C
```

represents their sum, and

```
(A \PLUS B \PLUS C) \TIMES (1.0/3.0)
```

represents their average. Note that due to the left-to-right grouping tendency in <BOP>s of *no precedence*, we could get the same effect by:

```
A \PLUS B \PLUS C \TIMES (1.0/3.0)
```

or even

```
A \PLUS B \PLUS C \TIMES 2.0/3.0 \TIMES 1.0/2.0
```

This ability to classify binary functions as <BOP>s enables their use in looping-<BOP> operations. For example, if Q were a string of THREE_POINTs then we can get the average in that string by writing

```
\PLUS    A   FOR A $E Q;  \TIMES  (1.0/ + 1 FOR A $E Q;)
(-BOP-)      (--QUANT---)
```

The "FOR A $E Q;" reads as "FOR A an element of Q".

Unary Operators - <UOP> and <RHUOP>

Unary operators were mentioned earlier in the <EXPR> rule, EUOP, which states that an <EXPR> may be prefixed by a <UOP> or postfixed by a <RHUOP>. We shall specify a <UOP>'s type requirements by

$$\langle TYPE_1 \rangle \quad -> \quad \langle TYPE_2 \rangle$$

This will mean that the <UOP> takes in an instance of type $\langle TYPE_1 \rangle$ and yields an instance of type $\langle TYPE_2 \rangle$. All inputs and outputs are assumed to be SOURCEs.

UOPMIN: <UOP> ::= -

has the type requirements

INT	->	INT
REAL	->	REAL
POINT	->	POINT
BOOL	->	BOOL
LOGICAL	->	LOGICAL

Unary minus operates like in FORTRAN for INTegers and REALs. A point is negated by negating each of its coordinates independently. When applied to a BOOLean, "-" is the function NOT. "-" performs ones complement on LOGICALs.

Examples

- TRUE	is FALSE
- L(4)	is L(3) if interpreted as a LOGICAL(3), or L(13) if interpreted as a LOGICAL(4), or L(33) if interpreted as a LOGICAL(5), etc.
- (1.0#2.0)	is -1.0 # -2.0
-X#Y	is (-X) # Y

-(X#Y) is -X # -Y

UTALLY: <UOP> ::= TALLY

ULFTZO: <UOP> ::= LEFTZEROS

UENCOD: <UOP> ::= ENCODE

have the type requirements

LOGICAL -> INT

TALLY counts the number of ones in a LOGICAL. LEFTZEROS counts the number of leading zeros. ENCODE counts the number of trailing zeros.

UDECOD: <UOP> ::= DECODE

UUNARY: <UOP> ::= UNARY

have the type requirements

INT -> LOGICAL

DECODE yields a LOGICAL having at most one bit set. The INT specifies the number of trailing zeros which are to follow that one bit. If the INT is greater than the word length of the LOGICAL, the result is L(0). UNARY yields a LOGICAL which has all zeros on the left and all ones on the right. The number of ones is specified by the INT.

UNORM: <UOP> ::= NORM

UBITSW: <UOP> ::= BITSWAP

have the type requirements

LOGICAL -> LOGICAL

NORM yields the input shifted left until a 1 bit occupies the leftmost position. BITSWAP reflects the bits so that the leftmost bit becomes the rightmost bit and the second to the left becomes the second to the right, etc.

The following unary operator is classified as an <RHUOP> because it can combine with <EXPR>s only by appearing on the righthand side of the <EXPR>. <RHUOP>s have *no precedence* in the sense that <RHUOP>s tend to apply *after* the text to its left has been combined.

UOPBID: <RHUOP> ::= \ <ID>

Type requirements

<ID> must be the name of a declared function which takes in one value and produces a value. Therefore, the input and output datatypes are determined by the particular function, <ID>.

Meaning

Apply the function, <ID>, to its parameter, the <EXPR> to the left.

Examples:

```
TALLY    L(5)      is 2
LEFTZEROS L(1)
```

is ambiguous because we do not know the word length of L(1). However, if X were declared to be a variable of type LOGICAL(6) and if X were assigned L(1), then

```
LEFTZEROS  X       would be 5
ENCODE  L(4)       is 2
```

```
DECODE  5          is L(40)
UNARY   5          is L(37)
NORM    X          would be L(40)
```

If X were assigned L(3), then

```
BITSWAP   X        would be L(60).
```

If L is of type LISP_ELEMENT, then

```
L  \CDR
```

is equivalent to CDR(L) and has the effect of extracting the CDR from L, referring to an earlier declaration.

```
L  \IS_ATOM
```

is equivalent to IS_ATOM(L).

```
WRITE( IF L \IS_ATOM THEN 'L is an ATOM'
            ELSE  'L isn''t an ATOM' FI );
```

reports whether L is an ATOM or not.

Sentence Forms: <SS>

The part-of-speech <SS> refers to a sentence, or action form. Unlike <EXPR>, <SS> has no resulting type and neither produces nor consumes a value. *Evaluating* a <SS> refers to performing the specified action. There are relatively few rules which produce <SS>s. In fact, besides the assignment statement and the concatenation form, all <SS>s are carry overs from the <EXPR> section. We shall begin with the assignment statement.

Assignment Statements and <SSRHS>

SSASS: <SS> ::= <EXPR$_1$> <SSRHS$_2$>

where

SSRHS1: <SSRHS> ::= := <EXPR$_1$> ;

SSRHS2: <SSRHS> ::= ::= <BOP$_1$> <EXPR$_2$> ;

SSRHS3: <SSRHS> ::= ::= <EXPR$_1$> <BOP$_2$> ;

These rules specify that a <SS> may be formed either by

<EXPR> := <EXPR> ; or by

<EXPR> ::= <BOP> <EXPR> ; or by

<EXPR> ::= <EXPR> <BOP> ;

The first form is the basic assignment statement. The others translate into the basic assignment statement via the following mappings:

<EXPR$_1$> ::= <BOP> <EXPR$_2$> ; becomes

<EXPR$_1$> := (<EXPR$_1$>) <BOP> <EXPR$_2$> ; and

<EXPR$_1$> ::= <EXPR$_2$> <BOP> ; becomes

<EXPR$_1$> := <EXPR$_2$> <BOP> (<EXPR$_1$>) ;

For example,

I ::= + 1 ; becomes I := I + 1 ;

I ::= -3 ; becomes I := I - 3 ;

I ::= 3 - ; becomes I := 3 - I ;

We can now specify the type and PASS3 requirements for each of these forms merely by specifying those for the basic assignment form.

Type Requirements

<EXPR$_1$> of SSASS = <EXPR$_1$> of SSRHS1

PASS3 Requirements

<EXPR$_1$> of SSASS = TARGET and

<EXPR$_1$> of SSRHS1 = SOURCE

Meaning

Evaluate the righthand <EXPR>, <EXPR$_1$> of SSRHS1, and then feed the resulting value to the lefthand <EXPR>, <EXPR$_1$> of SSASS. Conceptually, the assignment statement is a process by which the lefthand <EXPR> is made to be equal to the righthand <EXPR>. The lefthand <EXPR> is pliable whereas the righthand <EXPR> is fixed.

Examples:

```
I := 5 ;              sets I to contain a 5.
C := [REAL_PART:A  IMAGINARY_PART:B] ;
                      sets C to some complex number.
[REAL_PART:A  IMAGINARY_PART:B] := C ;
```

```
                    does the opposite: It sets A and B
                    to the real and imaginary
                    components of C respectively.

I ::= +1 ;          increments I.
K ::= +2*3 ;        adds 6 to K.
K ::= *2+3 ;        multiplies K by 5.
S := S $> 1 ;       appends the element 1
                    onto the string S.
S ::= $> 1 ;        does the same.
S ::= 3 <$ ;        appends 3 onto the front of S.
S ::= $$ {5;6} ;    appends the string {5;6} onto
                    the righthand end of S.
S ::= {5;6} $$ ;    appends the string {5;6} onto
                    the lefthand end of S.
```

The two auxilary assignment forms, SSRHS2 and SSRHS3, have counterparts for <UOP>s and <RHUOP>s:

SSRHS4: <SSRHS> ::= ::= <KUOP>

where

KUOP1: <KUOP> ::= <GUOP> ;

KUOP2: <KUOP> ::= $\langle GUOP_1 \rangle$ $\langle KUOP_2 \rangle$

and where

SEMNOP: <GUOP> ::= <UOP>

SEMNOP: <GUOP> ::= <RHUOP>

Informally, this states that an <SSRHS> may be formed by a ::= followed by a sequence of <UOP> and <RHUOP>. The sequence is terminated with a semicolon. The part-of-speech <GUOP> represents both <UOP> and <RHUOP>.

The *type* and *PASS3* requirements and the meaning are derived by transforming

<EXPR$_1$> ::= <GUOP> ... <GUOP> ; into

<EXPR$_1$> := <GUOP> ... <GUOP> <EXPR$_1$> ;

Examples:

```
I ::= - ;           becomes  I := - I ;
I ::= \CDR ;        becomes  I := I \CDR ;
                        or   I := CDR(I) ;
I ::= \CDR \CDR ;   becomes  I := I \CDR \CDR ;
                        or   I := CDR(CDR(I)) ;
```

Note that all assignment forms end with a semicolon!

ICL's Policy about Assignments, Pointers, and Copying

ICL's assignment statement appears to set the lefthand side to a copy of the righthand side. For example,

```
A:= [REAL_PART:1  IMAGINARY_PART:2] ;
B:= A;
B.REAL_PART:= 700;
WRITE(A.REAL_PART);
```

prints a 1. The second assignment appears to set B to a new complete copy of A. Thus, the third assignment modifies the copy referenced by B and does not affect the copy referenced by A.

Furthermore, each and every reference to a structure appears to generate a new complete copy. For example,

```
X:= [Q:1  R:2] ;
Y:= [A:X  B:X] ;
Y.A.Q:= 700;
WRITE(Y.B.Q);
```

prints a 1. The second assignment references X twice. Y appears to be set to a record which contains two distinct copies of X. The third assignment modifies one of the copies, that which is referenced by Y.A. However, the other copy, Y.B, appears unmodified.

ICL's apparent copy policy guarentees that distinct variables reference distinct structures and thus a modification incurred from the point of view of one variable is non-existent from the point of view of another variable. This policy applies everywhere, including to the passage of parameters to functions; a function appears to receive a

distinct copy for each of its parameters. This policy is a generalization of the generally accepted treatment for integers, reals, and any datatype whose instances are not represented with the aid of pointers.

For efficiency, ICL in fact does not generate copies as described above. Rather, ICL generates copies only when a modification is specified. In the two examples given above, copying ocours only upon execution of the third assignment in each example. A and B reference the same structure until the modification is specified.

The user who plans to use the @ operator, the pointer anchoring operator, or who wishes to understand ICL's efficiency must understand how ICL implements this copy appearence. The user who avoids the @ operator can effectively believe that ICL copies upon each reference and he can ignore the concept of pointers altogether. The rest of this section documents ICL's data implementation and gives examples using the @ operator.

ICL's Implementation is in Terms of Pointers

ICL uniformly minimizes copying and maximizes memory sharing by making extensive use of pointers. Instances of each of ICL's datatypes are represented as follows:

Non-pointer Types

An instance of INT, REAL, BOOL, CHAR, LOGICAL, or SCALAR is represented by a single word. These non-pointer datatypes are precisely those datatypes which are *not* equal to the NIL

pseudo-type. Instances of these datatypes can never take on the value NIL.

Pointer Types

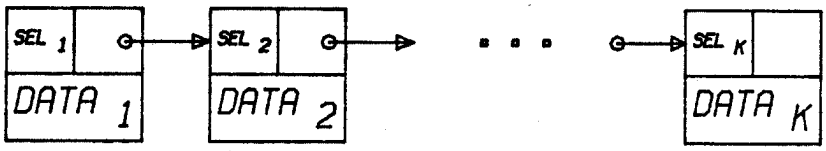
All other datatypes in ICL are represented by a single word which contains the memory address of a structure representing the instance.

Variant:

This is the result of the variant generation <EXPR>

state :: *data*

Record:



This is the result of the record generation <EXPR>

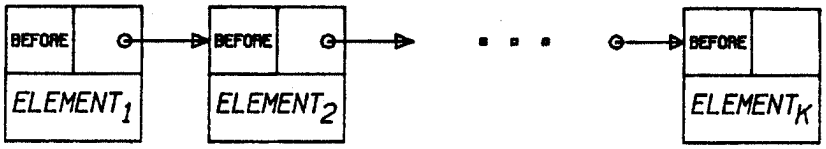
$[sel_1{:}data_1 \;\; sel_2{:}data_2 \;\; \ldots \;\; sel_k{:}data_k]$.

NIL or 0 values are not stored on this record list.

If a record component is set to NIL or 0, the corresponding memory element in the record list is removed and the record list appears to be shortened.

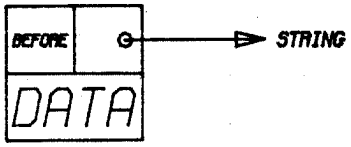
String:

A refreshed string is represented by



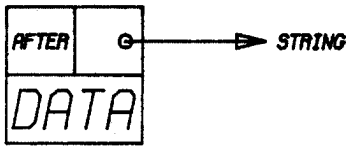
A general string, however, has three kinds of element representation:

Left Append



Data precedes all elements in *string*.

Right Append

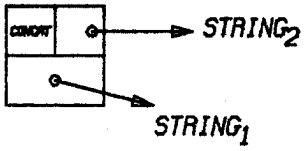


Data follows all elements in *string*.

Such a node is created with the $> operator:

string $> *data*

Concatenation



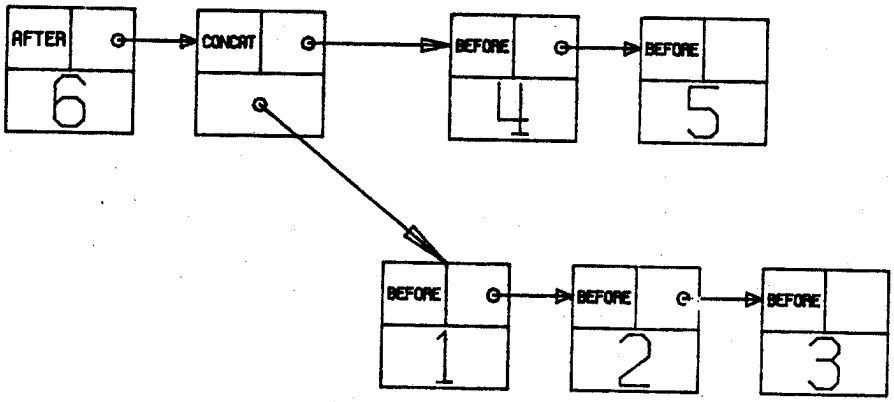
All elements in $string_1$ precede all elements in $string_2$. Such a node is created by

$$string_1 \$\$ string_2$$

For example, the string

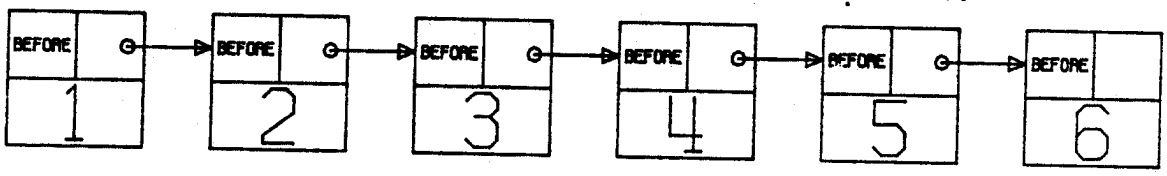
(((1 <$ {2;3}) $$ {4;5}) $> 6)

is represented by



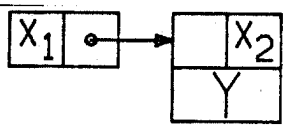
A more efficient representation will result from the form

{1;2;3;4;5;6} or {COLLECT I FOR I FROM 1 TO 6;}



The REFRESH operator applied to the former representation yields the latter representation. The latter representation is preferred because it is accessed most efficiently.

Point:



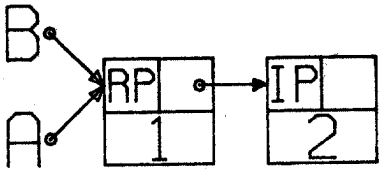
X_1 and X_2 together represent the point's x-coordinate. The whole x-coordinate is not stored in a single word; a few bits in the lefthalf of the first word of the referenced node are required by the garbage collector. This description is not quite accurate but it does bring up the difference between the representation for POINT and the representation for all other pointer types. The single word which contains the memory address for pointer types contains some non-pointer information for POINT. This difference is responsible for the exclusion of the type POINT from the @ operator's domain.

Memory Sharing

Memory sharing is automatically achieved throughout ICL's implementation by moving pointers rather than by copying the referenced structures. For example, the statements

```
A:= [REAL_PART:1  IMAGINARY_PART:2] ;
B:= A;
```

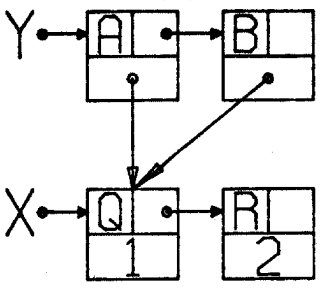
yield the following memory state:



Rather than setting B to a copy of A's record, the pointer in A is copied into B. A and B are left referencing the same record. Similarly,

```
X:= [Q:1  R:2] ;
Y:= [A:X  B:X] ;
```

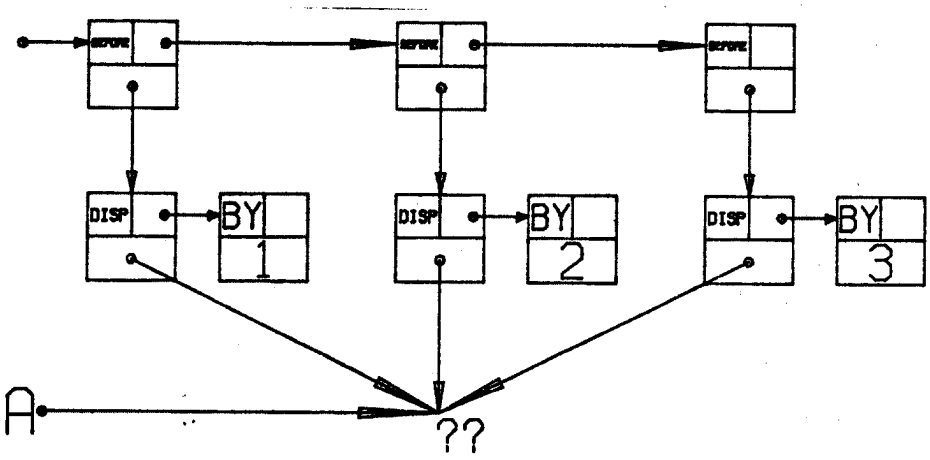
yields



The A and B components of Y reference the identical record. Finally, the <EXPR>

```
{COLLECT  [DISPLACE:A BY:I]  FOR I FROM 1 TO 3;}
```

generates the string of records:



All the DISPLACE components reference the unique memory location referenced by A.

Memory Modification

In ICL, a modification to an existing structure is specified by the appearance of a selection form as a TARGET. The following are examples of modification specification:

```
B.REAL_PART:= 700;
S[I]:= 700;
S[I-]:= R;
X.A.B:= Y;
X[2].A:= Y;
X.A.B[2][3][4].C:= Y;
```

In each example, ICL modifies only the *variable* which appears as a TARGET. Thus, the first example changes only the pointer residing in B and the final example changes only the pointer residing in X. The new pointer placed in the variable X references a newly created structure which is identical to old X with the exception that the access path

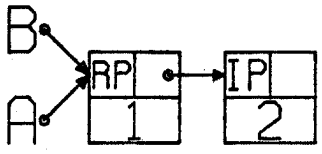
```
.A.B[2][3][4].C
```

leads to the value Y. Any references to old X, old X.A, old X.A.B, or old X.A.B[2], etc., are not affected by the modification to X. The modification is apparent only from the point of view of the variable X and not from the point of view of any other variable or any structure.

Modification is implemented by first copying the structure referenced by the target variable and then modifying that copy. In this process, ICL copies a minimal amount of memory.

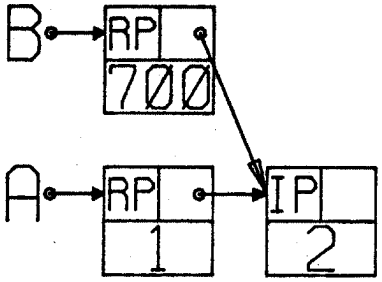
Examples:

Refering to a previous example, we had



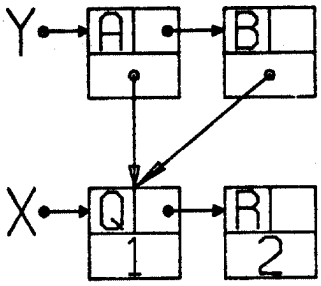
The modifying statement

B.REAL_PART:= 700; yields



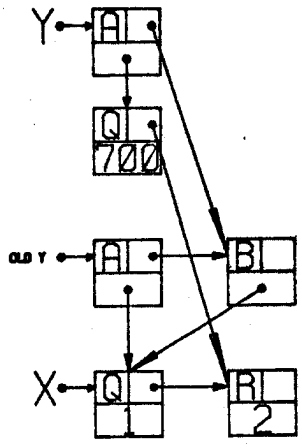
Note that B and A no longer reference the same memory element.

Given



the modifying statement

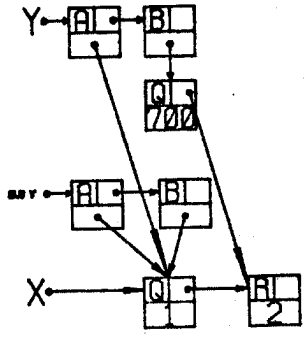
Y.A.Q:= 700; yields



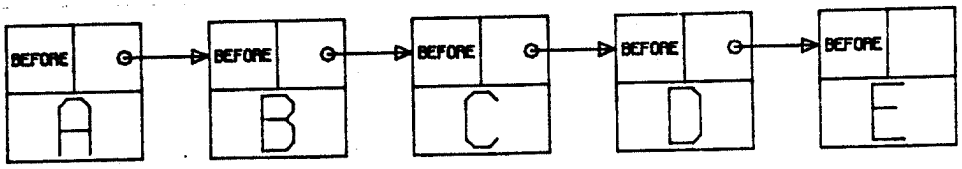
This modification costs two new nodes. However, if there are no references to old Y, the node immediately referenced by old Y will be freed by garbage collection. Note that if there is a reference to old Y, the nodes R and B are each shared by at least two references. New Y's B component still references the same structure it used to reference.

Given the same initial memory state, the modifying statement

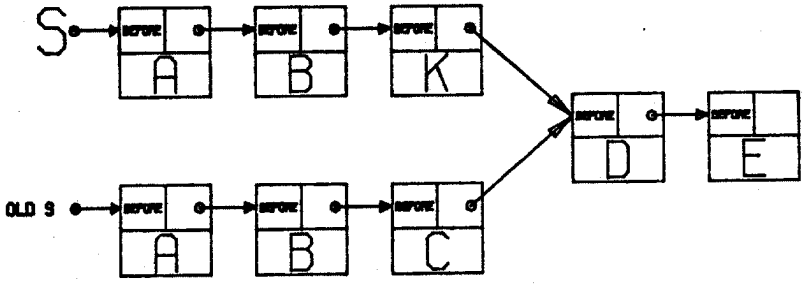
Y.B.Q:= 700; yields



Given the string S



the statement S[3]:= K; yields



The first three nodes in S are copied.

The remarkable savings achieved by the minimal copy are not the result of sharing portions of a record or string list, rather, it is the sharing of the elements which yields the major savings. For example, consider the string S given above. Let us assume that the elements A,B,C,D, and E each references some giant list structure. The modification to S still requires only three extra nodes. The structures A,B,D, and E are shared by S and old S. The minimal copy copies at most the top level structure of S.

In general, the user can predict the amount of copying ICL will perform given a modifying statement:

X.A:= <EXPR>; copies at most n nodes where
n= the number of components in
the record X.

X[I]:= <EXPR>; or

X[I-]:=<EXPR>; copies I nodes.

Multiple selections appearing on the lefthand side are also accountable:

X.A.B:= <EXPR>; copies at most n+m nodes where n= the number of components in the records X and where m= the number of components in the record X.A.

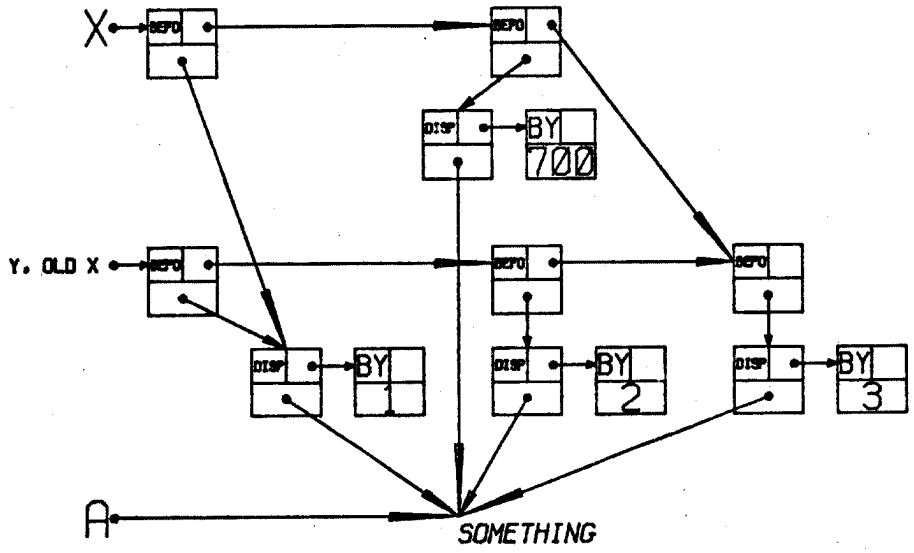
X[I][J]:=<EXPR>; copies I+J nodes.

The number of copied nodes is bounded above by the sum of the lengths of each relevant layer. The number of relevant layers equals the number of selection operators. The number of relevant layers is independent from the total number of layers making up the entire structure.

For another example of how shared data are modified, consider

