

PRELIMINARY

RED

PRELIMINARY DESIGN PHASE REPORT

**CDRL Item 0002AB
Contract MDA903-77-C-0330**

TABLE OF CONTENTS

	page
1.0 INTRODUCTION	1-1
1.1 Overview of Document	1-1
1.2 Relationship to Other Languages	1-2
2.0 COMPLIANCE WITH IRONMAN	2-1
3.0 SUMMARY OF KEY DESIGN DECISIONS	3-1
3.1 Program Structure and Scopes	3-1
3.2 Data Types	3-3
3.3 Capsules	3-7
3.4 Side Effects in Functions	3-11
3.5 Input/Output	3-13
3.6 Exception Handling	3-14
3.7 Multipath Facilities	3-15
3.8 Compile-Time Facilities	3-17
4.0 ANALYSIS OF IMPLEMENTATION FEASIBILITY	4-1
4.1 Formal Language Definition	4-2
4.2 Documentation Requirements	4-15
4.3 REDL Implementation	4-20
4.4 The REDL Language Support Facilities	4-40
5.0 REDL COMPLIANCE WITH DOD CRITERIA	5-1
5.1 Military Applications	5-1
5.2 Readable, Modifiable, Maintainable Programs	5-3
5.3 Rigid Standards	5-5
5.4 Efficient Execution	5-6
5.5 Simplicity	5-9
5.6 Reliable, Provable Programs	5-10
5.7 Ease of Teaching and Learning	5-11

	page
5.8 Machine Independence	5-12
5.9 Practical, Implementable	5-13
5.10 Addresses the Difficult Issues	5-14
 APPENDIX A: RECURSION AND DISPLAYS	 A-1
A.1 Background	A-1
A.2 Avoidance of Display in the Non-Recursive Case	A-1
A.3 Accounting for Recursion	A-4
A.4 Summary of Recommendations	A-6
 BIBLIOGRAPHY	 B-1

LIST OF FIGURES

FIGURE 4-1: Major Modules of REDL Interpreter	4-31
FIGURE 4-2: Notation	4-32
FIGURE 4-3: LEXER	4-33
FIGURE 4-4: Terminal Numbers	4-34
FIGURE 4-5: PARSER	4-35
FIGURE 4-6: BUILDER	4-36
FIGURE 4-7: PREPASS	4-37
FIGURE 4-8: NAMEANAL	4-38
FIGURE 4-9: RUN	4-39

1.0 INTRODUCTION

1.1 Overview of Document.

The purpose of this document is to demonstrate how REDL succeeds in realizing the DoD's criteria for a common language. Chapter 2 provides a description of REDL's compliance with Ironman. In those cases where a part of a requirement is not met, a supporting explanation is given. Chapter 3 focuses upon certain key decisions in the design process and gives the rationale behind them. Chapter 4 analyzes the feasibility of implementing REDL, concentrating on formal semantic specification, documentation requirements, test translator design, and language support facilities. Chapter 5 summarizes the compliance of REDL with Dod's Analyses Plan. Appendix A provides some details relevant to a design decision on recursion and displays.

1.2 Relationship to Other Languages

The design of REDL was influenced by a variety of modern High Order Languages. In this section we summarize some of the more direct influences of other HOLS.

1.2.1 Pascal [JW76].

The chapters of the Informal Language Specification contain a fairly detailed comparison between REDL and Pascal. In short, REDL borrowed from Pascal the general syntactic structure, the kinds of statements and types, and the application-level input/output features.

1.2.2 Modula [Wi76]

The capsule and monitor facility of REDL are derived from Modula ("module" and "interface module" are the terms used there).

1.2.3 Euclid [LHLMP77].

REDL's approach to namespaces, side effects in functions, and aliasing is based upon Euclid.

1.2.4 LIS [IHRC74].

REDL's treatment of pointers is based upon LIS.

1.2.5 SPL/I [Kos76].

The REDL rules for type naming are modeled after analogous conventions in SPL/I.

1.2.6 Other HOLS.

The separate compilation facilities in REDL are an extension of similar features in HAL/S [Int74] and CS-4 [Int75]. The machine-dependent aspects of REDL are based on CS-4. JOVIAL [AF76] inspired the use of directives in REDL. REDL's "overload" facility and some of its treatment of arithmetic types, exception handling, and compile-time facilities are based on PL/I [X75].

2.0 COMPLIANCE WITH IRONMAN

This chapter describes the compliance of REDL with Ironman, organized on a requirement-by-requirement basis. For each requirement, a "degree of compliance" is given: "T" represents total, "P" represents partial, and "PT" represents practically total compliance. Also, section references for the Informal Language Specification (ILS) are provided. In many cases, no supporting explanation is given for "T" compliance; this implies that the discussion in the ILS is sufficient to demonstrate that the requirement has been met.

The Ironman requirements 1A through 1F (General Design Criteria) are not considered in this chapter, since they will be dealt with in Chapter 5. Similarly, requirements 13A through 13G are not treated, since they will be subsumed under the discussion in Chapter 4.