```
X:={COLLECT [DISPLACE:A BY:I] FOR I FROM 1 TO 3;}
Y:=X;
...
X[2].BY:= 700;
```

This yields the following memory structure. In the following illustration, the term *BEFO* is used in place of *BEFORE* and *DISP* is used in place of *DISPLACE*.



The first and third elements of the strings X and Y are shared and A is shared by all.

Pointer Anchoring and Copying

We can infer several invariants from ICL's modification and copy policy:

1) ICL's data sharing is invisible.

2) Never is an existing structure modified.

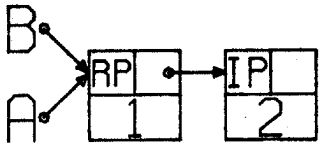
3) A modification is immediately apparent only from the point of view of exactly one variable.

4) A circular list structure never exists.

These invariants forbid many of the usual pointer operations. The @ operator is provided to enable the user to override these invariants. It is strongly suggested, however, that the user be careful about where he uses the @ operator. A strong dependence on the @ operator will inevitably lead to those popular bugs found in programs which are

written in languages where the user is free to manage pointers on his own.

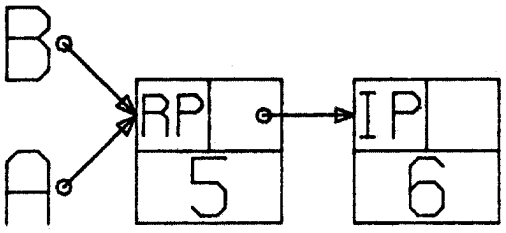
The @ operator turns a SOURCE into a TARGET by writing a given value directly over the memory element referenced by the SOURCE. The @ operator has the effect of making a modification apparent to *all* points of view. A modification made with the @ operator is said to be a global modification. For example, given the memory state



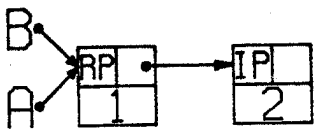
the anchored assignment

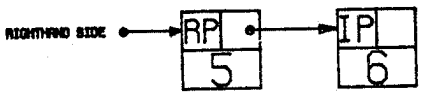
```
@(A):= [REAL_PART:5  IMAGINARY_PART:6] ;
```

yields the memory state



Both A and B and any other references to this unique memory location sense the change. The @ operation in the assignment happened as follows:



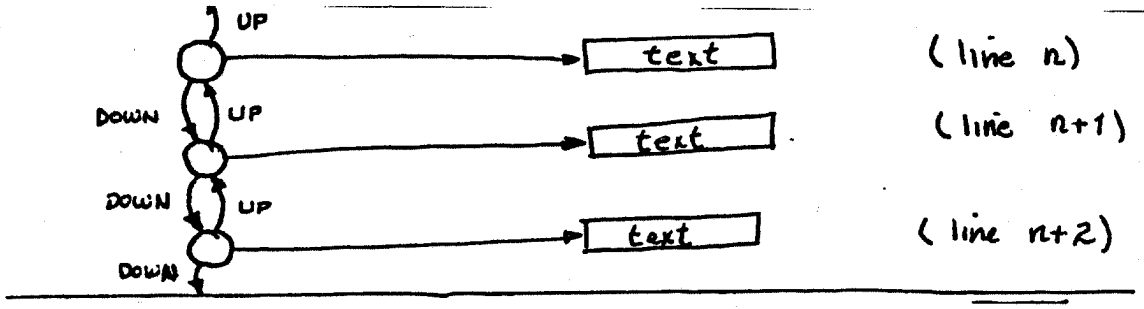


The @ operation writes the 5-node right over the 1-node.

The @ operator differs from conventional pointer manipulation in that it does not modify a reference in some structure, rather, it modifies the referenced structure itself. This is equivalent to modifying *all* references to the given structure. This scheme greatly contrasts ICL's default, single point of view modification. With the @ operator, the user can do any desired pointer manipulation.

Example - Line Editor:

Let us consider part of an editor for a line oriented terminal. We will want to move both up and down about the lines of the screen. The picture



may be represented in ICL by the type declarations

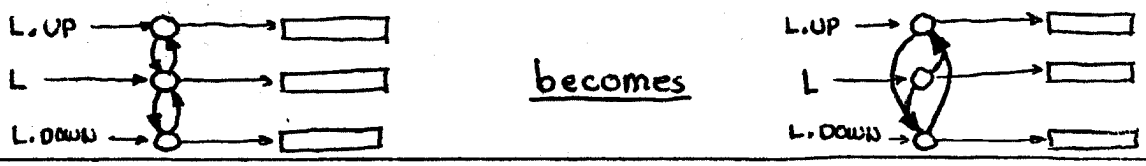
```
TYPE  LINE= [UP,DOWN:LINE  CHARS:LINE_OF_CHARACTERS] ;
      LINE_OF_CHARACTERS= { CHAR } ;
```

A LINE has an UP and a DOWN field which reference other LINEs. The CHARS field references the string of characters which reside on the line. The following procedure deletes a line by modifying its neighbors' UP and DOWN fields to bypass the given line:

```
DEFINE  DELETE(L:LINE):
```

```
        IF DEFINED(L.UP) THEN       @(L.UP).DOWN:= L.DOWN;  FI
        IF DEFINED(L.DOWN) THEN     @(L.DOWN).UP:= L.UP;    FI
    ENDDEFN
```

That is,



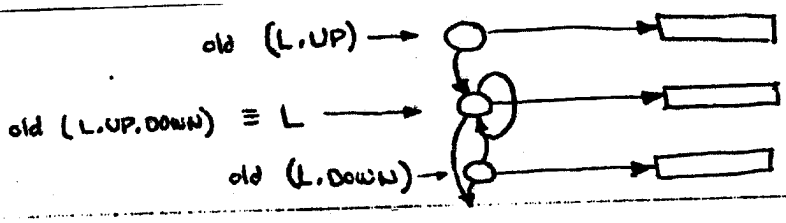
Note that DELETE's first sentence is

```
@(L.UP).DOWN:= L.DOWN;
```

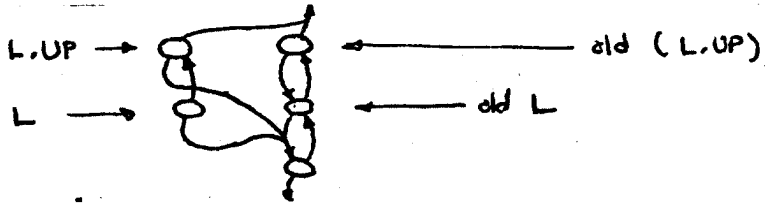
This modifies the node residing at L.UP. This is not equivalent to

1) `@(L.UP.DOWN):= L.DOWN;` or
2) `L.UP.DOWN:= L.DOWN;` or
3) `@(L).UP.DOWN:= L.DOWN;`.

The first modifies the node residing at L.UP.DOWN, which in this context is L itself. This would write the node residing at L.DOWN over the node at L.

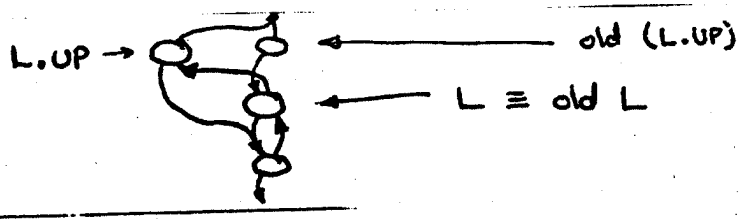


The second modifies the variable L and leaves the referenced structure unchanged.



From L's point of view, everything is the same except that the particular path .UP.DOWN is different. In other words, L references a copy of old L whose UP field is different. This UP field references a new node which is a copy of (old) L.UP whose DOWN field is different.

The third assignment modifies the node at L. This is like the second form except that the node at new L is written over the node at old L and the variable L is itself unchanged.



A procedure to insert line A before line B is

```
DEFINE  INSERT(A,B:LINE):
    IF DEFINED(B.DOWN) THEN      @(B.DOWN).UP:= A;  FI
    @(A).DOWN:=    B.DOWN;
    @(A).UP:=      B;
    @(B).DOWN:=    A;
  ENDDEFN
```

WARNING:

Any reference to an instance of type LINE will see any and all modifications made to that instance even if the modification was specified from the point of view of a different reference to the same instance. For example, with

```
LINE2:= LINE1;,
```

which sets LINE2 to reference what LINE1 references,

```
INSERT(LINE1,X);
```

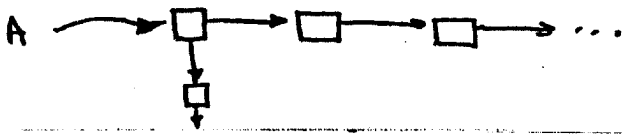
is equivalent to

```
INSERT(LINE2,X);
```

Because a LINE is modified with the @ operator, the apparent copy policy over assignment statements is lost for LINEs and for any structure which contains a reference to a LINE. We will return to this example after ICL's COPY operator is explained.

COPY

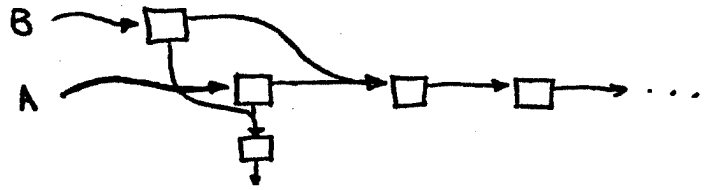
The rule ECOPY takes any pointer type and copies the referenced memory element to yield an identical structure which resides at a different memory location. For example, given the memory state



the sentence

```
B:= COPY(A);
```

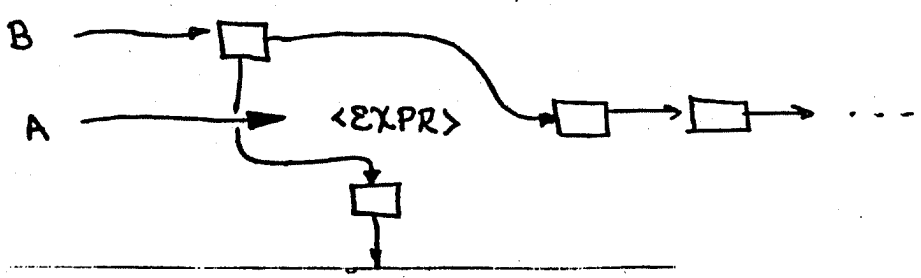
yields



COPY copies only one memory element. The essential fact is that A and B now reference distinct memory locations. Thus,

```
@(A):= <EXPR> ;
```

which creates the memory state



changes nothing from B's point of view. *The @ operator writes over only the single memory element directly referenced by A.*

ICL's ECOPY construct does not perform a complete copy, to the contrary, it copies only one memory element. While

```
LINE1:= COPY(LINE2);
```

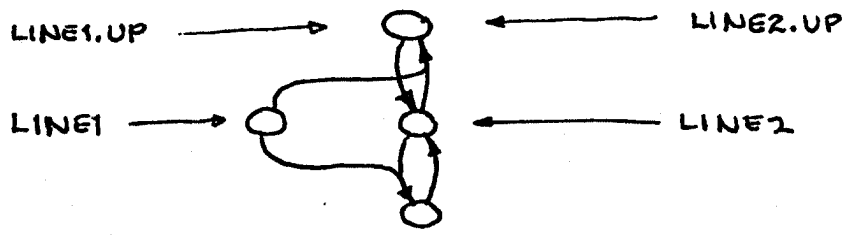
assures that

```
@(LINE1):= <EXPR>;
```

affects nothing from the point of view of LINE2, the assignment

```
@(LINE1.UP):= <EXPR>;
```

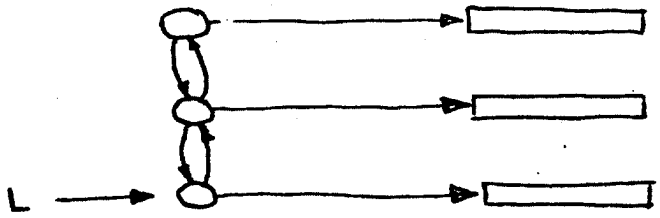
does make a change apparent from both LINE1 and LINE2's points of view. LINE1.UP and LINE2.UP are the same memory element:



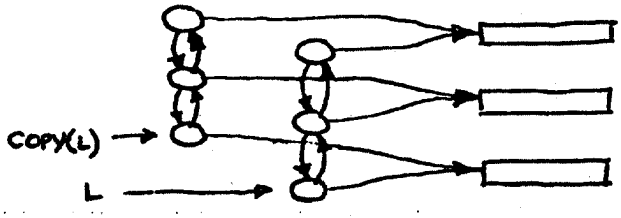
However, we can define a function which produces a complete copy of a given LINE:

```
DEFINE COPY(L:LINE)=LINE:
    IF DEFINED(L.UP) THEN   "copy the whole structure
                             referenced by L.UP "
        DO   L.UP:= COPY(L.UP);
             @(L.UP).DOWN:= L;
        GIVE   L
    ELSE  COPY!L) FI
ENDDEFN
```

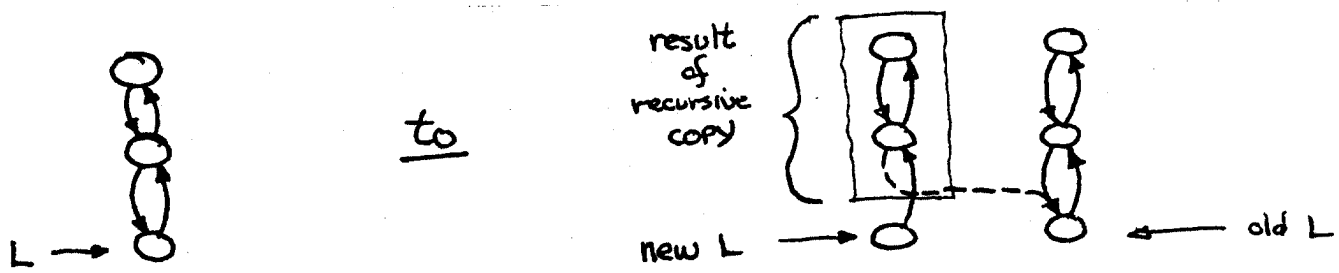
Recall that the <EXPR> COPY!L), which is used in the ELSE clause, is equivalent to COPY(L) in the absence of this function definition. Refer to the rule ECOPY. This function definition overrides the default meaning for COPY when applied to a LINE. However, COPY!L) has the original meaning of COPY(L). This function, given L:



produces



This function procedes by transforming



via the statement L.UP:=COPY(L.UP);. The statement @(L.UP).DOWN:=L; changes the dashed DOWN link to reference new L instead of old L. One can deduce from the very start that COPYing a LINE involves at least one @ operator: COPYing a LINE produces a circular structure, a two-way linked list. We know that without the @, a circular structure cannot be created.

If the given L has L.DOWN=NIL, then this COPY function yields a complete copy. If, on the other hand, L.DOWN is not NIL, then the copy will not be consistent because (COPY(L)).DOWN.UP will be L and *not* (COPY(L)). Therefore, the user might wish to make this copy function a subfunction of a new COPY function where the new COPY walks to the bottom of L and then performs the function presented above.

Note that L.CHARS is not copied. This is fine if L.CHARS or any of its substructure is never modified with the @ operator. However, if L.CHARS or any of its substructure is modified with @, a complete copy of a LINE must include copying the L.CHARS field:

```
DEFINE  COPY(L:LINE)=LINE:  BEGIN VAR C=CHAR;
     DO  L.CHARS:= {COLLECT C FOR C $E L.CHARS;}
     GIVE    the previous COPY function body
   END
  ENDDEFN
```

L is modified so that its CHARS field references a complete copy of the original L.CHARS. The identity-like

```
{COLLECT  X  FOR X $E S;}
```

forms a new string each of whose elements references the corresponding element in S. In other words, this COLLECT form produces a copy of S one level deep.

Now, if we write

```
LINE2:= COPY(LINE1);
```

LINE2 will remain unaffected by any operations performed upon LINE1. In summary, because instances of LINE are modified with the @ operator, ICL's apparent copy policy does not apply to LINEs. A new style of programming emerges when dealing with structures which are modified with @. Such structures appear to evolve independent from point of view. To obtain a completely distinct instance of such a structure whose further evolution is independent from the evolution of other instances, the user must explicitly specify a copy operation.

Example - Bounding Boxes and Property Lists:

This example differs from the previous example in that the @ modifications create no change in meaning. Rather, the @ operator is used to attach to some existing structure, properties, or values, any of which could be computed at any time. These values are characterized by being context-free: The value of a property does not depend on the point of view which references the structure having the property. The value depends only on structure below. An example of a context-free property for a picture is the picture's minimum-bounding-box. The minimum-bounding-box depends only on the picture and not on any references to the picture.

The advantage in storing context-free properties on existing structures is realized when the value of a property is requested more than once. The first request for a property may involve computation but further requests need not involve computation if

the first request stores the result of the computation. The savings is increased when the structure is shared by many different points of view.

Consider the following definition for the type RG, an IC mask:

```
TYPE  RG= EITHER
            POLY= POLYGON
            DISP= [DISPLACE:RG  BY:POINT]
            UNION= { RG }
          ENDOR;
```

This says that a region, RG, may be formed by specifying either a single polygon, a displacement upon an RG, or a union of RGs. The following form instances of RG, RG1 and RG2:

```
RG1:= { POLY1 ; [DISPLACE:POLY2  BY:3#4] };
RG2:= {COLLECT  [DISPLACE:RG1  BY:I#0]
                        FOR I FROM 1 TO 10;};
```

RG1 represents the union of POLY1 and a displaced POLY2. RG2 represents 10 copies of RG1, each of which is displaced in X by a different amount.

It turns out that the processing of RGs can be optimized by having some properties associated with each instance. The most popular property is known as an RG's minimum-bounding-box (*mbb*). We can define the type MRG to be the association of a box to an RG:

```
TYPE  MRG= [BODY:RG  VANISHING_MBB:BOX];
```

That is, an MRG is an RG along with its *mbb*. To make the *mbb* available at the appropriate places, let us redefine the type RG as follows:

```
TYPE  RG= EITHER
            POLY= POLYGON
            DISP= [DISPLACE:MRG  BY:POINT]
            UNION= { MRG }
          ENDOR;
```

All references to RGs have been replaced with references to MRGs. Thus, when processing an RG, the *mbb*s of its constituent parts are immediately available.

The disparity between RG and MRG is cosmetically removed by declaring

```
LET RG BECOME MRG BY [BODY:RG];
```

Any RG will automatically pass as an MRG. Now, any program text specifying an RG which worked under the old definition for the type RG will still work under the new definition for RG. RG's requirement that constituents be MRGs instead of RGs is resolved by the coercion.

Note that the coercion does not define the VANISHING_MBB field. We could, of course, change the coercion so that it calculates the *mbb* and sticks it in the VANISHING_MBB field. However, there is no real need to calculate the *mbb* until the *mbb* is actually sought. Once it is calculated, though, we should store the *mbb* in the VANISHING_MBB field so that it need not be

calculated again, e.g., when another reference to the MRG seeks the *mbb*.

The following function will actually obtain the *mbb* from an MRG.

```
DEFINE  MBB(M:MRG)=BOX:
  IF DEFINED(M.VANISHING_MBB) THEN M.VANISHING_MBB
  ELSE DO   @(M).VANISHING_MBB:=CALCULATE_MBB(M.RG);
       GIVE   M.VANISHING_MBB   FI
ENDDEFN
```

This function first sees if the VANISHING_MBB field is already defined. If it is, this field is immediately returned and that is all. Otherwise, this function calculates the *mbb* by calling CALCULATE_MBB, and via the @(..) operator, the function MBB modifies the actual memory location referenced by M to include the box. Now *any further* references to that MRG see the defined VANISHING_MBB field. Note that if the @(...) were not used, only the local variable M would be modified and so upon leaving the function, the calculated box would not be permanently associated to the given MRG. The assignment

```
@(M).VANISHING_MBB := <EXPR> ;
```

may be paraphrased as

" From the point of view of the *structure referenced by* M, the VANISHING_MBB field is defined to be <EXPR>. "

In contrast, the assignment

M.VANISHING_MBB := <EXPR> ;

says

" From the point of view of the *variable* M, the VANISHING_MBB field is defined to be <EXPR>. "

If M is an MRG, then

M.BODY is the RG and

MBB(M) is the *mbb*

The awkward name VANISHING_MBB was chosen to discourage direct access to that component. For example, if the user forgets that the *mbb* must be accessed via the function MBB, e.g., he writes

M.MBB

to fetch M's *mbb*, he will receive a datatype error. However, referring to the section on unary operators, the notation

M\MBB is equivalent to MBB(M).

Thus, the \ by itself appears to play the role of a generalized selection operator.

By declaring

LET MRG BECOME RG BY MRG.BODY ;

the user need not specify the .BODY on an MRG to obtain its RG. M by itself passes as an RG. In fact, because we have the coercions between MRG and RG going in both directions, instances of the two types are completely interchangeable.

Note that MBB applied to an RG still yields the RG's *mbb*. The RG will be coerced to an MRG before calling MBB. However, the *mbb* tacked onto the MRG by the function MBB will not be attached to the RG. The MRG passed to MBB is lost upon return from MBB; even though the RG may still be referenced, the MRG created by the coercion ceases to be referenced.

It is advantageous to declare variables to be of type MRG rather than to be of type RG. For example, the RG

```
{ A ; [DISPLACED:A  BY:10#10] }
```

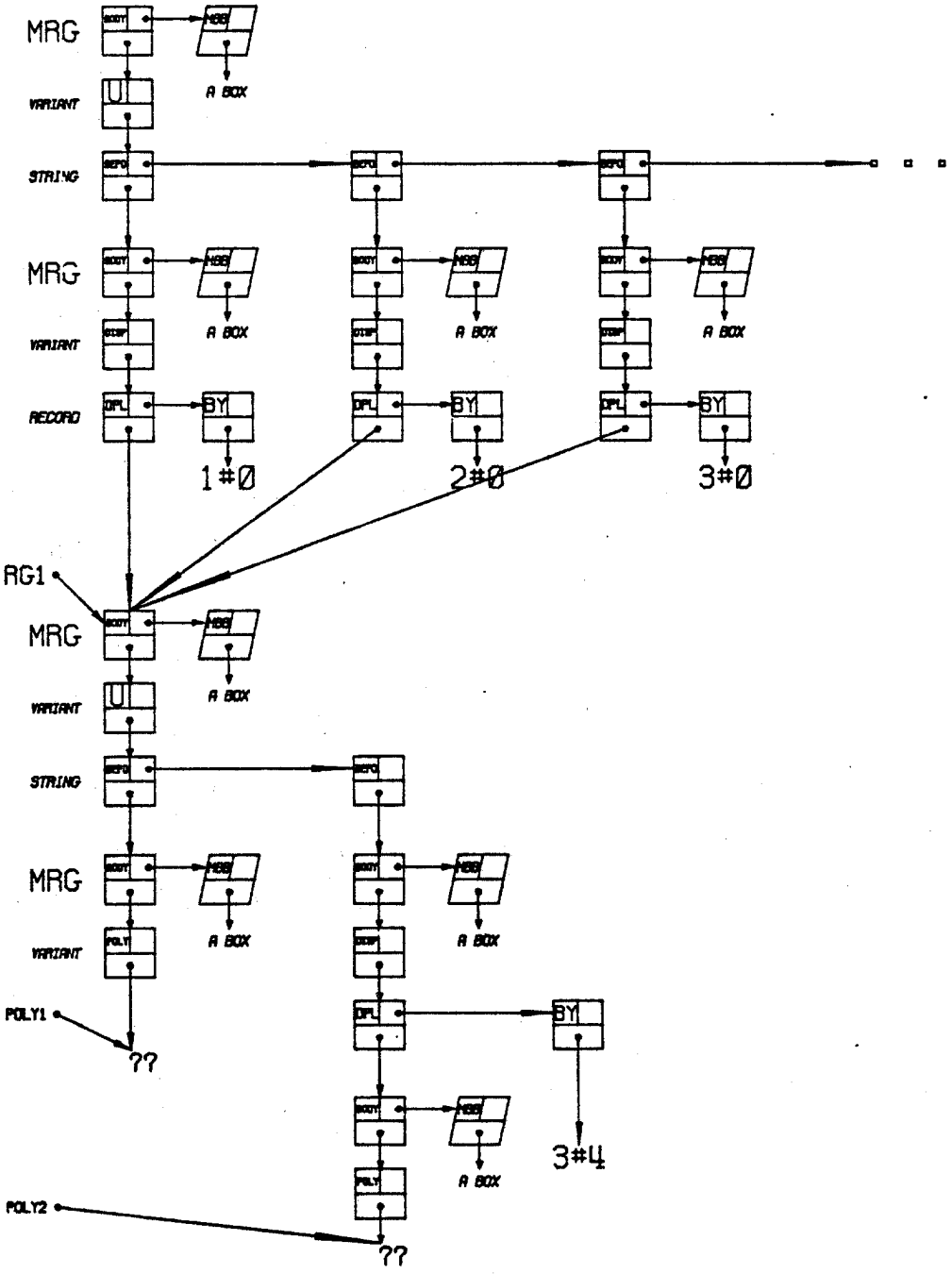
references the variable A twice. If A is of type RG, the coercion from RG to MRG will be applied twice and, in fact, the *mbb* for A will ultimately be calculated twice. However, if A is of type MRG, no coercion will be applied and the *mbb* of A will be (or has already been) calculated only once. It is similarly advantageous to use the type MRG in place of the type RG when declaring new types which reference IC-masks. In fact, the type RG should be forgotten altogether except in those few functions which examine MRGs.

An MRG may be defined to include more properties, e.g.:

```
TYPE  MRG=[BODY:RG  VANISHING_MBB:BOX
                    VANISHING_RECTS: RECTANGLES
                    DESIGN_RULES_OK: DESIGN_STATUS
                    SCHEMATIC: CIRCUIT_DIAGRAM];
```

Here we have properties including the representation of an RG in terms of rectangles, a design rule status, and a schematic. Each of these properties can be computed from an RG and these properties are independent from the points of view which reference an MRG or an RG. Accessing each property should be done via a function like MBB which manages one component in the MRG record. Such access functions manage the retrieval and storage of individual properties. It is conceivable that an access function might be written which conditionally stores its computed values. The conditions might depend on global variables which tell how much memory is available or they might depend on the state in which the RG resides, e.g., the DISP state has a trivial *mbb* calculation whereas the UNION state has a more expensive *mbb* calculation.

This scheme for implementing properties has the advantage that shared data implies shared computation. Let us assume that the variables RG1 and RG2 were declared to be of type MRG. Consider that the value in RG2 is represented by



In this illustration, the following substitutions have been made:

BEFO is used in place of *BEFORE*

DPL is used in place of *DISPLACE*

U is used in place of *UNION*, and

MBB is used in place of *VANISHING_MBB*.

The slanted elements are the VANISHING_MBB components of MRGs. These do not exist until MBB is called. The *mbb* at RG1 will be calculated only once even though it will be requested 10 times from the point of view of RG2. Note also that RG1 will find its *mbb* already calculated if RG2's *mbb* was previously sought. Similarly, if the *mbb* of RG1 is requested first, it will not be recalculated when computing RG2's *mbb*.

Finally, note that the overhead from introducing MRGs in place of RGs is one memory element per instance in the absence of any properties. Each *existing* property costs an additional overhead of one memory element. The number of declared properties is irrelevant.

Disasters

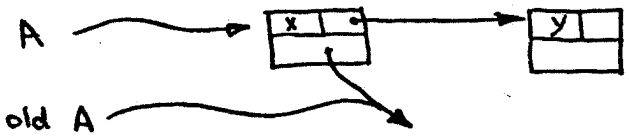
The @ operator is an untamed animal. Some very innocent actions can cause bizarre effects. This section documents some disasters which can come with the @ operator.

Example 1:

First, let us consider the non-anchored assignment

```
A := [X:A  Y:B];
```

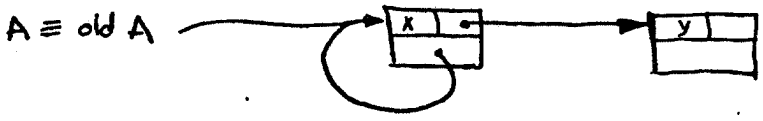
This assignment modifies the variable A so that it points to a new record, [X:A Y:B], and this new record's X component references what A *used* to reference. What A now references and what A used to reference are distinct memory locations.



In contrast, the anchored assignment

```
@(A) := [X:A  Y:B];
```

creates a circular structure and does not modify the variable A.

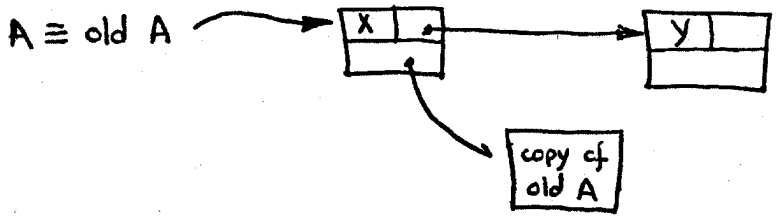


This assignment writes the new record [X:A Y:B] *over the location* referenced by both the variable A and the record's X component.

The location referenced by A and the location referenced by the record's X component are precisely the location now occupied by the record itself. However,

```
@(A) := [X:COPY(A)  Y:B];
```

modifies the structure referenced by A to be a record whose X component references a copy of what A used to reference.



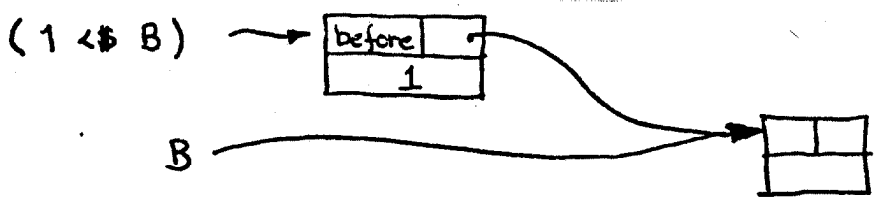
The COPY is used to avoid circularity.

Example 2:

The assignment

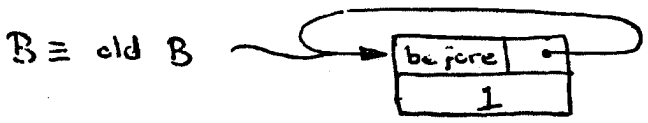
```
@(B):= 1 <$ B;
```

would seem to modify the node at B to be what B used to be with a 1 tacked on the front. However, this will not be the case. After the evaluation of the righthand side of the assignment, we get



Finally, the @ operator writes the 1-node over the node referenced

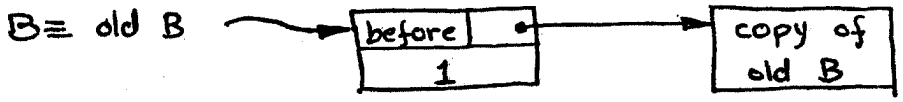
by B, yielding



B now references an infinitely long string of 1s. In contrast,

```
@(B) := 1 <$ COPY(B);
```

does the expected.



Example 3:

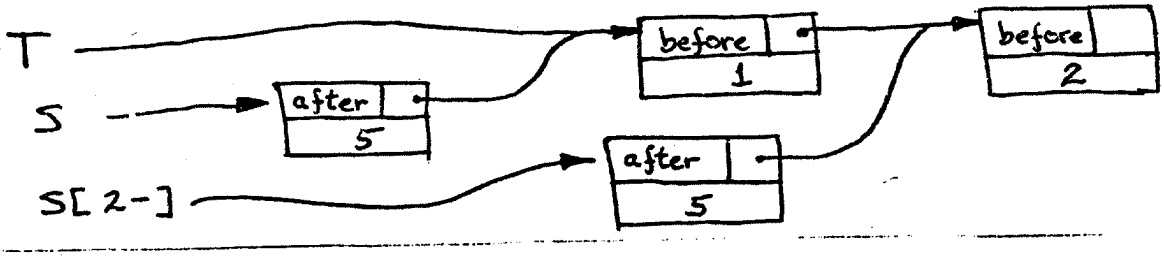
The <EXPR>

```
S[2-]
```

is not necessarily a tail of S in the sense that one could get from S to S[2-] by tracing nodes in memory. S[2-] only appears to be a tail of S. For example, if we specify

```
T:= {1;2};
S:= T $> 5;
```

S, T, and S[2-] will reference the memory structures



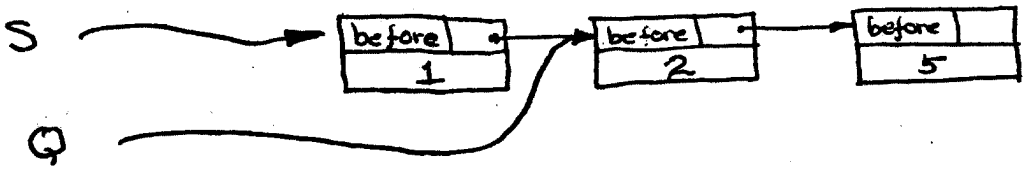
The node referenced by S[2-] is not on a path starting at S. Thus, for example,

```
@(S[2-]) := {10;11} ;
```

does not change anything from S's point of view. However, if we write

```
S:=REFRESH(S);
Q:= S[2-];
```

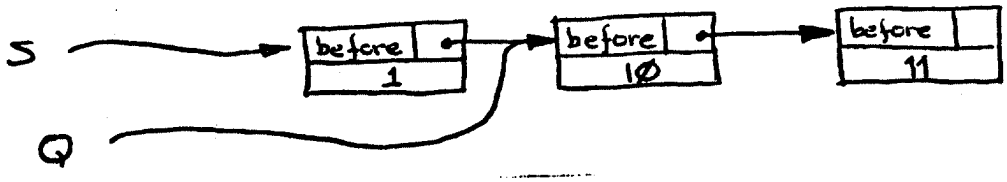
then we get



Thus,

```
@(S[2-]) := {10;11};
```

leaves



as expected.

Strings act like LISP's lists only when the string is refreshed. In summary, do not use @s on the tails of non-refreshed strings. Recall that a refreshed string is formed by

REFRESH(*any string*) or by

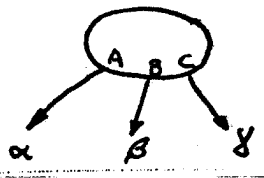
element <$ *refreshed string* or by

the string generation rules, e.g.,

{ ... ; *element* ; ... ; COLLECT *element* <QUANT> ; ... }.

Consolation:

Even though ICL represents records as linked lists like strings, record lists maintain an important property which is absent from strings: Any non-first node in a record list is not the first node in another record list. In other words, no record list is a proper tail of another record list. Thus, because the @ operator overwrites only the first node in a record list, the property given above guarantees that @ cannot overwrite a non-first node in any record list. The essential invariant is that the user can think of a record as being an indistinguishable unit of memory, e.g.,



all of which or none of which can be clobbered with an @.

Carry-overs from <EXPR>s

The following <SS> forms are carry overs from <EXPR>s. These rules are copied from the corresponding <EXPR> forms by merely substituting <SS> for <EXPR> and DO for GIVE in the appropriate places. Any *type* or *PASS3* requirements imposed on the <EXPR>s which have been replaced by <SS>s are simply to be ignored.

The IF-THEN-ELSE

EBIF: <SS> ::= <BIF_1> <$EXPR_2$> THEN <SS_3> ELSE <SS_4> FI

SBIF: <SS> ::= <BIF_1> <$EXPR_2$> THEN <SS_3> FI

where

BIF1: <BIF> ::= IF

BIF2: <BIF> ::= <BIF_{k1}> <$EXPR_{k2}$> THEN <SS_{k3}> EF

Type Requirements <$EXPR_2$> = BOOL = <$EXPR_{k2}$>

PASS3 Requirements <$EXPR_2$> = SOURCE = <$EXPR_{k2}$>

Meaning

Identical to the EBIF rule in the section for <EXPR>s. Note however that an ELSE clause is optional in the <SS> IF-THEN-ELSE. The extra rule, SBIF, allows a <SS> to be built without an ELSE.

Example:

```
IF A=B THEN I:=5; FI
```

If A=B, then I is assigned the value 5, otherwise, nothing is done.

```
IF A=B THEN I:=5; EF A<B THEN J:=20; FI
```

If A=B, then I is assigned 5, otherwise if A<B, then J is assigned 20, otherwise, nothing is done. The form

```
IF  A=B  THEN  I:=5; ELSE I:=23; FI
```

is equivalent to

```
I:= IF A=B THEN 5 ELSE 23 FI
```

The Scalar CASE form

ECASEE: <SS> ::= CASE <EXPR$_1$> OF <SSV$_2$>

where

EVCASE: <SSV> ::= <ID$_1$> : <SS$_2$> ENDCASE

EVCASB: <SSV> ::= <ID$_{k1}$> : <SS$_{k1}$> <SSV$_{k3}$>

Type and PASS3 Requirements

Refer to the <EXPR> ECASEE rule. <EXPR$_1$> must be a *scalar* type.

Meaning

Refer to the <EXPR> ECASEE rule. However, where the debugging package would be entered in the <EXPR> rule, nothing happens in the <SS> rule. That is, if <EXPR$_1$> yields none of the <ID$_1$> or <ID$_{k1}$>s in <SSV$_2$> and if there is no ELSE clause, the <SS>-CASE performs no action.

Examples:

```
CASE  A_COLOR  OF
    BLUE:  I:=5;
    RED:   I:=20;       ENDCASE
```

If the variable A_COLOR is BLUE, I is assigned 5. If A_COLOR is RED, I is assigned 20. If A_COLOR is neither RED or BLUE then I is unchanged. An equivalent <EXPR>-CASE form is:

```
I:= CASE   A_COLOR  OF
       BLUE: 5
       RED:  20
       ELSE: I          ENDCASE
```

The Variant CASE form

ECASE: <SS> ::= CASE <ID_1> OF <SSV_2>

where <SSV> is as defined above.

Type and PASS3 Requirements

Refer to the <EXPR> ECASE rule. <ID_1> must be a variable of some variant type.

Meaning

Refer to the <EXPR> ECASE rule. However, as in the <SS> scalar case form, the absence of an ELSE clause may render the <SS>-case a no-op; it won't cause the debugging package to be entered.

Example:

```
DEFINE  WRITE(L:LISP_ELEMENT):
  CASE  L  OF
    ATOM:    WRITE(L);
    INTEGER_NUMBER:  WRITE(L);
    FLOATING_NUMBER: WRITE(L);
    CONS_PAIR:       WRITE('('); WRITE(L.CAR);
                     WRITE(' . '); WRITE(L.CDR);
```

```
            WRITE(')');
        ENDCASE
    ENDDEFN
```

This defines WRITE of LISP_ELEMENT to print out a LISP_ELEMENT in the dot notation. Note that the WRITE functions named in the first three case-clauses are WRITE of QS, INT, and REAL respectively. Recursion occurs only in the CONS_PAIR clause; L.CAR and L.CDR are of type LISP_ELEMENT.

Note that the following CASE form:

```
CASE L OF
    CONS_PAIR:    L.CAR:=L.CDR;
ENDCASE
```

leaves L unmodified upon completion. Referring to the <EXPR> ECASE rule, note that the case-variable, L, always appears unchanged by anything within a CASE form.

The HOLDING Form

HOLDIT: <SS> ::= HOLDING <ASN$_1$> DO <SS$_1$> ENDHOLD

Meaning

Refer to the <EXPR> HOLDING form.

Example:

When processing pictures, it is useful to have a global variable defining the "current" orientation and to have plotting procedures which reference that global variable for the purpose of placing the given picture on the screen. Suppose the type PICTURE is defined

by

```
TYPE  PICTURE= EITHER
                 SIMPLE= POLYGON
                 DISPLACED= [P:PICTURE  BY:POINT]
                 UNION=  { PICTURE }
               ENDOR;
```

That is, a PICTURE may be formed by unions and displacements upon POLYGONs. Let us declare

```
VAR   POSITION=POINT;
```

so that POSITION is the global variable representing orientation. Assuming the existence of a procedure to plot POLYGONs at the orientation specified in POSITION, the following procedure will plot PICTUREs:

```
DEFINE  PLOT(X:PICTURE):
  CASE  X  OF
      SIMPLE:  PLOT_POLY(X);
      DISPLACED:  HOLDING POSITION:=POSITION+X.BY;
                      DO  PLOT(X.P);   ENDHOLD
      UNION:   BEGIN  VAR V=PICTURE;
                 DO  PLOT(V);  FOR V $E X;
               END
  ENDCASE
ENDDEFN
```

The DISPLACED case-clause modifies the orientation, POSITION, *for and only for* the plotting of the "displaced" picture. The global variable POSITION is being used in a recursive manner because there

The Arithmetic FOR Quantifier

This quantifier corresponds to FORTRAN's DO-loop.

AFORGO: <QUANT> ::= $\langle AFOR_1 \rangle$;

where

AFORID: <AFOR> ::= FOR $\langle ID_1 \rangle$

AFORFR: <AFOR> ::= $\langle AFOR_1 \rangle$ FROM $\langle EXPR_2 \rangle$

AFORTO: <AFOR> ::= $\langle AFOR_1 \rangle$ TO $\langle EXPR_2 \rangle$

AFORBY: <AFOR> ::= $\langle AFOR_1 \rangle$ BY $\langle EXPR_2 \rangle$

AFORIN: <AFOR> ::= $\langle AFOR_1 \rangle$ IN $\langle EXPR_2 \rangle$

AFORIS: <AFOR> ::= $\langle AFOR_1 \rangle$ IN* $\langle EXPR_2 \rangle$

Informally, a <QUANT> may be formed by

FOR <ID>

followed by a sequence of the clauses

FROM <EXPR>

TO <EXPR>

BY <EXPR>

IN <EXPR>

IN* <EXPR>

followed finally by a semicolon.

Syntax Requirements

Some clauses cannot appear together and some clauses require others.

1) Each clause may appear at most once.

2) IN, IN*, and BY are mutually exclusive.

3) TO is required in the absence of BY.

Looping with <SS>s

SSQ: <SS> ::= DO <SS_1> <$QUANT_2$>

SSQ: <SS> ::= <$QUANT_2$> DO <SS_1> END

Meaning

Execute <SS_1> once for each iteration caused by <$QUANT_2$>

Examples:

```
DO  WRITE(I); FOR I FROM 1 TO 9;
   (---SS----)(------QUANT------)
```

prints 123456789.

```
FOR I FROM 1 TO 9; DO WRITE(I); END
```

does exactly the same.

Function Calling

SSCALP: <SS> ::= <ID_1> <$ARGS_2$> ;

SSICAL: <SS> ::= <ID_1> ;

where <ARGS> is as defined in the <EXPR> function call rule, ECALLP.

Type Requirements

For the first rule, SSCALP, there must be a declared function whose name is <ID_1>, which produces no value, and whose input parameter types sequentially match the types of the <EXPR>s in <$ARGS_2$>. For the second rule, SSICAL, there must be a declared function whose name is <ID_1> and which has no input or output parameters.

PASS3 Requirements

The <EXPR>s in <ARGS$_2$> must be SOURCEs.

Meaning

Evaluate each <EXPR> in <ARGS$_2$> in order of specification and then call the appropriate function, <ID$_1$>.

Example:

```
DEFINE  TAB:   WRITE('      ');   ENDDEFN
```

defines TAB to be a procedure which prints a tab.

```
TAB;
```

invokes TAB and thus prints a tab.

A Sequence of <SS>s

Unlike the programming language PASCAL, sequences of statements need not be enclosed within BEGIN-ENDs. Also, in ICL, semicolons do not separate <SS>s. Semicolons are terminators for various independent constructs like the assignment statement and the procedure call. Statements in ICL are separated by blanks or by nothing at all.

SSSS: <SS> ::= <SS_1> <SS_2>

Meaning

Evaluate <SS_1>. Then evaluate <SS_2>.

Example:

```
I:=0; I:=I+3; I:=I*2;
```

leaves I containing a 6.

Quantifiers - Loop Generators: <QUANT>

Quantifiers are those linguistic forms in ICL which cause looping. Aside from looping via recursion, all looping in ICL is expressed via quantifiers.

We can characterize the meaning of <QUANT> by first noting that all of ICL's rules which incorporate <QUANT>s easily transform into the canonical form:

DO <SS> <QUANT>

Some action is performed repeatedly as dictated by <QUANT>, whether that action be accumulating a sum, forming a string, or performing some arbitrary action.

DO <SS> <QUANT>

is implemented by the program:

```
        prepare for the first iteration
LOOP:   <SS>
        prepare for the next iteration
        GOTO LOOP
EXIT:   --
```

where the two preparations have the option of branching to the EXIT lable, thus terminating the loop.

Primitive Quantifiers

The following are ICL's primitive <QUANT>s. In the next section, we will see ways to combine <QUANT>s to come up with more complex quantifiers.

The WHILE Quantifier

QWHIL: <QUANT> ::= WHILE <EXPR$_1$> ;

Type Requirements <EXPR$_1$> = BOOL

PASS3 Requirements <EXPR$_1$> = SOURCE

Meaning

Before each iteration, evaluate <EXPR$_1$> and exit as soon as <EXPR> yields FALSE. The WHILE quantifier may cause zero iterations!

Examples:

```
DO WRITE('x'); WHILE FALSE;
```

is a no-op.

```
DO WRITE('x'); WHILE TRUE;
```

is an infinite loop.

```
{ COLLECT C WHILE (C:=TTYCIN;) <> CR; }
```

forms a string of characters taken from the TTY. The function TTYCIN yields each character typed in at the TTY. The resulting string includes all characters up to but not including the first carriage-return, assuming that the variable CR contains the carriage-return character. Note that since the WHILE-<EXPR>

```
(C:=TTYCIN;) <> CR
```

evaluates before each iteration, the CHAR variable C contains a new input character upon each iteration. The form

(C:=TTYCIN;)

sets C to the input character and yields this character as its value, referring to the rule SETQX in the section *Embedding <SS>s in <EXPR>s*. Upon leaving this string <EXPR>, C contains a carriage return.

The UNTIL Quantifier

QUNTL: <QUANT> ::= UNTIL <EXPR$_1$> ;

Type Requirements <EXPR$_1$> = BOOL

PASS3 Requirements <EXPR$_1$> = SOURCE

Meaning

After each iteration, evaluate <EXPR$_1$> and exit as soon as <EXPR$_1$> yields TRUE. The UNTIL quantifier causes at least one iteration!

Examples:

DO WRITE('x'); UNTIL TRUE;

writes one x.

DO WRITE('x'); UNTIL FALSE;

is an infinite loop.

{ COLLECT C UNTIL (C:=TTYCIN;) = CR; }

forms a string of characters taken from the TTY. The first character in the string is *not* from the TTY, however; the first character is whatever C contained upon entry to this string <EXPR>. This string includes all characters up to and *including* the first

carriage-return, assuming that the variable CR contains a carriage-return. Upon leaving this <EXPR>, C contains a carriage-return.

The REPEAT Quantifier

REPET: <QUANT> ::= REPEAT <EXPR$_1$> ;

Type Requirements <EXPR$_1$> = INT

PASS3 Requirements <EXPR$_1$> = SOURCE

Meaning

Cause <EXPR$_1$> iterations. If <EXPR$_1$> is zero or less, cause no iterations.

Examples:

```
DO WRITE('x');  REPEAT 50;
```

writes 50 x's.

```
{ COLLECT TTYCIN  REPEAT 80;}
```

forms a string of 80 characters taken from the TTY.

The Arithmetic FOR Quantifier

This quantifier corresponds to FORTRAN's DO-loop.

AFORGO: <QUANT> ::= $\langle AFOR_1 \rangle$;

where

AFORID: <AFOR> ::= FOR $\langle ID_1 \rangle$

AFORFR: <AFOR> ::= $\langle AFOR_1 \rangle$ FROM $\langle EXPR_2 \rangle$

AFORTO: <AFOR> ::= $\langle AFOR_1 \rangle$ TO $\langle EXPR_2 \rangle$

AFORBY: <AFOR> ::= $\langle AFOR_1 \rangle$ BY $\langle EXPR_2 \rangle$

AFORIN: <AFOR> ::= $\langle AFOR_1 \rangle$ IN $\langle EXPR_2 \rangle$

AFORIS: <AFOR> ::= $\langle AFOR_1 \rangle$ IN* $\langle EXPR_2 \rangle$

Informally, a <QUANT> may be formed by

FOR <ID>

followed by a sequence of the clauses

FROM <EXPR>

TO <EXPR>

BY <EXPR>

IN <EXPR>

IN* <EXPR>

followed finally by a semicolon.

Syntax Requirements

Some clauses cannot appear together and some clauses require others.

1) Each clause may appear at most once.

2) IN, IN*, and BY are mutually exclusive.

3) TO is required in the absence of BY.

Type Requirements

<ID_1> must be a variable. <ID_1> and the <$EXPR_2$>s must either all be INTeger or all be REAL.

PASS3 Requirements All the <$EXPR_2$>s = SOURCE

Meaning

Set the loop variable, <ID_1>, for each iteration as directed by the specified clauses. Each clause has its own meaning:

FROM <EXPR>

sets the loop-variable to the value of <EXPR> before the first iteration. In the absence of FROM, the value of the loop-variable is whatever it was upon entrance to the loop. That is, the absence of FROM is equivalent to specifying

FROM <ID_1>

The TO clause,

TO <$EXPR_2$>

specifies that the loop is to terminate when the loop variable *exceeds* the value of <$EXPR_2$>. Note that if the increment is negative, *exceed* means *less than*. In the absence of the TO clause, the loop is infinite.

The BY, IN, and IN* clauses specify an increment. In the absence of these clauses, the increment is +1 or -1, depending on which of the FROM and TO <EXPR>s is greater.

BY <$EXPR_2$>

specifies that the increment is to be the value of <EXPR$_2$>. Before each non-first iteration, the loop variable is incremented by the value of <EXPR$_2$>. <EXPR$_2$> may be negative.

IN <EXPR$_2$>

specifies the increment (TO-FROM)/IN. That is, IN specifies the *number of iterations*. The increment is chosen to divide the FROM-TO interval evenly into <EXPR$_2$> intervals. The loop variable is set to the initial endpoint of each interval, e.g.,

```
FOR R FROM 0.0 TO 1.0 IN 4;
```

sets R to the values

0.0, 0.25, 0.5, 0.75, *but not* to 1.0

However,

IN* <EXPR$_2$>

specifies the same increment as the IN <EXPR> but the number of iterations is <EXPR$_2$>+1, not <EXPR$_2$>. The extra iteration sets the loop variable to the terminal endpoint of the last interval, e.g.,

```
FOR R FROM 0.0 TO 1.0 IN* 4;
```

sets R to the values

0.0, 0.25, 0.5, 0.75, *and* 1.0.

Note that if the loop variable is INTeger, the increment

(TO-FROM)/IN

is calculated using the integer divide, so

```
FOR I FROM 1 TO 10 IN* 3;
```

yields the sequence 1,4,7,10 and

```
FOR I FROM 1 TO 9 IN* 3;
```

yields the sequence 1,3,5,7.

The arithmetic FOR quantifier evaluates each <EXPR$_2$> *once*, before entering the loop. The arithmetic FOR quantifier also resets the loop variable for each iteration, ignoring its current value, e.g.,

```
DO WRITE(I); I:=20; FOR I FROM 1 TO 3;
```

writes the numbers 1,2, and 3. Also, the FOR quantifier *does not* increment the loop variable after the final iteration, and it does not reset it. Thus,

```
DO WRITE(I); FOR I FROM 1 TO 3;
```

leaves the variable I containing the value 3, not 4.

```
DO WRITE(I); I:=20; FOR I FROM 1 TO 3;
```

writes the numbers 1,2, and 3 and leaves I containing a 20 upon exit.

Examples:

```
DO  WRITE(I);WRITE(' '); FOR I FROM 5 TO 10;
```

writes 5 6 7 8 9 10.