2A. Character Set: T

ILS References: 3.1

As pointed out in ILS 3.1, no language construct requires the use of any characters outside of the 64 character subset of ASCII (in fact, the symbols " and \ are unused).

There was a design decision in connection with 2A as to whether the language should allow characters outside the 64 character ASCII subset to be used at all in programs. Contexts such as comments and string literals are obvious places where characters outside the set might be permitted. Also, lower case letters might be allowed in identifiers. The trade-off is between portability and readability: clearly, any program which contains implementation-specific characters, even in comments or string literals, will not necessarily be directly transportable to another implementation. On the other hand, the prohibition of such characters has an adverse effect on both writability and readability (e.g., there is no carriage return in 64-character ASCII). In REDL we decided to allow the use of implementation-specific characters (the ILS chooses full ASCII to illustrate which characters are legal in which contexts). Although portability thereby suffers, this is somewhat alleviated by the fact that conversions between character sets may be written in the language; hence a source program transliterator, written in REDL, can be provided as a tool in an implementation.

2B. Grammar: T

ILS References: 3.0, Appendix C

REDL programs can be parsed deterministically using bottom-up syntax analysis with one-symbol look-ahead. A LALR(1) grammar [La70] appears in ILS Appendix D. The lexical structure is simple, and the lexical diagrams in ILS Chapter 3 are directly transformable into tables for finite-state machine implementation.

2C. Syntactic Extensions: T

ILS References: 7.4, 8.0

No syntax extension is possible in REDL.

2D. Other Syntactic Issues: T

ILS References: 8.0, 9.0

2E. Mnemonic Identifiers: T

ILS References: 3.3.1.1

Underscore is the break character for identifiers.

2F. Reserved Words: P

ILS References: 3.3.1.2, 3.3.2, 4.2

REDL deviates somewhat from this requirement; in particular, the scope rules have the effect of reserving the identifiers of built-in types and routines, contrary to Ironman. In fact, in REDL, programmers have the opportunity of making their own identifiers reserved (i.e., non-overridable by inner declaration); viz., by making them pervasive. Why did we choose to depart from ALGOL 60 and Pascal namescope conventions? The tradeoff was basically between convenience of writing and both reliability and maintainability. Although the ability to override an outer declaration of a name (either language-supplied or programmer-defined) implies that the user not have to remember all the names known in the scope being composed, the price paid is the possibility of subtle errors and difficulties in maintenance. For example, if a program contains the scope

BEGIN

VAR I:...;

:

:

I

END;

and the programmer introduces a new inner scope overriding the outer declaration of I:

```
BEGIN      <*OLD SCOPE*>
VAR I:....; <*OLD DECLARATION*>

:
BEGIN      <*NEW SCOPE*>
VAR I:....; <*NEW DECLARATION*>
:
...I....    <*OLD REFERENCE*>
:
END;
:
END;
```

then the reference to I which was supposed to be bound to the original declaration is instead bound to the new inner declaration. If the types in the two declarations are the same (a strong likelihood, since the variables' names are the same), then the error will not be caught by the compiler. A related problem occurs in connection with readability and maintainability, especially for large systems.

2G. Numeric Literals: T

ILS References: 3.3.4.1, 5.4.3.2, 5.4.4.2

2H. String Literals: T

ILS References: 3.3.4.2, 5.6.5.2

The provision for special enumeration symbols such as "CR" and "LF" implies that the prohibition against string literals crossing line boundaries does not prevent these literals from containing line termination characters.

2I. Comments: T

ILS References: 3.4

3A. Strong Typing: T

ILS References: 5.1

3B. Implicit Type Conversions: T

ILS References: 5.4.3, 5.4.4, 5.7, 5.8, 5.9

There is a single FIXED type, independent of scale and range. There is a single FLOAT type, independent of precision and range. The declaration of range identifiers helps achieve the goal of having range not distinguish type. Representational differences due to differences in grouping do not distinguish type. A variety of explicit conversion routines are provided (e.g., FIXED to and from FLOAT, FIXED to and from enumeration types, STRING to and from some of the built-in types, type "lifting" and "lowering" functions); there are no implicit conversions.

3C. Type Definitions: PT

ILS References: 5.5., 5.6, 5.9, 6.0

The minor deviation from 3C is explained in connection with 5A.

We note that the last sentence in this requirement ("No restriction shall be imposed on defined types unless it is imposed on all types") influenced our decision not to treat EXCEPTIONs (11.0) and EVENTS (12.4) as data types. In particular, the requirement that each data type permit assignment between its objects (5F) would have been sacrificed, since assignment is not desirable for either EXCEPTIONs or EVENTS. Moreover, the prohibition of assignment for a particular type would not have been so simple, since initialization and CONST and RESULT parameter passing rely on assignment. Also, the issue would have arisen as to whether a non-assignable type could be permitted as a component of an aggregate, and, if so, whether assignment, initialization, and CONST and RESULT parameter passing were permitted for the aggregate. To avoid ad hoc and complex restrictions, we chose not to treat EXCEPTIONs and EVENTS as

data types, but instead to use special forms for their declarations.