```
DO  WRITE(I);WRITE(' '); FOR I FROM 10 TO 5;
```

writes 10 9 8 7 6 5.

```
I:=10;
DO WRITE(I); FOR I TO 5;
```

does the same.

```
DO WRITE(I); FOR I FROM 10 TO 5 BY 1;
```

writes the number 10.

```
DO WRITE(R);WRITE(' '); FOR R FROM 1.0 TO 0.0 IN 4;
```

writes

```
1.0 0.75 0.5 0.25
```

The following form generates a string of points:

```
{COLLECT COS(T)#SIN(T) FOR T FROM 0 TO 2*3.14 IN N;}
```

This string of points represents an N-gon without duplicating the first point.

```
{COLLECT COS(T)#SIN(T) FOR T FROM 0 TO 2*3.14 IN* N;}
```

makes an N-gon where the first point is duplicated at the end.

The Selection FOR Quantifier

The selection FOR quantifier is perhaps the main workhorse in ICL. As implied by its name, the selection FOR quantifier performs selection, mainly on strings. Its most popular use is for iterating thru the elements in a string. As we shall see, the FOR quantifier supports iterations thru the elements of elements of strings or thru the elements of string components of records or thru the elements of strings of records of strings, etc. In addition, the FOR quantifier can iterate thru a string by setting a *sequence* of variables to consecutive elements in the string. The user can even specify that the sequence of variables be allowed to *wrap around* back to the beginning of the string.

The FOR quantifier is basically an assignment statement. However, unlike the regular assignment statement, the FOR quantifier is free to cause looping. Within a FOR quantifier, the notion of TARGET is extended to include a new class of TARGET-like entities: looping-TARGETs. Looping-TARGETs include a new TARGET which is formed by the *string generation* rule, e.g., {I;J;...}. We shall formally introduce the class of looping-TARGETs after we present the linguistic contraction which abbreviates the FOR quantifier for the most common uses.

QFORE: <QUANT> ::= FOR <EXPR$_1$> $E <EXPR$_2$> ;

The $E reads as *an element of*.

Type Requirements

<EXPR$_2$> = a string of some type and

<EXPR$_1$> = that type of which <EXPR$_2$> is a string.

PASS3 Requirements

<EXPR$_2$> = SOURCE and

<EXPR$_1$> = TARGET or looping-TARGET

Meaning

The following describes the meaning only for those cases where <EXPR$_1$> is a TARGET and not a looping-TARGET: For each element in the string <EXPR$_2$>, feed that element to the TARGET <EXPR$_1$> and cause one iteration. The number of iterations is therefore equal to the length of the string <EXPR$_2$>.

Examples:

```
DO WRITE(I);WRITE(' '); FOR I $E {1;5;20;-3};
```

writes 1 5 20 -3.

```
+ I FOR I $E {1;5;20;-3};
```

yields the value 23, the sum of the elements in the specified string. (Refer to the section *Looping with <BOP>s*).

```
{ COLLECT I+1 FOR I $E S;}
```

yields a string identical to S except that each element is incremented.

```
MIN I FOR I $E S;
```

yields the minimum value in S. Recall that MIN is a <BOP>. If BOX is defined by

```
TYPE BOX = [LOW,HIGH: POINT];
```

where LOW refers to the lower lefthand corner and HIGH refers to the upper righthand corner, then

```
[LOW:  MIN P FOR P $E S;
 HIGH: MAX P FOR P $E S; ]
```

yields the minimum bounding box for an arbitrary string of POINTs, S.

The $E FOR-quantifer presented above is a special case of the more general, $C FOR-quantifier. In general, we can translate

FOR <EXPR$_1$> $E <EXPR$_2$> ; into

FOR { <EXPR$_1$> } $C <EXPR$_2$> ;

$E reads as *an element of* and $C reads as *contained in*.

QFORC: <QUANT> ::= FOR <EXPR$_1$> $C <EXPR$_2$> ;

Type Requirements <EXPR$_1$> = <EXPR$_2$>

PASS3 Requirements <EXPR$_2$> = SOURCE and

<EXPR$_1$> = TARGET or looping-TARGET.

Meaning

Feed the value of <EXPR$_2$> to the TARGET or looping-TARGET <EXPR$_1$>. If <EXPR$_1$> is a TARGET, act as a simple assignment and cause exactly one iteration. If, on the other hand, <EXPR$_1$> is a looping-TARGET, then set variables and cause looping *as directed by the* looping-TARGET.

What is a Looping-TARGET

A looping-TARGET is

1) any TARGET, or
2) any *string* of looping-TARGETs, or
3) any *record* of looping-TARGETs

In this context, we are viewing the set of looping-TARGETs as including the set of TARGETs. Please refer to the string generation rule, STRGEN, and the record generation rule, RGENF for the syntax of string and record generation. We now extend these generation rules' PASS3 requirements to include looping-TARGETs. Each string form, {}, represents one dimension of iteration.

Examples of Looping-TARGETs

{ I }

is a looping-TARGET. It sets I to each element in a given string and causes one iteration for each value of I. The number of iterations is therefore equal to the length of the given string.

{ I ; J }

sets I and J to consecutive elements in a given string. That is, I holds the first element and J holds the second element for the first iteration. For the second iteration, I holds the second value and J holds the third. The final iteration finds I holding the second to the last element and J holding the last element. The number of iterations equals (the length of the given string - 1). If the given string is of length one, then there are zero iterations, i.e., the template {I;J} cannot fit into a string of

length one.

```
{ I ; J ; K }
```

sets I, J, and K to consecutive elements in the given string. The number of iterations is two less than the length of the given string. Again, if the length of the given string is less than three, there are no iterations.

Examples of the FOR quantifier

```
FOR {I} $C {2;4;6;8;10};
```

sets I to the values 2,4,6,8,10. This is equivalent to

```
FOR I $E {2;4;6;8;10};
```

The quantifier

```
FOR {I;J} $C {2;4;6;8;10};
```

sets I and J for each iteration as follows:

```
iteration 1:  I,J=  2,4
iteration 2:  I,J=    4,6
iteration 3:  I,J=      6,8
iteration 4:  I,J=        8,10
```

Similarly,

```
FOR {I;J;K} $C {2;4;6;8;10};
```

sets I, J, and K for each iteration as:

```
iteration 1:  I,J,K=  2,4,6
iteration 2:  I,J,K=    4,6,8
iteration 3:  I,J,K=      6,8,10
```

Looping-TARGETs that Wrap-around

```
{ I ;* J }
```

is equivalent to

```
{ I ; J }
```

except that one more iteration occurs. This final iteration finds I containing the last element in the given string and J containing the first element. That is, the template {I ;* J} has *wrapped around* back to the beginning of the given string.

```
{ I ; J ;* K }
```

sets I, J, and K to the consecutive elements in the given string but in addition, K is allowed to wrap around. Hence, the final iteration finds K holding the first element and I and J holding the second to last and the last elements in the given string.

```
{ I ;* J ; K }
```

differs from the previous example by the placement of the ";*" separator. Here, both J and K are allowed to wrap around. The quantifier

```
FOR {I ;* J} $C {2;4;6;8;10};
```

causes the iterations:

```
iteration 1:  I,J=  2,4
iteration 2:  I,J=    4,6
iteration 3:  I,J=      6,8
iteration 4:  I,J=        8,10
iteration 5:  I,J=          10, 2
```

The quantifier

```
FOR  {I ; J ;* K} $C {2;4;6;8;10};
```

causes the iterations:

```
iteration 1:  I,J,K= 2,4,6
iteration 2:  I,J,K=   4,6,8
iteration 3:  I,J,K=     6,8,10
iteration 4:  I,J,K=       8,10, 2
```

The quantifier

```
FOR  {I ;* J ; K} $C {2;4;6;8;10};
```

causes the iterations:

```
iteration 1:  I,J,K= 2,4,6
iteration 2:  I,J,K=   4,6,8
iteration 3:  I,J,K=     6,8,10
iteration 4:  I,J,K=       8,10, 2
iteration 5:  I,J,K=         10, 2,4
```

In general, the *first* ";*" specifies that the following elements will wrap around. All but the first ";*" are ignored. The number of iterations depends on the length of the given string, and the number of target elements preceding the ";*":

More Complex Looping-TARGETs

```
{ { I } }
```

sets I to each element in a TWO_DIMENSIONAL_ARRAY (refer to the section *When are Types Equal* for the definition of TWO_DIMENSIONAL_ARRAY). That is, working form the outside in, the

{{I}} sets the looping-TARGET {I} to each vector in the given TWO_DIMENSIONAL_ARRAY. The {I} receives each vector by setting I to each element in the vector. Thus, {{I}} represents a two-dimensional loop.

A computer circuit board, or CARD, consists of a bunch of interconnected chips. Each chip has a name and a set of wires or signals to which it connects. In ICL, we can represent this by

```
TYPE  CARD= { CHIP };
      CHIP= [NAME: CHIP_NAME
             SIGNALS: { WIRE_NAME } ];
```

That is, a CARD is a bunch of CHIPs and each CHIP has a name and a set of wires. Now, suppose CARD is a variable of type CARD. Each of the following prints the names of the CHIPs in CARD.

```
DO  WRITE(CHIP.NAME); FOR CHIP $E CARD;
```

or

```
DO  WRITE(N);  FOR  [NAME:N] $E CARD;
```

The second form selects down to the chip-name in the FOR-quantifier whereas the first form selects down to the chip-name in the WRITE statement.

Each of the following forms prints each wire-name, WN, as many times as it is connected to a chip:

```
FOR  CHIP $E CARD;
     DO   FOR  WN $E CHIP.SIGNALS;
               DO  WRITE(WN); END END
```

or

```
FOR  [SIGNALS:S] $E CARD;
     DO   FOR  WN $E S;
               DO  WRITE(WN);  END END
```

or simply

```
FOR  [SIGNALS: {WN} ]  $E CARD;
     DO  WRITE(WN);  END
```

Each of these loops is a two-dimension loop. However, the final form has only one FOR-quantifier. Looking closer at the final form, we see that each element in CARD, a CHIP, is assigned to the looping-TARGET

```
[SIGNALS: {WN} ].
```

This looping-TARGET assigns the SIGNALS component to the looping-TARGET {WN}. {WN} assigns each element in SIGNALS to the variable WN. Therefore, WN is assigned each signal in each chip in CARD.

The following prints each wire-chip pair:

```
FOR [NAME: CN   SIGNALS: {WN} ]  $E CARD;
  DO  WRITE(CN); WRITE(WN);  END
```

Each chip in CARD is assigned to the looping-TARGET

```
[NAME: CN   SIGNALS: {WN} ]
```

This looping-TARGET sets CN to the name of the chip and it sets the looping-TARGET {WN} to the chip's SIGNALS component. The looping-TARGET {WN} then assigns each signal into the variable WN.

The following prints each chip-name to which a given WIRE_NAME is connected:

```
FOR [NAME: CN  SIGNALS: {WN}] $E CARD;
  DO   IF  WN=WIRE_NAME  THEN  WRITE(CN); FI END
```

Let us now consider the problem of sorting a CARD by signals. As CARD now stands, a particular wire-name is scattered among many chips. Our goal is to produce a reshaped CARD so that each WIRE is conveniently listed with all the chips it connects.

```
TYPE  SORTED_CARD= {  [WIRE: WIRE_NAME
                       CHIPS: SET_OF_CHIPS ] };
      SET_OF_CHIPS= { CHIP_NAME };
```

A SORTED_CARD is a set of records each having a unique wire-name along with the set of chip-names to which the wire connects. The following function, complete with declarations, should accomplish the task of translating a CARD into a SORTED_CARD.

```
DEFINE  SORT(CARD:CARD)= SORTED_CARD :
 BEGIN  VAR CN=CHIP_NAME; WN,WN1=WIRE_NAME;
            C= SET_OF_CHIPS;
            SORTED_CARD= SORTED_CARD;
   DO    SORTED_CARD:=NIL;
         FOR [NAME:CN  SIGNALS:{WN}] $E CARD;
            " For each chip-wire pair ... "
          DO     " Have we yet encountered this particular
                   wire-name? "
            IF  NEVER WN1=WN  FOR [WIRE:WN1
                                   CHIPS:C ] $E SORTED_CARD;
            THEN  "We have a new wire-name.
                    Expand SORTED_CARD to include an entry
                    for this new wire-name and its chip"
              SORTED_CARD::=[WIRE:WN
                             CHIPS: {CN}] <$;
            ELSE  "WN=WN1. Add CN to C, the
                    set of chips associated to WN1."
              @(C):= CN <$ COPY(C); FI
          END
   GIVE  SORTED_CARD
 END
ENDDEFN
```

This function may be analyzed as follows: The surface structure is

```
DEFINE  SORT(CARD:CARD)=SORTED_CARD:
   BEGIN    <DECL>
      DO   SORTED_CARD:=NIL;
```

```
        Expand SORTED_CARD to include
          each chip-wire pair
    GIVE  SORTED_CARD
  END
ENDDEFN
```

The main part of this function repeatedly updates SORTED_CARD to account for each individual chip-wire pair. We get each chip-wire pair by using the quantifier

```
FOR [NAME:CN  SIGNALS: {WN} ] $E CARD;
```

Having each chip-wire pair, we use SORTED_CARD as a table and look for an entry having WN as its wire. The <EXPR>

```
NEVER WN1=WN  FOR [WIRE:WN1
                   CHIPS:C ] $E SORTED_CARD;
```

is a BOOLean which yields TRUE if SORTED_CARD does not have an entry for the wire WN. That is, the quantifier

```
FOR [WIRE:WN1  CHIPS:C] $E SORTED_CARD;
```

sets WN1 to each wire in SORTED_CARD and sets C to the wire's accumulated chip set. If it is *never* true that WN1=WN then SORTED_CARD contains no entry for the wire WN. Here the program splits into two cases. First, if SORTED_CARD has no entry for WN, create a new entry on SORTED_CARD for the new wire, WN. This is done by

```
SORTED_CARD::= [WIRE:WN  CHIPS:{CN}] <$ ;
```

We append onto the front of SORTED_CARD a new entry, an entry whose wire is WN and whose associated set of chips is {CN}, the string containing CN as its only element.

On the other hand, if SORTED_CARD already contains an entry for WN, then we merely modify its associated chip set to include CN. The variable C contains WN's associated chip set because the NEVER <EXPR> yielded FALSE. The sentence

```
@(C):= CN <$ COPY(C);
```

appends CN to the front of C. The @ and COPY are used solely for the purpose of making this modification apparent from SORTED_CARD's point of view and not merely from C's point of view. That is, the string referenced by the CHIPS component of an element in SORTED_CARD is treated as an object in its own right which can be modified in a global sense. The @ operator makes the change apparent from *all* points of view. We are obliged, however, to assure ourselves that this global modification affects nothing besides those structures created in this program. We can look at this program and easily prove that the modification is apparent only from C's and SORTED_CARD's points of view. The location referenced by C, the location modified by the @ operator, will always be the CHIPS component of some record in SORTED_CARD. Each entry in SORTED_CARD is created by the record generating <EXPR>

```
[WIRE:WN  CHIPS:{CN}].
```

The CHIPS component, {CN}, is a newly created string and therefore it resides at a location referenced from no other point of view. It is precisely this location in memory which is affected by the @

operator. In short, the location modified by the @ operator is one which is created in this program and which is referenced only by SORTED_CARD and C.

Non-nested Looping-TARGETs

We now define what happens when two looping-TARGETs are disjoint, i.e., neither is nested within the other. For example, the looping-TARGET

```
[A: {X}   B: {Y} ]
```

has two string <EXPR>s which appear independently from one another. This looping-TARGET produces a two-dimensional loop: X and Y are set to each element in the A and B components of the given record in all possible ways. Thus,

```
FOR [A:{X} B:{Y}] $C [A:{1;2;3} B:{10;20}];
```

sets

```
X,Y= 1,10
X,Y= 2,10
X,Y= 3,10

X,Y= 1,20
X,Y= 2,20
X,Y= 3,20
```

or

```
X,Y= 1,10
X,Y= 1,20
```

X,Y= 2,10

X,Y= 2,20

X,Y= 3,10

X,Y= 3,20

One of these sequences occurs, but the user cannot be certain as to which. Refer to the uncertain evaluation order in the record generation rule, RGENF.

The looping-TARGET

{ {I} ; {J} }

defines a three-dimensional loop upon a string of strings. Let us refer to the given string of strings by the name S. This looping-TARGET sets the looping-TARGETs {I} and {J} to the consecutive strings in S. Each of these looping-TARGETs independently sets I and sets J to the elements in the two consecutive strings. In other words, I is set to each element in the first string of S and J is independently set to each element in the second string of S. Then, I is set to each element in the second string of S and J is independently set to each element in the third string of S, etc. In general, the dimensionality of any given looping-TARGET is equal to the number of string-<EXPR>s, {}, occuring within.

Non-primitive Quantifiers

We can combine the primitive quantifiers to form quantifiers of a more general sort. The following section covers the combination of quantifiers with quantifiers and the section there after covers the modification of quantifiers.

Binary Combinations

The following rules construct quantifiers which cause nested looping, lock-stepped looping, and sequenced looping.

QOR: <QUANT> ::= <QUANT$_1$> !! <QUANT$_2$>

QAND: <QUANT> ::= <QUANT$_1$> && <QUANT$_2$>

QTHEN: <QUANT> ::= <QUANT$_1$> THEN <QUANT$_2$>

Meaning

The operator !! nests quantifiers, the operator && lock-steps quantifiers, and the operator THEN sequences quantifiers. That is

<QUANT$_1$> !! <QUANT$_2$>

specifies that for each iteration caused by <QUANT$_1$>, run <QUANT$_2$>. The canonical

DO <SS> <QUANT$_1$> !! <QUANT$_2$>

becomes

DO DO <SS> <QUANT$_2$> <QUANT$_1$>

(-------SS---------)

The resulting number of iterations is the product of the numbers of iterations caused by <QUANT$_1$> and <QUANT$_2$>.

The quantifier

<QUANT$_1$> && <QUANT$_2$>

specifies that <QUANT$_1$> and <QUANT$_2$> step together. This quantifier terminates as soon as either <QUANT$_1$> or <QUANT$_2$> terminates. The canonical

DO <SS> <QUANT$_1$> && <QUANT$_2$>

becomes

```
        prepare for first iteration of <QUANT1>
        prepare for first iteration of <QUANT2>
LOOP:   <SS>
        prepare for next iteration of <QUANT1>
        prepare for next iteration of <QUANT2>
        GOTO LOOP
EXIT:   --
```

where each of the four preparations may spontaneously branch to EXIT. As soon as one quantifier is exhausted, the && combination is said to be exhausted. The resulting number of iterations is the minimum of the numbers of iterations caused by <QUANT$_1$> and <QUANT$_2$>.

The quantifier

<QUANT$_1$> THEN <QUANT$_2$>

specifies that when <QUANT$_1$> terminates, start up <QUANT$_2$>. The canonical

DO <SS> <QUANT$_1$> THEN <QUANT$_2$>

becomes the two sentences

DO <SS> <QUANT$_1$>

DO <SS> <QUANT$_2$>

Examples:

```
DO  WRITE(I);TAB;WRITE(J);CRLF; FOR I FROM 1 TO 3; !!
                                FOR J FROM I TO 3;
```

prints

```
1  1
1  2
1  3
2  2
2  3
3  3
```

The sentence

```
DO  WRITE(I*J);TAB;   FOR I FROM 1 TO 3; !!
                      FOR J FROM 1 TO 3;
```

prints

```
1  2  3  2  4  6  3  6  9
```

The expression

```
{COLLECT I#J  FOR I FROM 1 TO 10; !!
```

```
                    FOR J FROM 1 TO 9;      }
```

forms an array of points having 9 rows and 10 elements per row. The && operator may be used as follows:

```
    DO  WRITE(I);TAB;WRITE(J);CRLF; FOR I FROM 1 TO 10; &&
                                    FOR J FROM 0 TO 20 BY 5;
```

prints

```
    1  0
    2  5
    3  10
    4  15
    5  20
```

The string

```
    {COLLECT I   FOR I $E S; && WHILE I<10; }
```

forms the longest initial substring of S having all elements less than 10. The string

```
    {COLLECT I   FOR I $E S; && REPEAT 5; }
```

forms a string having the first 5 elements of S. If S has less than 5 elements, this new string is a mere copy of S. The string

```
    {COLLECT I    FOR I FROM 0 BY 5; &&
                  REPEAT 20;            }
```

forms a string of 20 elements. The first element is 0 and each following element is 5 greater than its predecessor. Notice that this example uses the && to limit the non-terminating quantifier

```
    FOR I FROM 0 BY 5;.
```

The BOOLean expression

```
ALWAYS A=B   FOR A $E S1; && FOR B $E S2;
```

compares two strings of characters and yields TRUE if one string is the initial segment of the other.

```
DO  WRITE('Element#');WRITE(I);WRITE(' is ');WRITE(J);
            FOR I FROM 1 BY 1; && FOR J $E S1;
```

prints out a table of two columns. The first column is the sequence of integers from 1 to the length of S1 and the second column is the corresponding elements in S1. If S1 is the string {5;10;-3} then we will get

```
Element#1 is 5
Element#2 is 10
Element#3 is -3
```

The following sentence uses the THEN operator:

```
DO  WRITE(I);TAB;  FOR I FROM 1 TO 5 BY 2; THEN
                   FOR I FROM 100 TO 102;  THEN
                   FOR I FROM 200 TO 202;
```

prints

```
1 3 5 100 101 102 200 201 202
```

The summation

```
+ I  FOR I $E S1; THEN FOR I $E S2;
```

yields the sum of the elements in both S1 and S2.

Unary Combinations

Any quantifier may be postfixed with a variety of modifiers. Concerning precedence, these modifiers are tacked on before any binary combinations are considered. For example,

```
FOR I $E S; && FOR J $E S1; WITH J >= 5;
```

groups as

```
FOR I $E S; && (FOR J $E S1; WITH J >= 5;)
```

and not as

```
(FOR I $E S; && FOR J $E S1;) WITH J >= 5;
```

QWITH: <QUANT> ::= <QUANT$_1$> WITH <EXPR$_2$> ;

QINH: <QUANT> ::= <QUANT$_1$> INHIBIT_IF <EXPR$_2$> ;

QRES: <QUANT> ::= <QUANT$_1$> RESET_IF <EXPR$_2$> ;

QECH: <QUANT> ::= <QUANT$_1$> EACH_DO <SS$_2$> ;

QFTM: <QUANT> ::= <QUANT$_1$> FIRST_DO <SS$_2$> ;

QOTH: <QUANT> ::= <QUANT$_1$> OTHER_DO <SS$_2$> ;

QFST: <QUANT> ::= <QUANT$_1$> INITIALLY <SS$_2$> ;

QFIN: <QUANT> ::= <QUANT$_1$> FINALLY_DO <SS$_2$> ;

Type Requirements <EXPR$_2$> = BOOL

PASS3 Requirements <EXPR$_2$> = SOURCE

Meaning

Each modifier has its own meaning:

WITH <EXPR$_2$> ;

filters the <QUANT$_1$> by removing those iterations for which <EXPR$_2$> yields FALSE. That is, the canonical

DO <SS> <QUANT$_1$> WITH <EXPR$_2$> ;

becomes

DO IF <EXPR$_2$> THEN <SS> FI <QUANT$_1$>
(----------SS---------)

The modifier

INHIBIT_IF <EXPR$_2$> ;

inhibits the stepping of <QUANT$_1$> when <EXPR$_2$> yields TRUE except on the first iteration. That is, *before* each non-first stepping of <QUANT$_1$>, evaluate <EXPR$_2$> and abandon the stepping if <EXPR$_2$> yields TRUE.

The modifier

RESET_IF <EXPR$_2$> ;

resets <QUANT$_1$> to start over from the beginning if <EXPR$_2$> yields TRUE. That is, before each non-first stepping of <QUANT$_1$>, evaluate <EXPR$_2$> and if it yields TRUE, reset <QUANT$_1$> so that it now restarts from the beginning.

The modifier

EACH_DO <SS$_2$> ;

specifies that <SS$_2$> be evaluated before each iteration after <QUANT$_1$> has been stepped. That is, the canonical

DO <SS> <QUANT$_1$> EACH_DO <SS$_2$> ;

becomes

DO <SS_2> <SS> <$QUANT_1$>
(----SS----)

The modifier

FIRST_DO <SS_2> ;

specifies that <SS_2> be evaluated before the first iteration but after <$QUANT_1$> is first stepped. That is, the canonical

DO <SS> <$QUANT_1$> FIRST_DO <SS_2> ;

becomes

DO IF *this is the first iteration* THEN <SS_2> FI
<SS> <$QUANT_1$>

The modifier

OTHER_DO <SS_2> ;

specifies that <SS_2> be evaluated before each non-first iteration. That is, the canonical

DO <SS> <$QUANT_1$> OTHER_DO <SS_2> ;

becomes

DO IF *this is not the first iteration*
THEN <SS_2> FI <SS> <$QUANT_1$>

It turns out that <SS_2> appears to be evaluated *between* iterations.

The modifier

INITIALLY <SS_2> ;

specifies that <SS_2> be evaluated before the first stepping of <$QUANT_1$>. The canonical

DO <SS> <$QUANT_1$> INITIALLY <SS_2> ;

becomes

<SS_2> DO <SS> <$QUANT_1$>

The modifier

FINALLY_DO <SS_2> ;

specifies that <SS_2> be evaluated after <$QUANT_1$> terminates. The canonical

DO <SS> <$QUANT_1$> FINALLY_DO <SS_2> ;

becomes

DO <SS> <$QUANT_1$> <SS_2>

(-------SS--------)

Examples:

{COLLECT I FOR I $E S; WITH I>5; }

forms the largest subset of S whose elements satisfy I>5.

```
{COLLECT I FOR I $E S1; WITH
                THERE_IS J=I FOR J $E S2; ; }
                (----------EXPR----------)
            (--------------QUANT-----------)
```

forms the intersection of the strings S1 and S2. We must assume, of course, that the elements of S1 and S2 are comparable with the <BOP> "=". This string <EXPR> collects each element in S1 *only if* that element is in S2.

```
+ J*(J-1)   FOR I $E S;
                    EACH_DO J:=I*SQRT(I); ;
                              (-----SS-----)
```

yields the sum of J*(J-1) where J=I*SQRT(I) for each I in S. This is equivalent to

```
+ (I*SQRT(I))*((I*SQRT(I))-1)  FOR I $E S;
```

The EACH_DO is generally useful for setting auxiliary loop variables which depend on the actual loop variable.
The quantifier

```
FOR I $E S; EACH_DO I::=MAX 5; ;
```

sets I to the maximum of each element in S and 5.
The following sentence plots the path represented by the string of points S:

```
FOR P $E S;
     FIRST_DO   PLOT(P,PEN_UP);;
     OTHER_DO   PLOT(P,PEN_DOWN);;
     DO NOTHING; END
```

The first point is plotted with the pen lifted up and the non-first points are plotted with the pen down. The quantifier itself does all the work. The <SS> being quantified is "NOTHING;", a no-op. The above sentence employs the rule

```
<QUANT>   DO  <SS>  END
```

and so we are obliged to write the "DO NOTHING; END".

The following sentence plots a polygon which is represented by a string of points where the first point is *not* duplicated at the end:

```
FOR  { P1 ;* P2 } $C S;
        FIRST_DO  PLOT(P1,PEN_UP);;
        DO   PLOT(P2,PEN_DOWN);  END
```

The quantifier

```
FOR  { P1 ;* P2 } $C S;
```

sets P1 and P2 to consecutive points in S where the final iteration leaves P2 containing the first point in S. The first iteration plots the first two points of S, P1 and P2, and the other iterations just plot P2.

<EXPR>s and <TYPE>s - Part 2

This section introduces three more datatypes and the corresponding <EXPR> forms which generate and select instances of the new types. Finally, we will introduce a concise notation for specifying strings of points using relative movements.

Another Primitive Type - ID

Just like INT, REAL, POINT, etc. are primitive types, the name ID is another primitive type in ICL. That is, we include the rule

<TYPE> ::= ID

Do not confuse the literal ID with the part-of-speech <ID>.

An instance of the type ID is any <ID>. The type ID is very similar to the type QS and to the SCALAR types. Instances of ID differ from instances of QS by their denotation and their efficiency in the comparison operators. Unlike instances of QS, equal instances of ID are represented by unique memory addresses, like ATOMs in LISP. Thus, comparing two IDs is as efficient as comparing two INTegers. The type ID differs from a SCALAR type in that *any* <ID> may be an instance of ID whereas *only* the <ID>s contained in a SCALAR's <IDLIST> can be instances of the SCALAR type.

ID <EXPR>s - The %

Instances of the type ID are generated by prefixing an <ID> with a percent sign:

EIDID: <EXPR> ::= % <ID>

Type Requirements result = ID

PASS3 Requirements result = SOURCE

Meaning

The resulting value is the <ID> as a literal value.

Examples:

```
%GROUND     is the ID GROUND
%A_B_C      is the ID A_B_C
```

Instances of ID may be compared by the compare operators

```
=  <>  =<  <  >=  >
```

These are the compare operators which have been documented in the section for <BOP>s. Now, we will extend the compare operators to compare two instances of ID:

```
ID  ID    -> BOOL
```

Two IDs are equal if and only if they are the same ID. IDs are ordered in a completely arbitrary way. Thus,

```
%GROUND = %GROUND     is TRUE,
%GROUND = %GND        is FALSE,
%GROUND <> %GND       is TRUE, and
%GROUND < %GND        is uncertain.
```

However, if

```
%GROUND < %GND
```

is TRUE once, then it is true from this time forward. The ordering between two IDs is determined as soon as ICL has seen each <ID> for the first time in any context. It turns out that the value of an

instance of ID is its address in ICL's internal symbol table.

Which Type is Appropriate: ID, QS, or SCALAR(<IDLIST>)?

The types ID, QS, and SCALAR(<IDLIST>) are so similar that one might ask what situations demand the use of one over the other. QS is the most general; any text string is an instance of QS. ID is less general; only those text strings which form valid <ID>s as defined in the section *Basic Conventions* are instances of ID. SCALAR(<IDLIST>) is the least general; only those <ID>s appearing in <IDLIST> are instances of SCALAR(<IDLIST>).

As a rule of thumb, use the least general type with which you can get by. IDs compare faster than QSs and they take up slightly less memory. SCALARs are the best because the compiler checks that any context which expects an instance of a SCALAR does indeed get one of the <ID>s in the SCALAR's <IDLIST>.

Two More Non-primitive Types

The following two type schema each offers a profound extension to ICL. One enables the creation of truly abstract datatypes in the sense that an abstract datatype may have invariant properties *besides* those inherent in a machine representation. The other type schema enables the creation of data which is a program along with some context.

PRIVATE Types

A new datatype, a restriction of an existing datatype, is formed by prefixing the existing datatype with the word PRIVATE:

<TYPE> ::= PRIVATE <TYPE>

The representation for the resulting type is the same as the representation for the original type. However, instances of the original type are not instances of the resulting type and visa versa. The PRIVATE construct is primarily useful for creating *distinct* types whose representations are identical. The user will typically define coercions between the distinct types so to remove the distinction. However, within the coercions, he can monitor the tranference from one type to the other. Here he can place checks and translations which will occur implicitly throughout his programs.

For example, let us consider polygons and convex polygons. A general polygon is suitably represented by a string of points tracing out its vertices:

```
TYPE  POLYGON = { POINT };
```

What is here agreed upon is that any newly formed string of points passes as an instance of POLYGON. Thus,

$$\{\ point_1\ ;\ point_2\ ;\ \ldots\ ;\ point_n\ \}$$

is an instance of POLYGON. Furthermore, any operations which apply to strings of points apply to POLYGONs:

polygon $> point

point <$ *polygon*

polygon $$ *polygon*

polygon [3-]

are all instances of POLYGON. In contrast, a convex polygon is *not* just any old string of points. The above expressions for POLYGONs do not guarantee convexity. In ICL, we can specify that the type CONVEX_POLYGON is a restricted sort of POLYGON by writing

```
TYPE  CONVEX_POLYGON = PRIVATE POLYGON ;
```

Of course, ICL doesn't know *how* CONVEX_POLYGONs are restricted POLYGONs, but the user can capture the restriction in the functions and coercions he writes which consume and produce CONVEX_POLYGONs. CONVEX_POLYGONs are so private that *none* of the above expressions for polygons pass as instances of CONVEX_POLYGON. The *only* way to create or examine an instance of CONVEX_POLYGON is to explicitly specify the transference from privacy to publicity or visa versa. The following section covers the notation for doing so.

Publication and Confirmation - Selection and Generation for PRIVATE Types

The following rules are the only rules which involve PRIVATE types. Instances of PRIVATE types are stripped of their privacy by

PUBLC: <EXPR> ::= PUBLICIZE:::(<EXPR$_1$>)

Type Requirements

<EXPR$_1$> must be a private type, say PRIVATE T. The resulting type is T, the less restricted type.

Meaning

An identity. No additional code is generated. This construct is used to gain access to an instance of a private type.

Example:

If C is a CONVEX_POLYGON, then

```
PUBLICIZE:::(C)
```

is a POLYGON. The coercion

```
LET CONVEX_POLYGON BECOME POLYGON BY
        PUBLICIZE:::(CONVEX_POLYGON) ;
```

specifies that any convex polygon is also a polygon. The privacy of CONVEX_POLYGON may therefore be lifted implicitly.

Instances of a private type are created by:

PRIVY: <EXPR> ::= <ID$_1$> :::(<EXPR$_2$>)

Type Requirements

<ID$_1$> is the name of a declared PRIVATE type and

<EXPR$_2$> = that type which is the generalization of the private type <ID$_1$>. That is, the following relation must hold:

$\langle ID_1 \rangle$ = PRIVATE the-type-of-$\langle EXPR_2 \rangle$

Meaning

An identity. No additional code is generated. This construct is used to *confirm* $\langle EXPR_1 \rangle$ as being a legitimate instance of a PRIVATE type.

Example:

If P is a POLYGON, then

CONVEX_POLYGON:::(P)

is a CONVEX_POLYGON. Similarly,

CONVEX_POLYGON:::($\{point_1; point_2; \ldots; point_n\}$)

is a CONVEX_POLYGON. Note that the *points* can be chosen so as not to form a convex polygon. ICL does not check or know what is meant by CONVEX_POLYGON. ICL only verifies that *except thru this doorway*, the notion of CONVEX_POLYGON is safely preserved. The coercion

```
LET  POLYGON  BECOME  CONVEX_POLYGON  BY
   IF  POLYGON \IS_CONVEX  THEN
                  CONVEX_POLYGON:::(POLYGON)
   ELSE  DO HELP; GIVE NIL    FI ;
```

specifies that any POLYGON passes as a CONVEX_POLYGON but in doing so, the POLYGON is automatically subject to a test. To understand what role this coercion plays, let us consider a function which works only on CONVEX_POLYGONs.

The process of cutting a polygon in two with a line is referred to as polygon clipping. It is a fact that any convex polygon clipped by a line results in another convex polygon. It is also a fact that a general polygon clipped by a line can yield several disconnected polygons. Without filling in the details, the following function clips a convex polygon by a line and yields the convex clipped polygon:

```
DEFINE  CLIP(V:CONVEX_POLYGON  BY:LINE)=CONVEX_POLYGON:
     DO    Clip the polygon V
     GIVE  the clipped polygon
  ENDDEFN
```

Because the argument to CLIP is of type CONVEX_POLYGON, the body of this function can be written assuming the convexity of the argument V. The argument V may be accessed simply as a string of points because the CONVEX_POLYGON -to- POLYGON coercion can render V as a POLYGON.

Where does the POLYGON -to- CONVEX_POLYGON coercion come in? It potentially comes in at two places. First, if CLIP is called with a POLYGON parameter, the coercion will apply before the function call and the parameter's convexity will be checked before entering the function CLIP. Secondly, the result of the clipping is a new string of points, which is called *the clipped polygon* in the program text given above. Before leaving the CLIP function, the coercion will be applied to the clipped string of points, thus verifying its convexity. If CLIP is ever called with a POLYGON which is not convex, the function HELP will be called from within

the POLYGON -to- CONVEX_POLYGON coercion. Similarly, if CLIP yields a non-convex polygon, HELP will be called. Note, however, *we know* that the result of the clipping is always convex. It is therefore a waste of time for the coercion to be invoked upon leaving CLIP. We may simultaneously relieve this final coercion and explicitly state in program text that this procedure always yields a convex polygon by writing

```
CONVEX_POLYGON:::( the clipped polygon )
```

in the GIVE clause. We are explicitly putting our stamp of approval on the result of this function.

Another example implements a restricted type of CHARacter, a capitalized character.

```
TYPE  CAP_CHAR = PRIVATE CHAR ;
```

declares CAP_CHAR to be a restricted CHAR. We can capture the meaning of capitalization by writing the coercions:

```
LET  CAP_CHAR  BECOME  CHAR  BY
                PUBLICIZE:::(CAP_CHAR) ;
LET  CHAR  BECOME  CAP_CHAR  BY
      CAP_CHAR:::( IF CHAR >='a' & CHAR =< 'z'
                   THEN  THE_CHAR(CHAR-'a'+'A')
                   ELSE  CHAR  FI  ) ;
```

The first coercion states that any CAP_CHAR is a valid CHAR. The second coercion states that any CHAR is a CAP_CHAR by capitalizing the CHAR. Before we discuss the ramifications, I must clarify the THEN-clause in the second coercion. The <EXPR>

```
CHAR - 'a' + 'A'
```

specifies arithmetic to be performed on characters. CHARs may not participate in arithmetic but INTegers can. This <EXPR> assumes the existence of a CHAR-to-INT coercion, one which maps a CHAR into its INTeger ASCII code. Assuming such a coercion,

```
CHAR - 'a' + 'A'
```

results in type INTeger, the ASCII code for a capital letter. The "identity" function THE_CHAR maps an INTeger into a CHAR. Thus,

```
THE_CHAR( CHAR - 'a' + 'A' )
```

is the desired capitalized character. Just as the INTeger-to-REAL coercion is generally assumed, the user may assume the existence of the CHAR-to-INTeger coercion and the THE_CHAR INTeger-to-CHAR function. This coercion and function are contained in the file BEGIN.ICL, the first file read into a freshly created ICL system.

WARNING:

A common user error accompanies coercions which coerce to a private type, e.g., the CHAR-to-CAP_CHAR coercion. The user might forget to write the confirmation, e.g., CAP_CHAR:::(...) around the body of the BY-clause in the coercion, e.g.,

```
LET CHAR BECOME CAP_CHAR BY  IF..THEN..ELSE..FI ;
```

This forgetfulness results in an infinite loop via recursion. ICL *will apply* the coercion to the body of the coercion itself in order to satisfy the requirement that the body of the coercion result in the type CAP_CHAR. Even though the IF-THEN-ELSE results in a CHAR which is capitalized, ICL

doesn't know that this is a CAP_CHAR. The user must explicitly confirm that the body is of type CAP_CHAR.

What do the CHAR and CAP_CHAR coercions buy us? First of all, the types CHAR and CAP_CHAR are now equivalent. One or the other may be used anywhere with no distinction. However, anywhere the user uses the type CAP_CHAR, he will be guaranteed to have a capital character. Variables declared as CAP_CHARs will always contain capital characters. No coercion will occur when passing an <EXPR> of type CAP_CHAR to a function requiring a CAP_CHAR.

Upon changing the declarations of some variables from CAP_CHAR to CHAR or visa versa, the placement of coercions will automatically vary in a given program. ICL always minimizes the number of applied coercions in a static sense. In this sense, ICL optimizes a program. However, the few places where ICL does place coercions might be inside a loop. In the dynamic sense, the program is not necessarily optimized. However, as in FORTRAN, the user can optimize his program by judiciously choosing which variables are to be of one type and which are to be of the other type.

The following exemplifies how ICL minimizes the application of coercions. Appending two points to an existing CONVEX_POLYGON might be expressed as

convex polygon $\$>$ $point_1$ $\$>$ $point_2$

The coercions will be placed as follows. For abbreviation, P will stand for the type POLYGON and CP will stand for the type CONVEX_POLYGON.

convex polygon \$> $point_1$ \$> $point_2$

(-----CP-----)

(-----P------)

(-------------P---------)

(--------------------P-------------)

(--------------------CP------------)

First, the CONVEX_POLYGON is coerced to a POLYGON. Then the two points are appended to the POLYGON. Finally, if the result must be viewed as a CONVEX_POLYGON, the finished POLYGON coerces to back to CONVEX_POLYGON and only this once, the POLYGON is tested for convexity. This interpretation requires the minimum number of coercions, two.

Processes - The //...\\ and the <*...*>

This section documents ICL's process datatypes and their instances. Procedures, like data structures, may be created, invoked, and passed around both in variables and within data structures. Coercions and functions can be defined which transform processes or data to yield other processes or data.

The term *Process Generation* refers to the creation of a process and the term *Invocation* refers to the transfering of control to a process. The symbols // and \\ are used to delimit the program text making up a process; they denote process generation. The symbols <* and *> are used to specify invocation of a process.

Examples

```
A := //  <SS>  \\ ;
```

sets A to represent the program action specified by <SS>. Writing

```
<* A *) ;
```

will cause <SS> to execute. A may be invoked as many times and in as many environments as desired.

```
A := //  I::=+1;  \\ ;
```

sets A so that <*A*> increments the global variable I.

Parameters may be passed:

```
F := //(X:REAL)  X*X  \\ ;
```

sets F to represent the function X*X.

```
<*F*>(5)
```

yields the value 25.

```
G:= //(X:REAL)  SIN( <*F*>(X) )  \\ ;
```

sets G to represent the function SIN of whatever <*F*>(X) yields.

```
<*G*>(0)     is 0    and
<*G*>(2)     is SIN(4).
```

However, if we now write

```
F:= //(X:REAL)  1-X  \\ ;
```

then

```
<*G*>(0)   is SIN(1)    and
<*G*>(2)   is SIN(-1).
```

Variables appear to represent the values they hold at the time of invocation and not at the time of process generation. Thus, a change in F is reflected in G because G makes reference to F.

The user may specify that values taken at the time of process generation be available at the time of invocation. Such values are called context. For example,

```
G:= //(X:REAL)[F;]    SIN( <*F*>(X) )    \\ ;
```

sets G to represent the function SIN of <*F*>(X) *where F represents* the value of F now, at the time of process generation. G is now immune to any change made to the variable F. When G is invoked, the value F in G will appear to be what it was at the time of the assignment and not what it will be at the time of invocation. The

context variable F is said to be frozen.

The user specifies the desired set of variables whose values are to appear frozen at the time of process generation by enclosing them in square brackets and inserting a semicolon after each variable. G now represents the function

```
SIN(1-X)
```

because F = (1-X) at the time G was assigned. Writing

```
F := //(X:REAL)  COS(X)  \\ ;
```

does not affect G at all. In fact, writing

```
F:= //(X:REAL)[G;]  <*G*>(X) / 2   \\ ;
```

sets F to represent one half the value of <*G*>(X) where G appears frozen now, i.e.,

```
F = SIN(1-X)/2.
```

Note that the F in the definition for G is still 1-X despite this new assignment because F was enclosed in square brackets in the assignment for G.

The sentences

```
F:= //(X:REAL)  X  \\;
DO F:= //(X:REAL)[F;]  <*F*>(X) * <*F*>(X) \\ ;   REPEAT 5;
```

set F to represent the function X raised to the 32nd power.

Process Types

The examples presented above were done so assuming that the variables A, F, and G were previously declared. They were to be declared to be variables of the types

```
// \\                  for A     and
//REAL(REAL)\\         for F and G.
```

A is a process which neither produces a value nor takes any parameters. F and G are each of the process type which produces a REAL and which expects exactly one parameter, whose type is REAL.

Formally, we have the following new rules for <TYPE>:

```
<TYPE>  ::=  // \\
<TYPE>  ::=  //  <TYPE1>  \\
<TYPE>  ::=  //  (  <IDLIST>  )  \\
<TYPE>  ::=  //  <TYPE1>  (  <IDLIST>  )  \\
```

The first <TYPE> denotes a process which returns no value. The second <TYPE> denotes a process which returns a value of type $\langle TYPE_1 \rangle$. The third <TYPE> denotes a process which returns no value but which does expect input parameters whose types are named by the <ID>s in <IDLIST>. The fourth <TYPE> is similar to the third <TYPE> except that not only does it expect input parameters, it also returns a value of type $\langle TYPE_1 \rangle$.

All the <ID>s in the <IDLIST>s must be the names of declared types. The reader might note that these four <TYPE> rules correspond to the four kinds of function headers presented in the section *Declarations*.

Examples:

TYPE SS = //\\;

declares that SS is the name of a process type. Instances of SS neither return a value nor do they accept input parameters.

TYPE FUNCTION = //REAL(REAL)\\ ;

declares that FUNCTION is the name of a process type. Each instance of FUNCTION accepts one parameter of type REAL and returns a value of type REAL.

TYPE PLOTTER = //(POINT,PLOTTER_COMMAND)\\ ;

declares PLOTTER to be a process type which expects two parameters, a POINT and a PLOTTER_COMMAND. The invocation of a PLOTTER returns no value.

TYPE CHAR_PRODUCER = // CHAR \\ ;

declares CHAR_PRODUCER to be a process type which yields a CHAR upon each invocation and which expects no input parameters.

Process <EXPR>s - Generating Forms for Process Types - The //...

The following rules define the syntax for making instances of process types:

SUSB1: <SUSB> ::= //

SUSB2: <SUSB> ::= <$SUSB_1$> (<$CTYPE_2$>)

SUSB3: <SUSB> ::= <$SUSB_1$> [<ASN_2>]

SUSB4: <SUSB> ::= <$SUSB_1$> { <ASN_2> }

SUSF1: <EXPR> :::= <$SUSB_1$> <$EXPR_2$> \\

SUSF1S: <EXPR> :::= <$SUSB_1$> <SS_2> \\

Informally, an instance of a process type is generated by enclosing an <EXPR> or an <SS> between a <SUSB> and a \\. A <SUSB> is a // optionally followed by parameter specification or by context specification or by both. Parameters are specified via the rule SUSB2 and context is specified via either of the rules SUSB3 and SUSB4.

Type Requirements

There must exist a declared process datatype whose parameter types sequentially match the parameter types specified in $\langle CTYPE_2 \rangle$ and whose return type is the type of $\langle EXPR_2 \rangle$ if the rule SUSF1 is used. If the rule SUSF1S is used, the process type must include no return type. The resulting type for the rules SUSF1 and SUSF1S is any such declared process type.

For example

```
//  <SS>  \\          is of type //\\.
// an INT \\          is of type //INT\\.
//(X,Y:REAL  B:BOOL)   <SS>  \\
                      is of type //(REAL,REAL,BOOL)\\.
//(X,Y:REAL  B:BOOL)   a POINT  \\
          is of type //POINT(REAL,REAL,BOOL)\\.
```

The $\langle ASN_2 \rangle$ in the rules SUSB3 and SUSB4 plays no part whatsoever in the type requirements. For example,

```
//[A;B;] <SS>  \\   is of type //\\.
//[A;B;] an INT \\  is of type //INT\\.
//(X:REAL)[A;B;] <SS>  \\
```

is of type //(REAL)\\.

//(X:REAL)[A;B;] *a POINT* \\

is of type //POINT(REAL)\\.

PASS3 Requirements <EXPR$_2$> = SOURCE = result

Meaning

The resulting value is a process which either produces the value <EXPR$_2$> or performs the action <SS$_2$> where the <EXPR$_2$> or <SS$_2$> is evaluated not now, but at the time this value is invoked. Invocation will be formally described with the next set of rules. This resulting value will expect parameters at the time of invocation if the rule SUSB2 has been used.

Further Requirements

Each variable named in <EXPR$_2$> or <SS$_2$> must either be

1) a global variable, or
2) a parameter variable specified in <CTYPE$_2$>, or
3) a context variable specified in <ASN$_2$>, or
4) a variable declared local within <EXPR$_2$> or <SS$_2$> itself.

A variable in <EXPR$_2$> or <SS$_2$> may *not* be a local variable declared outside of <EXPR$_2$> or <SS$_2$> except via (3). A violation will be reported by the cryptic error message:

?SLOAD or ?TSTORE: Address has illegal index field.

The *specified variables* of <ASN$_2$> must be variables declared outside <EXPR$_2$> or <SS$_2$> and they may be local or global.

Further Meaning

Each of the *specified variables* of <ASN$_2$> is automatically made local to the body, <EXPR$_2$> or <SS$_2$>. That is, <EXPR$_2$> or <SS$_2$> may read and write any of the *specified variables* of <ASN$_2$> and the effect will be apparent only to <EXPR$_2$> or <SS$_2$>.

The *implied assignments* of <ASN$_2$> are carried out now and not at the time of invocation. The *implied assignments* can be viewed simply as the initialization of the context variables for the process.

The distinction between the rules SUSB3 and SUSB4, the square brackets vs. the curly brackets, is as follows: The *specified variables* of the <ASN$_2$> enclosed in square brackets have the property that their values are reset to their initialized values upon each invocation of the process. The *specified variables* of the <ASN$_2$> enclosed in curly brackets are *not* reset upon each invocation and hence they may be used to remember information from the previous invocation.

WARNING:

Processes constructed with the rule SUSB4, the curly brackets, have a property unlike any other data in ICL. Such a process appears to evolve independently from all points of view. Thus, with

```
B:= //{I:=0;}   WRITE( (I::=+1;) );  \\ ;
A:= B;
```

we have the following scenario:

```
<*A*>;        prints a 1
<*A*>;        prints a 2
<*B*>;        prints a 3 and not a 1.
```

However, if we now write

```
A:= COPY(B);
<*B*>;        prints a 4
<*B*>;        prints a 5
<*A*>;        prints a 4 and not a 6
<*B*>;        prints a 6.
```

COPYing a process yields a process whose further evolution is independent from the evolution of the original process except for the following convention: Any (sub)processes referenced by the original process are now shared between the original process and the copied process. Thus, an invocation of a subprocess from either the original or the copied process will be apparent from both the original and the copied processes.

The reader who has examined the section about ICL's policy towards assignments, pointers, and copying may note that a process generated with the rule SUSB4 evolves as though it were modified via the @-operator upon each invocation. A process is represented much like a record is represented; there is a list of memory elements, one for each context variable, and a field containing the address of a program. The memory elements of this context list are updated *in place* upon completion of each invocation so that they hold the new current values for the process's context variables. An explicit COPY is required for the creation of an independent instance precisely because of this @-like, in place, treatment for a process's context list. The COPY operation makes a copy of the context list copying only the top level structure: Structures

referenced from the list are not copied, rather, they are shared by both the original and the copied list.

Examples:

The declarations

```
TYPE  SS = //\\ ;
      PROCESS_QUEUE= { SS } ;
VAR   RUNABLE_PROCESSES= PROCESS_QUEUE;

DEFINE RUN_ONE_PROCESS:
     IF DEFINED(RUNABLE_PROCESSES) THEN
          <* RUNABLE_PROCESSES[1] *>;
          RUNABLE_PROCESSES:=RUNABLE_PROCESSES[2-]; FI
ENDDEFN

DEFINE RUNABLE(S:SS):
     RUNABLE_PROCESSES::=$> S ;
ENDDEFN
```

define a dumb scheduler which has a global variable of type PROCESS_QUEUE, a string of processes. The function RUNABLE puts a process on the queue and the function RUN_ONE_PROCESS executes the first process on the queue and removes that process from the queue.

Non-linear transformation upon pictures are supported by the declaration

```
TYPE  POINT_XFRM= //POINT(POINT)\\ ;
```

A POINT_XFRM is a function which takes a POINT and which yields the transformed POINT.

```
VAR  ITALICIZE= POINT_XFRM;
ITALICIZE:= //(P:POINT)  P.X+P.Y # P.Y \\;
```

sets the POINT_XFRM ITALICIZE to be a mapping which tilts a picture 45 degrees to the right. The function

```
DEFINE COMPOSED_WITH(A,B:POINT_XFRM)= POINT_XFRM:
    //(P:POINT)[A;B;]  <*A*>( <*B*>(P) ) \\
ENDDEFN
```

will form a POINT_XFRM which is the composition of two given POINT_XFRMs. The resulting POINT_XFRM takes its input POINT, passes it thru B and then passes the result thru A. The variables A and B are enclosed in square brackets so that their values will be available at the time of invocation. If A and B were not specified as context variables, an error message would be issued at compile time because the variables A and B inside the resulting POINT_XFRM are not global variables. The POINT_XFRM

```
ITALICIZE \COMPOSED_WITH ITALICIZE
```

yields a POINT_XFRM which applies ITALICIZE twice.

The declaration

```
TYPE  FUNCTION= //REAL(REAL)\\ ;
```

defines FUNCTION to be the process type which maps a REAL to a REAL. The function

```
DEFINE  PRIME(F:FUNCTION)= FUNCTION:
```

```
//(R:REAL)[F;]  (<*F*>(R+EPSILON)-<*F*>(R)) /
                                EPSILON      \\
    ENDDEFN
```

maps a FUNCTION into its derivative, assuming EPSILON is a global variable. When the derivative is invoked, the function F will be invoked twice, once at R+EPSILON and once at R.

The declaration

```
TYPE  FUNCTION_PRODUCER= //FUNCTION(REAL)\\ ;
```

defines FUNCTION_PRODUCER to be a process type which maps a REAL to a FUNCTION. The following is an instance of FUNCTION_PRODUCER:

```
//(N:REAL)
        //(T:REAL)[N;]   SIN(N*2*3.141592 * T) \\ \\
```

This instance takes in a REAL, N, and yields the SIN function which maps the interval between 0 and 1 into N cycles. Thus, with

```
VAR  FMAP= FUNCTION_PRODUCER;
     F = FUNCTION;

FMAP:= //(N:REAL)
          //(T:REAL)[N;] SIN(N*2*3.141592 * T) \\ \\ ;
```

the statement

```
F:= <*FMAP*>(3) ;
```

sets F to a function which maps the interval between 0 and 1 into 3 cycles of the SIN function.

The resulting function, F, is not as optimized as it might be. Upon repeated invocation of F, the value

```
N*2*3.141592
```

will repeatedly be calculated even though it does not depend on the parameter to F. We can remove this calculation from F and put it into FMAP by assigning FMAP as follows:

```
FMAP:=//(N:REAL)
        //(T:REAL)[N:=N*2*3.141592;]   SIN(N*T)  \\ \\ ;
```

Now, when we write

```
F:= <*FMAP*>(3);
```

the resulting function, F, involves one multiply and one SIN calculation. The calculation

```
N:= N*2*3.141592 ;
```

is performed at the time FMAP is invoked and not at the time F is invoked. That is, the variable N is initialized to the convenient value during the process generation for the return value from FMAP.

The following instances of FUNCTION yield identical results:

```
1)  //(R:REAL)[A;B;C;]  SIN(A*COS(B)+C*R) \\
2)  BEGIN VAR D=REAL;
        DO  D:= A*COS(B);
        GIVE //(R:REAL)[C;D;]  SIN(D+C*R) \\
    END
3)  BEGIN VAR D=REAL;
        //(R:REAL)[C;D:=A*COS(B);] SIN(D+C*R) \\
```

```
    END
4)  BEGIN VAR D=REAL;
        //(R:REAL)[D:=A*COS(B);C;] SIN(D+C*R) \\
    END
```

The BEGIN-END is used to create an auxiliary variable, D, which will contain the intermediate value A*COS(B).

The difference between the square brackets and the curly brackets is exemplified in the following:

```
TYPE  INT_PRODUCER= //INT\\ ;
VAR   A,B= INT_PRODUCER;  I=INT;

A:= //[I:=5;] DO I:=I+1; GIVE I \\ ;
B:= //{I:=5;} DO I:=I+1; GIVE I \\ ;
```

The expression <*A*> will always yield 6 whereas the expression <*B*> will yield the number 6 upon first invocation, the number 7 upon second invocation, etc. Each invocation of B yields a number one greater than the result of the previous invocation.

The user can define coercions between process types:

```
TYPE  BUNCH_OF_SS= { SS } ;
LET BUNCH_OF_SS  BECOME  SS  BY
    //[BUNCH_OF_SS;]  BEGIN  VAR S=SS;
                DO  <*S*>;  FOR S $E BUNCH_OF_SS;  END \\ ;
```

This says that any string of SSs may be viewed as a single SS which sequentially evaluates each SS in the string. Thus,

```
{ //I:=21;\\ ; //J:=I+1;\\ ; //WRITE(J);\\ }
```

may be seen as a single SS whose invocation prints the number 22.

Another example of coercion between process types involves another interpretation for the type FUNCTION_PRODUCER. A FUNCTION_PRODUCER can be seen as a single function which takes *two* REAL parameters and which yields a REAL, like the following type:

```
TYPE  TWO_DIM= //REAL(REAL,REAL)\\ ;
```

That is, we can declare

```
LET  FUNCTION_PRODUCER  BECOME  TWO_DIM  BY
    //(R,T:REAL)[FUNCTION_PRODUCER;]   "takes two parameters"
        <* <*FUNCTION_PRODUCER*>(R) *>(T)  \\ ;
```

A FUNCTION_PRODUCER is viewed as a TWO_DIM by using the first parameter of TWO_DIM to select a function from FUNCTION_PRODUCER and evaluating that function at the second parameter. Similarly, we can go backwards with

```
LET  TWO_DIM  BECOME  FUNCTION_PRODUCER  BY
   //(R:REAL)[TWO_DIM;]      "takes one parameter"
      //(T:REAL)[TWO_DIM;R;]  <*TWO_DIM*>(R,T) \\ \\ ;
```

The resulting FUNCTION_PRODUCER, given R, yields the function TWO_DIM where TWO_DIM's first parameter is frozen at R.

Our final example involves the definition of a process type called PICTURE. We will adopt the point of view that a PICTURE is so general that all we know is that a PICTURE may be invoked and that this invocation may invoke a global variable called PLOTTER, sending to PLOTTER a POINT and a pen-up or pen-down directive. Furthermore, we shall assume that PLOTTER will automatically

transform its given point by displacing that point by another global variable, ORIENTATION.

```
TYPE  PLOTTER= //(POINT,PLOTTER_COMMAND)\\ ;
                  "Expects ORIENTATION to be preset"

      PLOTTER_COMMAND= SCALAR(PEN_UP,PEN_DOWN);
      PICTURE= //\\ ;      " PICTURE expects PLOTTER
                  and ORIENTATION to be preset "

VAR   PLOTTER= PLOTTER;   ORIENTATION= POINT;
```

The following is an instance of PLOTTER:

```
PLOTTER:= //(P:POINT E:PLOTTER_COMMAND)
              P::= + ORIENTATION;
              CASE E OF
                   PEN_UP: WRITE('Up'); WRITE(P);
                   PEN_DOWN: WRITE('Down');WRITE(P);
              ENDCASE
              CRLF;     \\    ;
```

This plotter is especially well suited for terminals which have no plotting capabilities. This plotter prints the points on the terminal. Note that this plotter displaces its given point by ORIENTATION like any instance of PLOTTER should.

We can form many instances of PICTURE by introducing a type for a special sort of picture and by defining a coercion from this type to the type PICTURE:

```
TYPE  SIMPLE_PIC= { CURVE } ;
      CURVE = { POINT } ;
```

A SIMPLE_PIC is a string of CURVEs where each CURVE is a set of points meant to be drawn with the pen down. That is, the pen is to be lifted only for the first point in each CURVE. The following coercion lets a SIMPLE_PIC be viewed as a PICTURE:

```
LET SIMPLE_PIC  BECOME  PICTURE  BY
    //[SIMPLE_PIC;]    BEGIN  VAR P=POINT; C=CURVE;
        FOR C $E SIMPLE_PIC; DO
          FOR P $E C;    FIRST_DO  <*PLOTTER*>(P,PEN_UP);;
                         OTHER_DO  <*PLOTTER*>(P,PEN_DOWN);;
                  DO NOTHING; END END
      END \\ ;
```

In fact, any datatype which can be plotted can be coerced to a PICTURE. One needs merely to place the plotting procedure, enclosed in //...\\, as the body of the coercion.

The following function makes use of the global variable ORIENTATION.

```
DEFINE  DISPLACED_BY(V:PICTURE DISP:POINT)=PICTURE:
    //[V;DISP;]
              HOLDING ORIENTATION::=+DISP;
                 DO  <*V*>;   ENDHOLD \\
    ENDDEFN
```

That is, the resulting PICTURE is the process which invokes the given PICTURE, V, in the environment where the ORIENTATION has been moved by the given POINT, DISP. Thus,

picture \DISPLACED_BY 10#12

yields the PICTURE *picture* all of whose points will be displaced by 10#12. Note that DISPLACED_BY's modification to the variable ORIENTATION is done with the HOLDING form. Thus, when V finishes, ORIENTATION will be reset to its old value, as is appropriate.

The following function takes two PICTUREs and produces a PICTURE which repeatedly draws the first picture displaced by each point in the second picture:

```
DEFINE  OUTER_PRODUCT(A,B:PICTURE)= PICTURE:
   //[A;B;]    BEGIN  VAR V=PLOTTER;
    HOLDING PLOTTER:=//[V:=PLOTTER;B;](P:POINT
                                       E:PLOTTER_COMMAND)
                                  HOLDING PLOTTER:=V;
                                      ORIENTATION::=+P;
                                   DO <*B*>; ENDHOLD  \\ ;
        DO  <*A*>;  ENDHOLD    END \\
ENDDEFN
```

The first picture, A, is invoked in the context where PLOTTER has been set to a procedure which draws B. Thus, each point which A sends to PLOTTER will be received by the procedure

```
//[V:=PLOTTER;B;](P:POINT  E:PLOTTER_COMMAND)
            HOLDING  PLOTTER:=V;
                    ORIENTATION::=+P;
                DO  <*B*>;  ENDHOLD  \\
```

This procedure ignores E, the PLOTTER_COMMAND and displaces ORIENTATION by P, the point sent from A. This procedure also sets the global variable PLOTTER back to its original value so that upon invocation of B, B will send its points to the original plotter and not back to this procedure. This procedure finally invokes B. Upon completion of B, ORIENTATION's and PLOTTER's old values are restored and so A's next point will be sent to this procedure and B will be replotted, this time displaced by the new point issued from A.

The statement

```
HOLDING PLOTTER:=//[V:=PLOTTER;B;] ... \\ ; DO...
```

sets PLOTTER to a procedure which has access to the old value of PLOTTER and to B. The reader might wonder why we've gone to the trouble of assigning V the value of PLOTTER instead of merely writing

```
HOLDING PLOTTER:=//[PLOTTER;B;] ... \\ ;  DO ...
```

Indeed, this second form does give the procedure body access to the old value of PLOTTER and to B. However, the procedure body loses access to the global variable PLOTTER. Recall that each of the *specified variables* of the <ASN> between the square brackets is automatically made local to the procedure body. Thus, within the procedure body, the name PLOTTER refers to a local variable and not to the global variable named PLOTTER. The first form, which uses the variable V, does not lose access to the global variable named PLOTTER.

Selection Forms for Process <EXPR>s - Invocation - The <...*>*

The formal rules for process invocation are:

SEMNOP: <FID> ::= <* $<EXPR_1>$ *>

FID1: <EXPR> ::= <FID>

FID2: <EXPR> ::= $<FID_1>$ $<ARGS_2>$

FID3: <EXPR> ::= <FID> ;

FID4: <EXPR> ::= $<FID_1>$ $<ARGS_2>$;

These rules are identical to the rules for procedure and function calling in ICL except that the <FID> replaces the procedure or function name.

Type Requirements

$<EXPR_1>$ must be a process type.

The rule FID1 requires that $<EXPR_1>$ expects no parameters and that it returns a value. The type of the returned value is the resulting type for the rule FID1.

The rule FID2 requires that $<EXPR_1>$ expects parameters whose types sequentially match the types of the <EXPR>s in $<ARGS_2>$. In addition, $<EXPR_1>$ must return a value. The type of the return value is the resulting type for the rule FID2.

The rules FID3 and FID4 are similar to the rules FID1 and FID2 except that it is required that $<EXPR_1>$ returns no value.

PASS3 Requirements

<EXPR$_1$> = SOURCE = result and

each <EXPR> in <ARGS$_2$> = SOURCE

Meaning

Evaluate <EXPR$_1$>, thus yielding an instance of a process type. Then evaluate each <EXPR> in <ARGS$_2$>. Finally call the process yielded by <EXPR$_1$>, passing the <EXPR>s in <ARGS$_2$> as parameters. The resulting value for the rules FID1 and FID2 is the value returned by the invocation.

The debugging package will be entered if the value of <EXPR$_1$> is NIL. Unfortunately, ICL finds this error to be fatal: When the user leaves the debugging package, ICL will gracefully crash.

Examples:

```
<*  // WRITE('Hi'); \\  *> ;
```

prints Hi.

```
<*  //(R:REAL)  WRITE(R); \\  *>(1.7) ;
```

prints 1.7.

```
<*  //(R:REAL)  R*R\\  *>(5.0)
```

yields the REAL 25.0.

More examples are found in the previous section.

Just as the name of a function may be prefixed with a backslash to produce a <BOP> or <RHUOP>, an <FID> may similarly be prefixed:

SEMNOP: <BOP> ::= \ <FID>

SEMNOP: <RHUOP> ::= \ <FID>

The precedence of the resulting <BOP> is the same as the precedence for the rule BOPBID in the section for <BOP>s. The type and PASS3 requirements and the meaning are derived by transforming

<EXPR$_1$> \<FID> <EXPR$_2$> to <FID>(<EXPR$_1$>,<EXPR$_2$>)

and

<EXPR$_1$> \<FID> to <FID>(<EXPR$_1$>)

Examples:

```
5 \<* //(A,B:INT) A+B*A\\ *> 6
```

yields 5+6*5, or 35.

```
K:= //(A,B:REAL) A+B \\;
WRITE( \<*K*> R FOR R FROM 1 TO 10; );
```

prints the sum of 1 thru 10.

Process Generation - The Short Form

No new semantics are presented in this section. Rather, a short form for specifying process generation is presented. The short form is applicable when the user wishes to form a process whose body already exists as an ICL function. For example, the long form

```
//(R:REAL) SIN(R) \\
```

forms a process which merely calls the existing function SIN. The corresponding short form is

```
//: SIN(REAL) \\
```

The short form includes a colon immediately after the //. The body of the short form consists of an ICL function name along with its parameter types.

Besides saving a few characters of typing, the short form saves a little of both execution and memory expense. In the above example with SIN, the invocation of the long form involves two function calls, one for the // and one for SIN. In contrast, the short form involves only one function call, a call to SIN. The compiler allocates space for a process's machine code in chunks of 32 words. This allocation occurs only once, at compile time. The short form allocates no machine-code space whereas the long form has to allocate at least 32 words, even though only a few words are actually used for calling the SIN function.

```
SEMNOP:   <QSUSB>  ::=  //: <ID>

 - - -:   <QSUSE>  ::=  \\
 - - -:   <QSUSE>  ::=  ;\\

QSUS2:    <EXPR> ::=  <QSUSB1>  <QSUSE>
QSUS3:    <EXPR> ::=  <QSUSB1>  (  <IDLIST2>  )  <QSUSE>

QSUS4:    <EXPR> ::=  <QSUSB1>  <SARGS2>  <QSUSE>
QSUS5:    <EXPR> ::=  <QSUSB1>  (  <IDLIST2>  )  <SARGS3>  <QSUSE>

                where

ARGS3:    <SARGS>  ::=  <SARGX>  ]
ARGS2:    <SARGX>  ::=  <SARGX1>  ,  <EXPR2>
ARGS1:    <SARGX>  ::=  [  <EXPR1>
```

All short forms begin with a <QSUSB>, the

//: <ID>

The <ID> is the name of an existing ICL function. All short forms end with either \\ or ;\\. The semicolon is entirely optional. Thus, a short form looks like

//: <ID> ... \\ or

//: <ID> ... ;\\

The ... may be blank or may be any one of the following:

(<IDLIST>) or

[<EXPR> , <EXPR> , ...] or

(<IDLIST>) [<EXPR> , <EXPR> , ...]

The part-of-speech <SARGS> represents the form

[<EXPR> , <EXPR> , ... , <EXPR>]

This form is precisely the form represented by the part-of-speech <ARGS> except that the enclosing parentheses are replaced by square brackets.

The type and PASS3 requirements and the meaning for the resulting <EXPR>s will be given for each of the <EXPR>-producing rules, QSUS2 thru QSUS5.

QSUS2 looks like //: <ID> \\

This short form is equivalent to the long form

// <ID> \\ or // <ID>; \\

For example, if the function NUMBER_OF_JOBS takes no parameters and yields an INTeger, then

//: NUMBER_OF_JOBS \\

is a process which calls NUMBER_OF_JOBS and yields the result yielded by NUMBER_OF_JOBS.

QSUS3 looks like //: <ID_0> (<ID_1> , <ID_2>) \\

This short form is equivalent to the long form

//(X: <ID_1> Y: <ID_2>) <ID_0>(X,Y) \\

For example,

//:SIN(REAL)\\ is equivalent to

//(X:REAL) SIN(X) \\ and

//:DISTANCE(POINT,POINT)\\ is equivalent to

//(X,Y:POINT) DISTANCE(X,Y) \\.

QSUS4 looks like //: <ID_0> [<$EXPR_1$> , <$EXPR_2$>] \\

This short form is equivalent to the long form

//[X:=<$EXPR_1$>; Y:=<$EXPR_2$>;] <ID_0>(X,Y) \\

That is, the <EXPR>s between the square brackets are taken as parameters to the function <ID_0> whose values are frozen now, at the time of process generation. For example,

//:SIN[I+J]\\ is equivalent to the long form

//[K:=I+J;] SIN(K) \\

Invocation of this process will always yield the same number because SIN depends on no global variables. This process's invocation will call SIN passing the value I+J. The value I+J is a single number which is computed at the time of process generation and not at the time of invocation.

QSUS5 looks like //: <ID_0> (<ID_1> , <ID_2>) [<$EXPR_1$>] \\

This short form is equivalent to

```
//(X: <ID1>  Y: <ID2> )  [Z:= <EXPR1> ;]
                    <ID0>(X,Y,Z)  \\
```

That is, the <ID>s between the parentheses are the names of the parameter types for the function <ID_0>'s first two parameters. The <$EXPR_1$> is the function's third parameter. The third parameter is evaluated now, at the time of process generation, whereas the first two parameters are taken at the time of invocation. For example,

```
//:DISTANCE(POINT)[3#4]\\     is equivalent to
//(P:POINT)  DISTANCE(P,3#4) \\
```

This process expects one parameter of type POINT. It calls the function DISTANCE, passing 3#4 as the second parameter. Similarly,

```
//:DISTANCE(POINT)[P1+P2]\\    is equivalent to
//(P:POINT)[V:=P1+P2;]   DISTANCE(P,V) \\
```

The P1+P2 is evaluated now, at the time of process generation, and not at the time of invocation.

A Concise Notation for Specifying Relative Points - The "."

A string of points may be specified as follows:

{ 1#2 ; .#5 ; .#7 ; 20#. ; .+1#5 ; .+7#.-8 }

The "." refers to the previously specified point's x or y coordinate. For example, the string mentioned above is equivalent to

{ 1#2 ; 1#5 ; 1#7 ; 20#7 ; 21#5 ; 28#-3 }

If the period lies to the left of the #, it refers to the previous point's x-coordinate. If the period lies to the right of the #, it refers to the previous point's y-coordinate.

This concise notation for specifying relative points is implemented by a combination of the built in rule

CURENT: <EXPR> ::= .

and an ICL program whose text resides in the file BEGIN.ICL. The "." <EXPR> can be used only in the context of point generation. The "." <EXPR> can be combined only with REALs and only via + or -. Thus, for example, the following are illegal:

.+. # 5

.*3 # 5

.+5-. # 5

Misuse of the "." <EXPR> comes up as a datatype error.

To aid the ICL program which implements relative points, ICL has the following built in primitive datatypes:

PRELX *a point relative in X,*

PRELY *a point relative in Y, and*

PRELB *a point relative in Both*

Instances of these datatypes are *not* instances of POINT. However, instances of these datatypes are represented like POINTs where the "." is interpreted as zero. Refer to the datatypes RELATIVE_POINT and SRP and refer to the coercion from SRP to SP as defined in the file BEGIN.ICL

The Debugging Package

This section documents ICL's debugging package. The debugging package is a set of ICL functions which provides services for on-line debugging. Each function will be described separately. Each function name begins with the characters ICLDDT. The function declarations can be found in the file BEGIN.ICL, the first file read into a freshly created ICL system. The functions are:

```
ICLDDT_HELP

ICLDDT_BT

ICLDDT_WHAT_FUNCTIONS

ICLDDT_BREAK_ON(FW)
ICLDDT_BREAK_OFF(FW)

ICLDDT_TRACE_ON(FW)
ICLDDT_TRACE_OFF(FW)

ICLDDT_INIT_LOCALS_ON
ICLDDT_INIT_LOCALS_OFF

ICLDDT_STACK_CHECKING_ON
ICLDDT_STACK_CHECKING_OFF

ICLDDT_KILL
```

In addition to these functions, the debugging package can be entered via the ↑C-handler's *Abort* command.

Throughout the rest of this section, the term *function* will encompass both functions and coercions.

ICLDDT_HELP *or simply* HELP

Enter the debugging package. The following message will be printed on the user's terminal:

Help! From within function *function name*

The *function name* is the name of the function containing the call to ICLDDT_HELP. However, if ICLDDT_HELP is called from within a process, i.e., program text contained between the symbols // and \\, then *function name* refers to the most recently entered ICL function: The message

Help! From within function (SKIPPING OVER A SUSFUNC)

will appear a number of times before the message with *function name* appears. The number of appearences equals the number of nested process invocations between the call to *function name* and the call to ICLDDT_HELP. The term SUSFUNC (SUSpendable FUNCtion) is another name for *process*.

After the "Help!" message is printed, an asterisk will appear, signalling that a new incarnation of ICL is ready to receive input from the user. The user is now free to interact with ICL as he would at any other time. The user has access to all functions, coercions, and datatypes. The user also has access to all global variables and to the arguments of *function name*. The user does not have access to any other local variables, unfortunately. The user will typically examine variables by printing them or by calling functions which can give him more information. The user can assign new values to variables if he wishes.

The user leaves this new incarnation of ICL by typing a ↑Z (control-Z). Upon receipt of ↑Z, ICLDDT_HELP returns and program execution continues where it left off. Variables assigned new values retain their new values. However, all functions, coercions, datatypes and variables *declared* during the new incarnation are lost upon the ↑Z

If ICLDDT_HELP is called from within a new incarnation of ICL, a still newer incarnation of ICL is created. In newer incarnations, the user has access to all accessable variables of the previous incarnations plus those variables which are arguments to *function name*. If an argument to *function name* has the same name as another accessable variable, the argument to *function name* takes precedence; the user loses access to the old meaning for the argument name.

Throughout this manual, the phrase *Enter the debugging package* refers to an automatic call to ICLDDT_HELP. For example, the rule STRSEL, string indexing, states that the sentence

S[N] := <EXPR> ;

will automatically enter the debugging package if N is greater than the length of the string S. Any rule which conditionally enters the debugging package acts as a no-op if it does indeed enter the debugging package. Once the debugging package is entered, the user can see how his program arrived there by invoking the function ICLDDT_BT. When the user types a ↑Z, his program resumes execution where it left off. If the user wishes to cancel his program rather than to let it continue execution, he should invoke the function

ICLDDT_KILL before he types the ↑Z.

Example:

The declaration

```
DEFINE  LOG(X:REAL)=REAL:
     IF X =< 0 THEN  DO ICLDDT_HELP; GIVE 0
               ELSE  <EXPR>  FI
ENDDEFN
```

defines LOG to be a function which maps a REAL to a REAL. If LOG is ever called with a non-positive number, ICLDDT_HELP will be called.

Let us suppose that LOG is called with a non-positive number. ICL will print the message

```
Help!  From within function  LOG(X:REAL)=REAL
```

Now, if the user types

```
WRITE(X);↑G
```

he will see the non-positive argument to LOG. If the user types

```
ICLDDT_BT;↑G
```

he will see the function calling sequence which finally called LOG with the bad value. When the user types a ↑Z, execution will resume and LOG will return a 0. If the user had typed

```
ICLDDT_KILL;↑G
```

before he typed the ↑Z, LOG would not return and his program would be cancelled. The next asterisk he would see would be prompted by the previous incarnation.

ICLDDT_BT

Print a backtrace of function calls. ICLDDT_BT prints each function name in order from the most recently called function to the earliest function call. For example, suppose the user has declared

```
DEFINE  F1(X:REAL)=REAL:  LOG(X)*20  ENDDEFN
DEFINE  F2(R:REAL)=REAL:  F1(X) + 5  ENDDEFN
```

If the user types

```
WRITE( F1( -COS(0) ) );↑G
```

ICLDDT_HELP will be called from within LOG because LOG will have received a negative argument. The user sees

```
Help!  From within function  LOG(X:REAL)=REAL
```

Now, if the user types

```
ICLDDT_BT;↑G
```

he will see the backtrace

```
LOG(X:REAL)=REAL
F1(X:REAL)=REAL
F2(R:REAL)=REAL
```

This says that LOG was called from within F1 and that F1 was called from within F2. The function WRITE does not appear in the backtrace becaues WRITE hasn't been called yet. Recall that in ICL, a function's parameter is evaluated before the function is entered. The error occured during the evaluation of WRITE's parameter and not within the function WRITE itself.

Each line of the backtrace has in addition to the function name, two octal numbers. These octal numbers are not shown in the example above. The first octal number is the address of a stack frame and the second octal number is the address of the function. The user will typically ignore these numbers. However, for compiler bugs, these numbers are useful for the ICL maintenance person.

The following is an unfortunate feature which should be undone someday: The user has access to the parameters of only the top function in the backtrace and he has no access to the parameters of the other functions listed in the backtrace.

Some lines of a backtrace may be of the form

(SUSFUNC)

Such a line refers to a process call. Refer to the treatment for processes in the documentation for the function ICLDDT_HELP.

ICLDDT_WHAT_FUNCTIONS

Print the name of each defined ICL function. Preceding each function name appears the octal address of the function. This octal address is useful for identifying the function for the trace and break facilities. Functions which were defined with the MACRO-10 form are not included in the listing. Functions are listed in the reverse order of definition, e.g., the most recently defined function appears first.

ICLDDT_BREAK_ON(FW) *and* ICLDDT_BREAK_ON

Set a breakpoint at the entry and exit of the function whose address is FW. If no parameter is specified, then set a breakpoint at the entry and exit of each and every currently defined ICL function.

The datatype FW is declared in the file BEGIN.ICL by

```
TYPE  FW= LOGICAL(36);
```

Instances of FW are created as described with the rule ELOG: A function address may be specified in octal with the form

L(*the function's octal address*)

The address of a function can be found via the function ICLDDT_WHAT_FUNCTIONS.

Having set a breakpoint at a function's entry and exit, the debugging package will be entered each time the function is entered or left. Upon entrance to the function, ICL prints the message

In Break Package: Entering *function name*

Upon leaving the function, ICL prints the message

In Break Package: Leaving *function name*

After either message is printed, an asterisk will appear, signalling that a new incarnation of ICL is ready to receive input from the user. At this point, the user is free to interact with ICL as he would at any other time. The situation is identical to the situation created by the function ICLDDT_HELP except for the following: If *function name* is being left and not entered, the user is *not* given access to the function's parameters. The user's

access rights depend on whether the function is being entered or left.

If the function is being entered, the user has access to the function's parameters. If the user assigns new values to the parameters, the function will execute exactly as though it were called with the newly assigned values. When the user types a ↑Z, the new incarnation of ICL dies and execution resumes by actually entering the function.

If the function is being left and if the function returns a value, the user has access to the variable named ICLDDT_RETURN. This special variable contains the value being returned by the function. If the user assigns a new value to the variable, his program will execute exactly as though the function actually returned the newly assigned value. When the user types a ↑Z, the new incarnation of ICL dies and execution resumes by actually leaving the function.

ICLDDT_BREAK_OFF(FW) *and* ICLDDT_BREAK_OFF

Undo ICLDDT_BREAK_ON. Remove the breakpoints from the function whose address is FW. If no parameter is specified, remove the breakpoints from all functions.

ICLDDT_TRACE_ON(FW) *and* ICLDDT_TRACE_ON

Trace the function whose address is FW. If no parameter is specified, trace all currently defined ICL functions.

A traced function prints its name each time it is entered and it prints a backslash each time it is left. Execution is not interrupted. The dynamic nesting of functions is communicated by the indentation of the trace information.

ICLDDT_TRACE_OFF(FW) *and* ICLDDT_TRACE_OFF

Undo ICLDDT_TRACE_ON. The function whose address is FW is no longer traced. If no parameter is specified, all functions will no longer be traced.

ICLDDT_INIT_LOCALS_ON

Set up all currently defined ICL functions so that upon entry, they initialize all their local variables to NIL, 0, or FALSE.

In general, functions do not take the time to zero their locals. If, by chance, the user forgets to initialize a local variable and if that variable becomes a part of a newly created structure, the newly created structure may very well contain a garbage value. All sorts of system error messages can ensue and ICL might crash at some unpredictable time in the future.

If the user ever gets a system error message, he should try rerunning his program having first invoked ICLDDT_INIT_LOCALS_ON. If his program runs without system errors, chances are that he forgot to initialize a variable somewhere. The function ICLDDT_STACK_CHECKING_ON is another quieter of system error messages.

ICLDDT_INIT_LOCALS_OFF

Undo ICLDDT_INIT_LOCALS_ON.

ICLDDT_STACK_CHECKING_ON

Set up all currently defined ICL functions so that upon entry, they check the stack for overflow.

In general, functions do not check for stack overflow. An infinite loop via recursion will surely overflow the stack. Once the stack has overflowed, the ICL system is lost. A stack overflow will typically announce itself by the execution of an illegal instruction.

If the user has invoked ICLDDT_STACK_CHECKING_ON, when the stack is about to overflow, the message

?STKCHK: Runtime stack nearing overflow

will appear and the debugging package will be entered as though called by ICLDDT_HELP. At this point, the user can invoke ICLDDT_BT to see the lengthy calling sequence that has filled the stack. The user can resume execution by typing a ↑Z or he can safely abort execution by invoking ICLDDT_KILL before typing the ↑Z.

ICLDDT_STACK_CHECKING_OFF

Undo ICLDDT_STACK_CHECKING_ON

ICLDDT_KILL

Abort the execution of the program in the previous incarnation of ICL. That is, ICLDDT_KILL sets an internal flag so that upon termination of the current incarnation, i.e., upon typing a ↑Z, the

program running in the newly current incarnation aborts.

Warning:

The program is aborted by simply resetting the top-of-stack pointer. This means that some global variables may not be reset properly. For example, variables specified in the HOLDING form, the rule HOLDIT, might not have their old values restored.

The ↑C-Handler's Abort Command

The only asynchronous entry to the debugging package is thru the ↑C-handler's *Abort* command. Any time an ICL program is running, the user can intercept its execution by typing ↑C A. The ↑C-handler's *Abort* command prints the message

    Waiting for function call ...

and resumes execution. As soon as the running ICL program either enters or leaves a function, ICL enters the debugging package exactly as though that particular function had had breakpoints previously set by ICLDDT_BREAK_ON. The breakpoints created by the ↑C's *Abort* command are only temporary: The function does not retain the breakpoints unless the function already had breakpoints previous to the *Abort*.

Rules Sorted by Part-of-speech

Page	Name			
341	AFORID:	⟨AFOR⟩	::=	FOR ⟨ID⟩
341	AFORFR:	⟨AFOR⟩	::=	⟨AFOR⟩ FROM ⟨EXPR⟩
341	AFORTO:	⟨AFOR⟩	::=	⟨AFOR⟩ TO ⟨EXPR⟩
341	AFORBY:	⟨AFOR⟩	::=	⟨AFOR⟩ BY ⟨EXPR⟩
341	AFORIN:	⟨AFOR⟩	::=	⟨AFOR⟩ IN ⟨EXPR⟩
341	AFORIS:	⟨AFOR⟩	::=	⟨AFOR⟩ IN* ⟨EXPR⟩
246	ARGS3:	⟨ARGS⟩	::=	⟨ARGSX⟩)
246	ARGS1:	⟨ARGSX⟩	::=	(⟨EXPR⟩
246	ARGS2:	⟨ARGSX⟩	::=	⟨ARGSX⟩ , ⟨EXPR⟩
265	ASN1:	⟨ASN⟩	::=	⟨ID⟩ ;
265	ASNRHS:	⟨ASN⟩	::=	⟨ID⟩ ⟨SSRHS⟩
265	ASNX:	⟨ASN⟩	::=	⟨ASN⟩ ⟨ASN⟩
262	DCOUGH:	⟨BEXPR⟩	:::=	BEGIN ⟨DECL⟩ ⟨EXPR⟩ END
262	DCOUGH:	⟨BEXPR⟩	:::=	BEGIN ⟨EXPR⟩ ⟨DECL⟩ END
328	BIF1:	⟨BIF⟩	::=	IF
328	BIF2:	⟨BIF⟩	:::=	⟨BIF⟩ ⟨EXPR⟩ THEN ⟨SS⟩ EF
213	BIF1:	⟨BIFE⟩	::=	IF
213	BIF2:	⟨BIFE⟩	::=	⟨BIFE⟩ ⟨EXPR⟩ THEN ⟨EXPR⟩ EF
275	BOPADD:	⟨BOP⟩	::=	+
275	BOPSUB:	⟨BOP⟩	::=	-
275	BOPMUL:	⟨BOP⟩	::=	*
275	BOPDIV:	⟨BOP⟩	::=	/
275	BOPEXP:	⟨BOP⟩	::=	↑
276	BOPAND:	⟨BOP⟩	::=	&
276	BOPOR:	⟨BOP⟩	::=	!
276	BOPXOR:	⟨BOP⟩	::=	XOR
276	BOPBIT:	⟨BOP⟩	::=	BIT
277	BOPLSL:	⟨BOP⟩	::=	SHIFTL
277	BOPLSR:	⟨BOP⟩	::=	SHIFTR
277	BOPMIN:	⟨BOP⟩	::=	MIN
277	BOPMAX:	⟨BOP⟩	::=	MAX
278	BSHARP:	⟨BOP⟩	::=	#
279	COMPEQ:	⟨BOP⟩	::=	=
279	COMPNE:	⟨BOP⟩	::=	<>
279	COMPGT:	⟨BOP⟩	::=	>
279	COMPGE:	⟨BOP⟩	::=	>=
279	COMPLT:	⟨BOP⟩	::=	<
279	COMPLE:	⟨BOP⟩	::=	=<
280	BOPSTR:	⟨BOP⟩	::=	$>
280	BOPSTC:	⟨BOP⟩	::=	$$
280	BOPSTL:	⟨BOP⟩	::=	<$
281	BOPBID:	⟨BOP⟩	::=	\ ⟨ID⟩
404	SEMNOP:	⟨BOP⟩	::=	\ ⟨FID⟩

```
333  DCOUGH: ⟨BSS⟩    :::=  BEGIN ⟨DECL⟩ ⟨SS⟩ END
333  DCOUGH: ⟨BSS⟩    :::=  BEGIN ⟨SS⟩ ⟨DECL⟩ END

192          ⟨CTYPE⟩  ::=   ⟨IDLIST⟩ : ⟨TYPE⟩
192          ⟨CTYPE⟩  ::=   ⟨CTYPE⟩ ⟨CTYPE⟩

196          ⟨DECL⟩   ::=   ⟨TDECL⟩
199          ⟨DECL⟩   ::=   ⟨VDECL⟩
203          ⟨DECL⟩   :::=  DEFINE ⟨ID⟩ : ⟨SS⟩ ENDDEFN
203          ⟨DECL⟩   :::=  DEFINE ⟨ID⟩ = ⟨TYPE⟩ : ⟨EXPR⟩ ENDDEFN
203          ⟨DECL⟩   :::=  DEFINE ⟨ID⟩ ( ⟨CTYPE⟩ ) : ⟨SS⟩ ENDDEFN
203          ⟨DECL⟩   :::=  DEFINE ⟨ID⟩ ( ⟨CTYPE⟩ ) = ⟨TYPE⟩ : ⟨EXPR⟩
ENDDEFN
206          ⟨DECL⟩   :::=  DEFINE ⟨ID⟩ : MACRO-10( ⟨QS⟩ )
206          ⟨DECL⟩   :::=  DEFINE ⟨ID⟩ = ⟨TYPE⟩ : MACRO-10( ⟨QS⟩ )
206          ⟨DECL⟩   :::=  DEFINE ⟨ID⟩ ( ⟨CTYPE⟩ ) : MACRO-10( ⟨QS⟩ )
206          ⟨DECL⟩   :::=  DEFINE ⟨ID⟩ ( ⟨CTYPE⟩ ) = ⟨TYPE⟩ : MACRO-10(
⟨QS⟩ )
206          ⟨DECL⟩   :::=  LET ⟨ID⟩ BECOME ⟨ID⟩ BY ⟨EXPR⟩ ;
207          ⟨DECL⟩   :::=  LET ⟨ID⟩ BECOME ⟨ID⟩ BY MACRO-10( ⟨QS⟩ )
208          ⟨DECL⟩   ::=   ⟨DECL⟩ ⟨DECL⟩

213  EBIF:   ⟨EXPR⟩   :::=  ⟨BIFE⟩ ⟨EXPR⟩ THEN ⟨EXPR⟩ ELSE ⟨EXPR⟩ FI
217  ENU:    ⟨EXPR⟩   ::=   ⟨NU⟩
217  EQS:    ⟨EXPR⟩   ::=   ⟨QS⟩
217  ELOG:   ⟨EXPR⟩   ::=   L ( ⟨NU⟩ )
217  ELOG:   ⟨EXPR⟩   ::=   L ( ⟨NU⟩ ⟨NU⟩ )
218  EFNU:   ⟨EXPR⟩   ::=   a floating number
218  ETRU:   ⟨EXPR⟩   ::=   TRUE
219  EFALS:  ⟨EXPR⟩   ::=   FALSE
219  ENIL:   ⟨EXPR⟩   ::=   NIL
219  EID:    ⟨EXPR⟩   ::=   ⟨ID⟩
221  STRGEN: ⟨EXPR⟩   ::=   ( ⟨REXPR⟩
223  STRSEL: ⟨EXPR⟩   ::=   ⟨EXPR⟩ [ ⟨EXPR⟩ ]
225  ETAIL:  ⟨EXPR⟩   ::=   ⟨EXPR⟩ [ ⟨EXPR⟩ - ]
226  ERFRSH: ⟨EXPR⟩   ::=   REFRESH ( ⟨EXPR⟩ )
226  ERFRSH: ⟨EXPR⟩   ::=   REFRESH ! ⟨EXPR⟩ )
227  EREVRS: ⟨EXPR⟩   ::=   REVERSE ( ⟨EXPR⟩ )
227  EREVRS: ⟨EXPR⟩   ::=   REVERSE ! ⟨EXPR⟩ )
228  RGENF:  ⟨EXPR⟩   ::=   ⟨RECX⟩
229  RSELQ:  ⟨EXPR⟩   ::=   ⟨EXPR⟩ . ⟨ID⟩
231  PTSELX: ⟨EXPR⟩   ::=   ⟨EXPR⟩ . X
231  PTSELY: ⟨EXPR⟩   ::=   ⟨EXPR⟩ . Y
233  ECASEE: ⟨EXPR⟩   :::=  CASE ⟨EXPR⟩ OF ⟨EXPRV⟩
235  TYPDIS: ⟨EXPR⟩   ::=   ⟨ID⟩ :: ⟨EXPR⟩
239  ECASE:  ⟨EXPR⟩   :::=  CASE ⟨ID⟩ OF ⟨EXPRV⟩
244  TYPDIS: ⟨EXPR⟩   ::=   ⟨ID⟩ :: ⟨EXPR⟩
246  ECALLP: ⟨EXPR⟩   ::=   ⟨ID⟩ ⟨ARGS⟩
248  SEMNOP: ⟨EXPR⟩   ::=   ( ⟨EXPR⟩ )
248  EBOP:   ⟨EXPR⟩   ::=   ⟨EXPR⟩ ⟨BOP⟩ ⟨EXPR⟩
248  EBOPG:  ⟨EXPR⟩   ::=   ⟨EXPR⟩ ⟨BOP⟩ ⟨EXPR⟩
250  EUOP:   ⟨EXPR⟩   ::=   ⟨UOP⟩ ⟨EXPR⟩
250  EUOP:   ⟨EXPR⟩   ::=   ⟨EXPR⟩ ⟨RHUOP⟩
250  EUOPG:  ⟨EXPR⟩   ::=   ⟨EXPR⟩ ⟨RHUOP⟩
251  EBOPQ:  ⟨EXPR⟩   ::=   ⟨BOP⟩ ⟨EXPR⟩ ⟨QUANT⟩
```

```
251  EBOPQ:  <EXPR>   ::=   <QUANT> GIVE <BOP> <EXPR> END
251  EBOPQ:  <EXPR>   ::=   <QUANT> <BOP> <EXPR>
254  QBOOL1: <EXPR>   ::=   <QUANT> <EXPR> <QBOOL>
254  QBOOL1: <EXPR>   ::=   <QUANT> <QBOOL> <EXPR>
254  QBOOL1: <EXPR>   ::=   <QUANT> GIVE <QBOOL> <EXPR> END
254  QBOOL1: <EXPR>   ::=   <QBOOL> <EXPR> <QUANT>
258  EGIVE:  <EXPR>   ::=   DO <SS> GIVE <EXPR>
258  EGRAB:  <EXPR>   ::=   GIVING <EXPR> DO <SS> END
258  EGRAB:  <EXPR>   ::=   DO <SS> GRABBING <EXPR>
259  SETQX:  <EXPR>   :::=  ( <EXPR> <SSRHS> )
262  EDECL:  <EXPR>   ::=   <BEXPR>
265  HOLDIT: <EXPR>   :::=  HOLDING <ASN> GIVE <EXPR> ENDHOLD
270  EAT:    <EXPR>   ::=   @ ( <EXPR> )
272  ECOPY:  <EXPR>   ::=   COPY ( <EXPR> )
272  ECOPY:  <EXPR>   ::=   COPY ! <EXPR> )
274  EDEF:   <EXPR>   ::=   DEFINED ( <EXPR> )
372  EIDID:  <EXPR>   ::=   % <ID>
377  PUBLC:  <EXPR>   ::=   PUBLICIZE:::( <EXPR> )
377  PRIVY:  <EXPR>   ::=   <ID> :::( <EXPR> )
388  SUSF1:  <EXPR>   :::=  <SUSB> <EXPR> \\
388  SUSF1S: <EXPR>   :::=  <SUSB> <SS> \\
403  FID1:   <EXPR>   ::=   <FID>
403  FID2:   <EXPR>   ::=   <FID> <ARGS>
403  FID3:   <EXPR>   ::=   <FID> ;
403  FID4:   <EXPR>   ::=   <FID> <ARGS> ;
406  QSUS2:  <EXPR>   ::=   <QSUSB> <QSUSE>
406  QSUS3:  <EXPR>   ::=   <QSUSB> ( <IDLIST> ) <QSUSE>
406  QSUS4:  <EXPR>   ::=   <QSUSB> <SARGS> <QSUSE>
406  QSUS5:  <EXPR>   ::=   <QSUSB> ( <IDLIST> ) <SARGS> <QSUSE>

233  EVCASE: <EXPRV>  :::=  <ID> : <EXPR> ENDCASE
233  EVCASB: <EXPRV>  :::=  <ID> : <EXPR> <EXPRV>
239  EVCASE: <EXPRV>  :::=  <ID> : <EXPR> ENDCASE
239  EVCASB: <EXPRV>  :::=  <ID> : <EXPR> <EXPRV>

403  SEMNOP: <FID>    ::=   <* <EXPR> *>

290  SEMNOP: <GUOP>   ::=   <UOP>
290  SEMNOP: <GUOP>   ::=   <RHUOP>

175          <IDLIST>:::=   <ID>
175          <IDLIST>:::=   <IDLIST> , <ID>

290  KUOP1:  <KUOP>   ::=   <GUOP> ;
290  KUOP2:  <KUOP>   ::=   <GUOP> <KUOP>

254  QBALW:  <QBOOL>  ::=   ALWAYS
254  QBNVR:  <QBOOL>  ::=   NEVER
254  QBEXS:  <QBOOL>  ::=   EXISTS
254  QBEXS:  <QBOOL>  ::=   THERE_IS

406  SEMNOP: <QSUSB>  ::=   //: <ID>

406  - - -:  <QSUSE>  ::=   \\
406  - - -:  <QSUSE>  ::=   ;\\
```

```
338  QWHIL:   ⟨QUANT⟩ ::=   WHILE ⟨EXPR⟩ ;
339  QUNTL:   ⟨QUANT⟩ ::=   UNTIL ⟨EXPR⟩ ;
340  REPET:   ⟨QUANT⟩ ::=   REPEAT ⟨EXPR⟩ ;
341  AFORGO:  ⟨QUANT⟩ ::=   ⟨AFOR⟩ ;
346  QFORE:   ⟨QUANT⟩ ::=   FOR ⟨EXPR⟩ $E ⟨EXPR⟩ ;
348  QFORC:   ⟨QUANT⟩ ::=   FOR ⟨EXPR⟩ $C ⟨EXPR⟩ ;
361  QOR:     ⟨QUANT⟩ ::=   ⟨QUANT⟩ !! ⟨QUANT⟩
361  QAND:    ⟨QUANT⟩ ::=   ⟨QUANT⟩ && ⟨QUANT⟩
361  QTHEN:   ⟨QUANT⟩ ::=   ⟨QUANT⟩ THEN ⟨QUANT⟩
366  QWITH:   ⟨QUANT⟩ ::=   ⟨QUANT⟩ WITH ⟨EXPR⟩ ;
366  QINH:    ⟨QUANT⟩ ::=   ⟨QUANT⟩ INHIBIT_IF ⟨EXPR⟩ ;
366  QRES:    ⟨QUANT⟩ ::=   ⟨QUANT⟩ RESET_IF ⟨EXPR⟩ ;
366  QECH:    ⟨QUANT⟩ ::=   ⟨QUANT⟩ EACH_DO ⟨SS⟩ ;
366  QFTM:    ⟨QUANT⟩ ::=   ⟨QUANT⟩ FIRST_DO ⟨SS⟩ ;
366  QOTH:    ⟨QUANT⟩ ::=   ⟨QUANT⟩ OTHER_DO ⟨SS⟩ ;
366  QFST:    ⟨QUANT⟩ ::=   ⟨QUANT⟩ INITIALLY ⟨SS⟩ ;
366  QFIN:    ⟨QUANT⟩ ::=   ⟨QUANT⟩ FINALLY_DO ⟨SS⟩ ;

221  RFUNC:   ⟨RANGE⟩ ::=   $ ⟨EXPR⟩ ⟨QUANT⟩
221  RFUNC:   ⟨RANGE⟩ ::=   COLLECT ⟨EXPR⟩ ⟨QUANT⟩
221  RFUNC:   ⟨RANGE⟩ ::=   ⟨QUANT⟩ $ ⟨EXPR⟩
221  RFUNC:   ⟨RANGE⟩ ::=   ⟨QUANT⟩ COLLECT ⟨EXPR⟩

228  SEMNOP:  ⟨RECX⟩  ::=   [ ⟨RECXT⟩

228  RGENQ:   ⟨RECXT⟩ ::=   ⟨ID⟩ : ⟨EXPR⟩ ]
228  RGEN1:   ⟨RECXT⟩ ::=   ⟨ID⟩ : ⟨EXPR⟩ ⟨RECXT⟩

221  SEXP:    ⟨REXPR⟩ ::=   ⟨EXPR⟩ }
221  SEMNOP:  ⟨REXPR⟩ ::=   ⟨RANGE⟩ }
221  SCRNG:   ⟨REXPR⟩ ::=   ⟨RANGE⟩ ; ⟨REXPR⟩
221  SCEXP:   ⟨REXPR⟩ ::=   ⟨EXPR⟩ ; ⟨REXPR⟩
221  SCCONX:  ⟨REXPR⟩ ::=   ⟨EXPR⟩ ;* ⟨REXPR⟩

286  UOPBID:  ⟨RHUOP⟩ ::=   \ ⟨ID⟩
406  SEMNOP:  ⟨RHUOP⟩ ::=   \ ⟨FID⟩

406  ARGS3:   ⟨SARGS⟩ ::=   ⟨SARGX⟩ ]

406  ARGS2:   ⟨SARGX⟩ ::=   ⟨SARGX⟩ , ⟨EXPR⟩
406  ARGS1:   ⟨SARGX⟩ ::=   [ ⟨EXPR⟩

288  SSASS:   ⟨SS⟩    ::=   ⟨EXPR⟩ ⟨SSRHS⟩
328  EBIF:    ⟨SS⟩    ::::= ⟨BIF⟩ ⟨EXPR⟩ THEN ⟨SS⟩ ELSE ⟨SS⟩ FI
328  SBIF:    ⟨SS⟩    ::::= ⟨BIF⟩ ⟨EXPR⟩ THEN ⟨SS⟩ FI
329  ECASEE:  ⟨SS⟩    ::::= CASE ⟨EXPR⟩ OF ⟨SSV⟩
330  ECASE:   ⟨SS⟩    ::::= CASE ⟨ID⟩ OF ⟨SSV⟩
331  HOLDIT:  ⟨SS⟩    ::::= HOLDING ⟨ASN⟩ DO ⟨SS⟩ ENDHOLD
333  EDECL:   ⟨SS⟩    ::::= ⟨BSS⟩
334  SSQ:     ⟨SS⟩    ::=   DO ⟨SS⟩ ⟨QUANT⟩
334  SSQ:     ⟨SS⟩    ::=   ⟨QUANT⟩ DO ⟨SS⟩ END
334  SSCALP:  ⟨SS⟩    ::=   ⟨ID⟩ ⟨ARGS⟩ ;
334  SSICAL:  ⟨SS⟩    ::=   ⟨ID⟩ ;
336  SSSS:    ⟨SS⟩    ::=   ⟨SS⟩ ⟨SS⟩
```

```
288  SSRHS1: <SSRHS> ::=    := <EXPR> ;
288  SSRHS2: <SSRHS> ::=    ::= <BOP> <EXPR> ;
288  SSRHS3: <SSRHS> ::=    ::= <EXPR> <BOP> ;
290  SSRHS4: <SSRHS> ::=    ::= <KUOP>
290  SSRHS4: <SSRHS> ::=    ::= <KUOP>

329  EVCASE: <SSV>   :::=  <ID> : <SS> ENDCASE
329  EVCASB: <SSV>   :::=  <ID> : <SS> <SSV>

388  SUSB1:  <SUSB>  ::=   //
388  SUSB2:  <SUSB>  ::=   <SUSB> ( <CTYPE> )
388  SUSB3:  <SUSB>  ::=   <SUSB> [ <ASN> ]
388  SUSB4:  <SUSB>  ::=   <SUSB> { <ASN> }

196          <TDECL> :::=  TYPE <ID> = <TYPE> ;
196          <TDECL> :::=  <TDECL> <ID> = <TYPE> ;

190          <TYPE>  ::=   INT
190          <TYPE>  ::=   REAL
190          <TYPE>  ::=   POINT
190          <TYPE>  ::=   BOOL
190          <TYPE>  ::=   CHAR
190          <TYPE>  ::=   QS
190          <TYPE>  ::=   LOGICAL ( <NU> )
192          <TYPE>  ::=   { <TYPE> }
192          <TYPE>  ::=   [ <CTYPE> ]
193          <TYPE>  ::=   EITHER <VTYPE> ENDOR
193          <TYPE>  ::=   SCALAR ( <IDLIST> )
193          <TYPE>  ::=   <ID>
372          <TYPE>  ::=   ID
375          <TYPE>  ::=   PRIVATE <TYPE>
387          <TYPE>  ::=   // \\
387          <TYPE>  ::=   // <TYPE> \\
387          <TYPE>  ::=   // ( <IDLIST> ) \\
387          <TYPE>  ::=   // <TYPE> ( <IDLIST> ) \\

284  UOPMIN: <UOP>   ::=   -
285  UTALLY: <UOP>   ::=   TALLY
285  ULFTZO: <UOP>   ::=   LEFTZEROS
285  UENCOD: <UOP>   ::=   ENCODE
285  UDECOD: <UOP>   ::=   DECODE
286  UUNARY: <UOP>   ::=   UNARY
285  UNORM:  <UOP>   ::=   NORM
285  UBITSW: <UOP>   ::=   BITSWAP

199          <VDECL> :::=  VAR <IDLIST> = <TYPE> ;
199          <VDECL> :::=  <VDECL> <IDLIST> = <TYPE> ;

193          <VTYPE> ::=   <IDLIST> = <TYPE>
193          <VTYPE> ::=   <VTYPE> <VTYPE>

178          <file>  ::=   <ID>
178          <file>  ::=   <ID> .
178          <file>  ::=   <ID> . <ID>
178          <file>  ::=   <file> - <file>
```

```
178         <file>  ::=  <ID> : <file> ;
178         <file>  ::=  <file> [ <NU> , <ID> ]

175         (       ::=  [ )

175         )       ::=  ( ]
```

Rules Sorted by Name

Page	Name		
175	⟨IDLIST⟩	::=	⟨ID⟩
175	⟨IDLIST⟩	::=	⟨IDLIST⟩ , ⟨ID⟩
175	{	::=	[ )
175	}	::=	( ]
178	⟨file⟩	::=	⟨ID⟩
178	⟨file⟩	::=	⟨ID⟩ .
178	⟨file⟩	::=	⟨ID⟩ . ⟨ID⟩
178	⟨file⟩	::=	⟨file⟩ - ⟨file⟩
178	⟨file⟩	::=	⟨ID⟩ : ⟨file⟩ ;
178	⟨file⟩	::=	⟨file⟩ [⟨NU⟩ , ⟨ID⟩]
190	⟨TYPE⟩	::=	INT
190	⟨TYPE⟩	::=	REAL
190	⟨TYPE⟩	::=	POINT
190	⟨TYPE⟩	::=	BOOL
190	⟨TYPE⟩	::=	CHAR
190	⟨TYPE⟩	::=	QS
190	⟨TYPE⟩	::=	LOGICAL (⟨NU⟩)
192	⟨TYPE⟩	::=	{ ⟨TYPE⟩ }
192	⟨TYPE⟩	::=	[⟨CTYPE⟩]
192	⟨CTYPE⟩	::=	⟨IDLIST⟩ : ⟨TYPE⟩
192	⟨CTYPE⟩	::=	⟨CTYPE⟩ ⟨CTYPE⟩
193	⟨TYPE⟩	::=	EITHER ⟨VTYPE⟩ ENDOR
193	⟨VTYPE⟩	::=	⟨IDLIST⟩ = ⟨TYPE⟩
193	⟨VTYPE⟩	::=	⟨VTYPE⟩ ⟨VTYPE⟩
193	⟨TYPE⟩	::=	SCALAR (⟨IDLIST⟩)
193	⟨TYPE⟩	::=	⟨ID⟩
196	⟨DECL⟩	::=	⟨TDECL⟩
196	⟨TDECL⟩	::=	TYPE ⟨ID⟩ = ⟨TYPE⟩ ;
196	⟨TDECL⟩	::=	⟨TDECL⟩ ⟨ID⟩ = ⟨TYPE⟩ ;
199	⟨DECL⟩	::=	⟨VDECL⟩
199	⟨VDECL⟩	::=	VAR ⟨IDLIST⟩ = ⟨TYPE⟩ ;
199	⟨VDECL⟩	::=	⟨VDECL⟩ ⟨IDLIST⟩ = ⟨TYPE⟩ ;
203	⟨DECL⟩	::=	DEFINE ⟨ID⟩ : ⟨SS⟩ ENDDEFN
203	⟨DECL⟩	::=	DEFINE ⟨ID⟩ = ⟨TYPE⟩ : ⟨EXPR⟩ ENDDEFN
203	⟨DECL⟩	::=	DEFINE ⟨ID⟩ (⟨CTYPE⟩) : ⟨SS⟩ ENDDEFN
203	⟨DECL⟩	::=	DEFINE ⟨ID⟩ (⟨CTYPE⟩) = ⟨TYPE⟩ : ⟨EXPR⟩ ENDDEFN
206	⟨DECL⟩	::=	DEFINE ⟨ID⟩ : MACRO-10(⟨QS⟩)
206	⟨DECL⟩	::=	DEFINE ⟨ID⟩ = ⟨TYPE⟩ : MACRO-10(⟨QS⟩)
206	⟨DECL⟩	::=	DEFINE ⟨ID⟩ (⟨CTYPE⟩) : MACRO-10(⟨QS⟩)
206	⟨DECL⟩	::=	DEFINE ⟨ID⟩ (⟨CTYPE⟩) = ⟨TYPE⟩ : MACRO-10(⟨QS⟩)
206	⟨DECL⟩	::=	LET ⟨ID⟩ BECOME ⟨ID⟩ BY ⟨EXPR⟩ ;
207	⟨DECL⟩	::=	LET ⟨ID⟩ BECOME ⟨ID⟩ BY MACRO-10(⟨QS⟩)
208	⟨DECL⟩	::=	⟨DECL⟩ ⟨DECL⟩
372	⟨TYPE⟩	::=	ID
375	⟨TYPE⟩	::=	PRIVATE ⟨TYPE⟩
387	⟨TYPE⟩	::=	// \\
387	⟨TYPE⟩	::=	// ⟨TYPE⟩ \\
387	⟨TYPE⟩	::=	// (⟨IDLIST⟩) \\

```
387          <TYPE>  ::=   // <TYPE> ( <IDLIST> ) \\

406  - - -:  <QSUSE> ::=   \\
406  - - -:  <QSUSE> ::=   ;\\

341  AFORBY: <AFOR>  ::=   <AFOR> BY <EXPR>
341  AFORFR: <AFOR>  ::=   <AFOR> FROM <EXPR>
341  AFORGO: <QUANT> ::=   <AFOR> ;
341  AFORID: <AFOR>  ::=   FOR <ID>
341  AFORIN: <AFOR>  ::=   <AFOR> IN <EXPR>
341  AFORIS: <AFOR>  ::=   <AFOR> IN* <EXPR>
341  AFORTO: <AFOR>  ::=   <AFOR> TO <EXPR>
246  ARGS1:  <ARGSX> ::=   ( <EXPR>
406  ARGS1:  <SARGX> ::=   [ <EXPR>
246  ARGS2:  <ARGSX> ::=   <ARGSX> , <EXPR>
406  ARGS2:  <SARGX> ::=   <SARGX> , <EXPR>
246  ARGS3:  <ARGS>  ::=   <ARGSX> )
406  ARGS3:  <SARGS> ::=   <SARGX> ]
265  ASN1:   <ASN>   ::=   <ID> ;
265  ASNRHS: <ASN>   ::=   <ID> <SSRHS>
265  ASNX:   <ASN>   ::=   <ASN> <ASN>

213  BIF1:   <BIFE>  ::=   IF
328  BIF1:   <BIF>   ::=   IF
213  BIF2:   <BIFE>  ::=   <BIFE> <EXPR> THEN <EXPR> EF
328  BIF2:   <BIF>   :::=  <BIF> <EXPR> THEN <SS> EF
275  BOPADD: <BOP>   ::=   +
276  BOPAND: <BOP>   ::=   &
281  BOPBID: <BOP>   ::=   \ <ID>
276  BOPBIT: <BOP>   ::=   BIT
275  BOPDIV: <BOP>   ::=   /
275  BOPEXP: <BOP>   ::=   ↑
277  BOPLSL: <BOP>   ::=   SHIFTL
277  BOPLSR: <BOP>   ::=   SHIFTR
277  BOPMAX: <BOP>   ::=   MAX
277  BOPMIN: <BOP>   ::=   MIN
275  BOPMUL: <BOP>   ::=   *
276  BOPOR:  <BOP>   ::=   !
280  BOPSTC: <BOP>   ::=   $$
280  BOPSTL: <BOP>   ::=   <$
280  BOPSTR: <BOP>   ::=   $>
275  BOPSUB: <BOP>   ::=   -
276  BOPXOR: <BOP>   ::=   XOR
278  BSHARP: <BOP>   ::=   #

279  COMPEQ: <BOP>   ::=   =
279  COMPGE: <BOP>   ::=   >=
279  COMPGT: <BOP>   ::=   >
279  COMPLE: <BOP>   ::=   =<
279  COMPLT: <BOP>   ::=   <
279  COMPNE: <BOP>   ::=   <>

262  DCOUGH: <BEXPR> :::=  BEGIN <DECL> <EXPR> END
262  DCOUGH: <BEXPR> :::=  BEGIN <EXPR> <DECL> END
333  DCOUGH: <BSS>   :::=  BEGIN <DECL> <SS> END
```

```
333  DCOUGH: ⟨BSS⟩    :::=  BEGIN ⟨SS⟩ ⟨DECL⟩ END

270  EAT:    ⟨EXPR⟩   ::=   @ ( ⟨EXPR⟩ )
213  EBIF:   ⟨EXPR⟩   :::=  ⟨BIFE⟩ ⟨EXPR⟩ THEN ⟨EXPR⟩ ELSE ⟨EXPR⟩ FI
328  EBIF:   ⟨SS⟩     :::=  ⟨BIF⟩ ⟨EXPR⟩ THEN ⟨SS⟩ ELSE ⟨SS⟩ FI
248  EBOP:   ⟨EXPR⟩   ::=   ⟨EXPR⟩ ⟨BOP⟩ ⟨EXPR⟩
248  EBOPG:  ⟨EXPR⟩   ::=   ⟨EXPR⟩ ⟨BOP⟩ ⟨EXPR⟩
251  EBOPQ:  ⟨EXPR⟩   ::=   ⟨BOP⟩ ⟨EXPR⟩ ⟨QUANT⟩
251  EBOPQ:  ⟨EXPR⟩   ::=   ⟨QUANT⟩ GIVE ⟨BOP⟩ ⟨EXPR⟩ END
251  EBOPQ:  ⟨EXPR⟩   ::=   ⟨QUANT⟩ ⟨BOP⟩ ⟨EXPR⟩
246  ECALLP: ⟨EXPR⟩   ::=   ⟨ID⟩ ⟨ARGS⟩
239  ECASE:  ⟨EXPR⟩   :::=  CASE ⟨ID⟩ OF ⟨EXPRV⟩
330  ECASE:  ⟨SS⟩     :::=  CASE ⟨ID⟩ OF ⟨SSV⟩
233  ECASEE: ⟨EXPR⟩   :::=  CASE ⟨EXPR⟩ OF ⟨EXPRV⟩
329  ECASEE: ⟨SS⟩     :::=  CASE ⟨EXPR⟩ OF ⟨SSV⟩
272  ECOPY:  ⟨EXPR⟩   ::=   COPY ( ⟨EXPR⟩ )
272  ECOPY:  ⟨EXPR⟩   ::=   COPY ! ⟨EXPR⟩ )
262  EDECL:  ⟨EXPR⟩   ::=   ⟨BEXPR⟩
333  EDECL:  ⟨SS⟩     :::=  ⟨BSS⟩
274  EDEF:   ⟨EXPR⟩   ::=   DEFINED ( ⟨EXPR⟩ )
219  EFALS:  ⟨EXPR⟩   ::=   FALSE
218  EFNU:   ⟨EXPR⟩   ::=   a floating number
258  EGIVE:  ⟨EXPR⟩   ::=   DO ⟨SS⟩ GIVE ⟨EXPR⟩
258  EGRAB:  ⟨EXPR⟩   ::=   GIVING ⟨EXPR⟩ DO ⟨SS⟩ END
258  EGRAB:  ⟨EXPR⟩   ::=   DO ⟨SS⟩ GRABBING ⟨EXPR⟩
219  EID:    ⟨EXPR⟩   ::=   ⟨ID⟩
372  EIDID:  ⟨EXPR⟩   ::=   % ⟨ID⟩
217  ELOG:   ⟨EXPR⟩   ::=   L ( ⟨NU⟩ )
217  ELOG:   ⟨EXPR⟩   ::=   L ( ⟨NU⟩ ⟨NU⟩ )
219  ENIL:   ⟨EXPR⟩   ::=   NIL
217  ENU:    ⟨EXPR⟩   ::=   ⟨NU⟩
217  EQS:    ⟨EXPR⟩   ::=   ⟨QS⟩
227  EREVRS: ⟨EXPR⟩   ::=   REVERSE ( ⟨EXPR⟩ )
227  EREVRS: ⟨EXPR⟩   ::=   REVERSE ! ⟨EXPR⟩ )
226  ERFRSH: ⟨EXPR⟩   ::=   REFRESH ( ⟨EXPR⟩ )
226  ERFRSH: ⟨EXPR⟩   ::=   REFRESH ! ⟨EXPR⟩ )
225  ETAIL:  ⟨EXPR⟩   ::=   ⟨EXPR⟩ [ ⟨EXPR⟩ - ]
218  ETRU:   ⟨EXPR⟩   ::=   TRUE
250  EUOP:   ⟨EXPR⟩   ::=   ⟨UOP⟩ ⟨EXPR⟩
250  EUOP:   ⟨EXPR⟩   ::=   ⟨EXPR⟩ ⟨RHUOP⟩
250  EUOPG:  ⟨EXPR⟩   ::=   ⟨EXPR⟩ ⟨RHUOP⟩
233  EVCASB: ⟨EXPRV⟩  :::=  ⟨ID⟩ : ⟨EXPR⟩ ⟨EXPRV⟩
239  EVCASB: ⟨EXPRV⟩  :::=  ⟨ID⟩ : ⟨EXPR⟩ ⟨EXPRV⟩
329  EVCASB: ⟨SSV⟩    :::=  ⟨ID⟩ : ⟨SS⟩ ⟨SSV⟩
233  EVCASE: ⟨EXPRV⟩  :::=  ⟨ID⟩ : ⟨EXPR⟩ ENDCASE
239  EVCASE: ⟨EXPRV⟩  :::=  ⟨ID⟩ : ⟨EXPR⟩ ENDCASE
329  EVCASE: ⟨SSV⟩    :::=  ⟨ID⟩ : ⟨SS⟩ ENDCASE

403  FID1:   ⟨EXPR⟩   ::=   ⟨FID⟩
403  FID2:   ⟨EXPR⟩   ::=   ⟨FID⟩ ⟨ARGS⟩
403  FID3:   ⟨EXPR⟩   ::=   ⟨FID⟩ ;
403  FID4:   ⟨EXPR⟩   ::=   ⟨FID⟩ ⟨ARGS⟩ ;

266  HOLDIT: ⟨EXPR⟩   :::=  HOLDING ⟨ASN⟩ GIVE ⟨EXPR⟩ ENDHOLD
331  HOLDIT: ⟨SS⟩     :::=  HOLDING ⟨ASN⟩ DO ⟨SS⟩ ENDHOLD
```

```
290  KUOP1:  <KUOP>   ::=   <GUOP> ;
290  KUOP2:  <KUOP>   ::=   <GUOP> <KUOP>

377  PRIVY:  <EXPR>   ::=   <ID> :::( <EXPR> )
231  PTSELX: <EXPR>   ::=   <EXPR> . X
231  PTSELY: <EXPR>   ::=   <EXPR> . Y
377  PUBLC:  <EXPR>   ::=   PUBLICIZE:::( <EXPR> )

361  QAND:   <QUANT>  ::=   <QUANT> && <QUANT>
254  QBALW:  <QBOOL>  ::=   ALWAYS
254  QBEXS:  <QBOOL>  ::=   EXISTS
254  QBEXS:  <QBOOL>  ::=   THERE_IS
254  QBNVR:  <QBOOL>  ::=   NEVER
254  QBOOL1: <EXPR>   ::=   <QUANT> <EXPR> <QBOOL>
254  QBOOL1: <EXPR>   ::=   <QUANT> <QBOOL> <EXPR>
254  QBOOL1: <EXPR>   ::=   <QUANT> GIVE <QBOOL> <EXPR> END
254  QBOOL1: <EXPR>   ::=   <QBOOL> <EXPR> <QUANT>
366  QECH:   <QUANT>  ::=   <QUANT> EACH_DO <SS> ;
366  QFIN:   <QUANT>  ::=   <QUANT> FINALLY_DO <SS> ;
348  QFORC:  <QUANT>  ::=   FOR <EXPR> $C <EXPR> ;
346  QFORE:  <QUANT>  ::=   FOR <EXPR> $E <EXPR> ;
366  QFST:   <QUANT>  ::=   <QUANT> INITIALLY <SS> ;
366  QFTM:   <QUANT>  ::=   <QUANT> FIRST_DO <SS> ;
366  QINH:   <QUANT>  ::=   <QUANT> INHIBIT_IF <EXPR> ;
361  QOR:    <QUANT>  ::=   <QUANT> !! <QUANT>
366  QOTH:   <QUANT>  ::=   <QUANT> OTHER_DO <SS> ;
366  QRES:   <QUANT>  ::=   <QUANT> RESET_IF <EXPR> ;
406  QSUS2:  <EXPR>   ::=   <QSUSB> <QSUSE>
406  QSUS3:  <EXPR>   ::=   <QSUSB> ( <IDLIST> ) <QSUSE>
406  QSUS4:  <EXPR>   ::=   <QSUSB> <SARGS> <QSUSE>
406  QSUS5:  <EXPR>   ::=   <QSUSB> ( <IDLIST> ) <SARGS> <QSUSE>
361  QTHEN:  <QUANT>  ::=   <QUANT> THEN <QUANT>
339  QUNTL:  <QUANT>  ::=   UNTIL <EXPR> ;
338  QWHIL:  <QUANT>  ::=   WHILE <EXPR> ;
366  QWITH:  <QUANT>  ::=   <QUANT> WITH <EXPR> ;

340  REPET:  <QUANT>  ::=   REPEAT <EXPR> ;
221  RFUNC:  <RANGE>  ::=   $ <EXPR> <QUANT>
221  RFUNC:  <RANGE>  ::=   COLLECT <EXPR> <QUANT>
221  RFUNC:  <RANGE>  ::=   <QUANT> $ <EXPR>
221  RFUNC:  <RANGE>  ::=   <QUANT> COLLECT <EXPR>
228  RGEN1:  <RECXT>  ::=   <ID> : <EXPR> <RECXT>
228  RGENF:  <EXPR>   ::=   <RECX>
228  RGENQ:  <RECXT>  ::=   <ID> : <EXPR> ]
229  RSELQ:  <EXPR>   ::=   <EXPR> . <ID>

328  SBIF:   <SS>     :::=  <BIF> <EXPR> THEN <SS> FI
221  SCCONX: <REXPR>  ::=   <EXPR> ;* <REXPR>
221  SCEXP:  <REXPR>  ::=   <EXPR> ; <REXPR>
221  SCRNG:  <REXPR>  ::=   <RANGE> ; <REXPR>
221  SEMNOP: <REXPR>  ::=   <RANGE> }
228  SEMNOP: <RECX>   ::=   [ <RECXT>
248  SEMNOP: <EXPR>   ::=   ( <EXPR> )
290  SEMNOP: <GUOP>   ::=   <UOP>
290  SEMNOP: <GUOP>   ::=   <RHUOP>
```

```
403  SEMNOP: <FID>    ::=   <* <EXPR> *>
404  SEMNOP: <BOP>    ::=   \ <FID>
405  SEMNOP: <RHUOP>  ::=   \ <FID>
406  SEMNOP: <QSUSB>  ::=   //: <ID>
259  SETQX:  <EXPR>   :::=  ( <EXPR> <SSRHS> )
221  SEXP:   <REXPR>  ::=   <EXPR> }
288  SSASS:  <SS>     ::=   <EXPR> <SSRHS>
334  SSCALP: <SS>     ::=   <ID> <ARGS> ;
334  SSICAL: <SS>     ::=   <ID> ;
334  SSQ:    <SS>     ::=   DO <SS> <QUANT>
334  SSQ:    <SS>     ::=   <QUANT> DO <SS> END
288  SSRHS1: <SSRHS>  ::=   := <EXPR> ;
288  SSRHS2: <SSRHS>  ::=   ::= <BOP> <EXPR> ;
288  SSRHS3: <SSRHS>  ::=   ::= <EXPR> <BOP> ;
290  SSRHS4: <SSRHS>  ::=   ::= <KUOP>
290  SSRHS4: <SSRHS>  ::=   ::= <KUOP>
336  SSSS:   <SS>     ::=   <SS> <SS>
221  STRGEN: <EXPR>   ::=   { <REXPR>
223  STRSEL: <EXPR>   ::=   <EXPR> [ <EXPR> ]
388  SUSB1:  <SUSB>   ::=   //
388  SUSB2:  <SUSB>   ::=   <SUSB> ( <CTYPE> )
388  SUSB3:  <SUSB>   ::=   <SUSB> [ <ASN> ]
388  SUSB4:  <SUSB>   ::=   <SUSB> { <ASN> }
388  SUSF1:  <EXPR>   :::=  <SUSB> <EXPR> \\
388  SUSF1S: <EXPR>   :::=  <SUSB> <SS> \\

235  TYPDIS: <EXPR>   ::=   <ID> :: <EXPR>
244  TYPDIS: <EXPR>   ::=   <ID> :: <EXPR>

285  UBITSW: <UOP>    ::=   BITSWAP
285  UDECOD: <UOP>    ::=   DECODE
285  UENCOD: <UOP>    ::=   ENCODE
285  ULFTZO: <UOP>    ::=   LEFTZEROS
285  UNORM:  <UOP>    ::=   NORM
286  UOPBID: <RHUOP>  ::=   \ <ID>
284  UOPMIN: <UOP>    ::=   -
285  UTALLY: <UOP>    ::=   TALLY
285  UUNARY: <UOP>    ::=   UNARY
```

Deterministic Rules

```
Page  Name

262   DCOUGH: <BEXPR>  :::=  BEGIN <DECL> <EXPR> END
262   DCOUGH: <BEXPR>  :::=  BEGIN <EXPR> <DECL> END

328   BIF2:   <BIF>    :::=  <BIF> <EXPR> THEN <SS> EF

333   DCOUGH: <BSS>    :::=  BEGIN <DECL> <SS> END
333   DCOUGH: <BSS>    :::=  BEGIN <SS> <DECL> END

203           <DECL>   :::=  DEFINE <ID> : <SS> ENDDEFN
203           <DECL>   :::=  DEFINE <ID> = <TYPE> : <EXPR> ENDDEFN
203           <DECL>   :::=  DEFINE <ID> ( <CTYPE> ) : <SS> ENDDEFN
203           <DECL>   :::=  DEFINE <ID> ( <CTYPE> ) = <TYPE> : <EXPR>
ENDDEFN
206           <DECL>   :::=  DEFINE <ID> : MACRO-10( <QS> )
206           <DECL>   :::=  DEFINE <ID> = <TYPE> : MACRO-10( <QS> )
206           <DECL>   :::=  DEFINE <ID> ( <CTYPE> ) : MACRO-10( <QS> )
206           <DECL>   :::=  DEFINE <ID> ( <CTYPE> ) = <TYPE> : MACRO-10(
<QS> )
206           <DECL>   :::=  LET <ID> BECOME <ID> BY <EXPR> ;
207           <DECL>   :::=  LET <ID> BECOME <ID> BY MACRO-10( <QS> )

213   EBIF:   <EXPR>   :::=  <BIFE> <EXPR> THEN <EXPR> ELSE <EXPR> FI
233   ECASEE: <EXPR>   :::=  CASE <EXPR> OF <EXPRV>
239   ECASE:  <EXPR>   :::=  CASE <ID> OF <EXPRV>
269   SETQX:  <EXPR>   :::=  ( <EXPR> <SSRHS> )
265   HOLDIT: <EXPR>   :::=  HOLDING <ASN> GIVE <EXPR> ENDHOLD
388   SUSF1:  <EXPR>   :::=  <SUSB> <EXPR> \\
388   SUSF1S: <EXPR>   :::=  <SUSB> <SS> \\

233   EVCASE: <EXPRV>  :::=  <ID> : <EXPR> ENDCASE
233   EVCASB: <EXPRV>  :::=  <ID> : <EXPR> <EXPRV>
239   EVCASE: <EXPRV>  :::=  <ID> : <EXPR> ENDCASE
239   EVCASB: <EXPRV>  :::=  <ID> : <EXPR> <EXPRV>

328   EBIF:   <SS>     :::=  <BIF> <EXPR> THEN <SS> ELSE <SS> FI
328   SBIF:   <SS>     :::=  <BIF> <EXPR> THEN <SS> FI
329   ECASEE: <SS>     :::=  CASE <EXPR> OF <SSV>
330   ECASE:  <SS>     :::=  CASE <ID> OF <SSV>
331   HOLDIT: <SS>     :::=  HOLDING <ASN> DO <SS> ENDHOLD
333   EDECL:  <SS>     :::=  <BSS>

329   EVCASE: <SSV>    :::=  <ID> : <SS> ENDCASE
329   EVCASB: <SSV>    :::=  <ID> : <SS> <SSV>

196           <TDECL>  :::=  TYPE <ID> = <TYPE> ;
196           <TDECL>  :::=  <TDECL> <ID> = <TYPE> ;

199           <VDECL>  :::=  VAR <IDLIST> = <TYPE> ;
199           <VDECL>  :::=  <VDECL> <IDLIST> = <TYPE> ;
```