3-1A. Numeric Values: T

ILS References: 5.4.3, 5.4.4

We note that there is a single fixed point type; integers are a special case (scale=1).

3-1B. Numeric Operations: T

ILS References: 5.4.3.3, 5.4.4.3

We note that an exponentiation operator is built-in.

3-1C. Numeric Variables: T

ILS References: 5.4.3.1, 5.4.4.1, 5.7

3-1D. Floating Point Precision: T

ILS References: 5.4.4

We acknowledge that the treatment of floating point precision is based on [FW77 Tables 2 and 3].

3-1E. Floating Point Implementation: T

ILS References: 5.4.4.1

3-1F. Integer and Fixed Point Numbers: T

ILS References: 5.4.3

We acknowledge that the treatment of integer and fixed point numbers is based on [FW77, Section III].

3-1G. Fixed Point Scale: T

ILS References: 5.4.3.1

3-1H. Integer and Fixed Point Operations: T

ILS References: 5.4.3.3, 5.4.3.4

3-2A. Enumeration Type Definitions: PT

ILS References: 5.5.1

The only facet of 3-2A not met by REDL is the requirement that subsequences be permitted for both ordered and unordered enumeration types. REDL allows subsequences only for ordered enumeration types. The reason for this decision is that the notion of a subsequence carries with it a natural ordering. It is potentially confusing for the programmer to have to remember that a subsequence X..Y of an unordered enumeration type includes those elements from X through Y (where "through" involves the lexical ordering of elements in the declaration of the type) but that these elements (chosen on the basis of an implicit ordering) are then to be treated as though unordered.

3-2B. Ordered Enumeration Types: T

ILS References: 5.5.1

3-2C. Boolean Type: T

ILS References: 5.4.1

The BOOLEAN type may be regarded as an unordered enumeration type comprising the identifiers FALSE and TRUE, with a set of built-in operations provided.

3-2D. Character Types: T

ILS References: 5.4.2, 5.5.1, Appendix B

3-3A. Composite Type Definitions: T

ILS References: 5.5.2, 5.5.3, 5.5.4, 5.5.5

ARRAY, RECORD, UNION, and POINTER type generators are provided.

3-3B. Component Specifications: T

ILS References: 5.5.2, 5.5.3, 5.5.4, 5.5.5

3-3C. Operations on Composite Types: T

ILS References: 5.5.2.3, 5.5.2.5, 5.5.3.2, 5.5.3.3, 5.5.4.2, 5.5.4.3, 5.5.5.3, 5.5.5.4

3-3D. Array Specifications: PT

ILS References: 5.5.1.1, 5.5.2, 5.7

The only aspect of 3-3D not met by REDL is the requirement that subscript ranges be permitted to be "any contiguous sequence from an enumeration type." As in 3-2A, contiguous sequences are allowed for ordered enumeration types only.

3-3E. Operations on Subarrays: T

ILS References: 5.5.2.4

3-3F. Nonassignable Record Components: P

ILS References: 5.5.3, 6.0

Although the RECORD type generator does not directly allow the definition of nonassignable record components, this effect may be achieved via a CAPSULE declaration. For example, if R is to be a record type with assignable component A and nonassignable constant component B (of types TA and TB, respectively), then the user may declare the following capsule:

```
CAPSULE C;
  !IMPORT TA, TB;
  !EXPORT R, FETCH_A, FETCH_B, STORE_B, CONSTRUCT_R;

  TYPE R: RECORD
    A: TA;
    B: TB;
  END RECORD,

  FUNCTION FETCH_A (CONST r:R) RESULT a:TA INIT r.A;
    !INLINE;
  END FUNCTION FETCH_A;
```

```

FUNCTION FETCH_B (CONST r:R) RESULT b:TB INIT r.B;
  !INLINE;
END FUNCTION FETCH_B;

PROCEDURE STORE_B (VAR r:R, CONST b:TB);
  !INLINE:
  r.B:=b;
END PROCEDURE STORE_B;

FUNCTION CONSTRUCT_R (CONST a:TA, CONST b:TB)
  RESULT r:R INIT R(A:a, B:b);
  !INLINE;
END FUNCTION CONSTRUCT_R;

END CAPSULE C;

```

The technique for obtaining a nonassignable component whose value is the dynamic value of an expression is analogous (there is no need for an actual component in the record; instead, the "fetch" function simply returns the result of the expression).

3-3G. Variant Types: PT

ILS References: 5.5.4

REDL provides the UNION type generator to obtain the effect of a "discriminated union" required by 3-3G. The only point of difference is that the value of a variant may not be used "anywhere a value of the variant type is permitted," since this would imply either an implicit conversion or a defeat of the language's strong type checking. To obtain the behavior specified in 3-3G, the user must construct a union value from the value of the variant. As an example, if type U is declared as follows:

```

TYPE U: UNION
  A: BOOLEAN;
  B: CHAR;
END UNION;

```

and routine R takes a parameter of type U, then it is not permissible to pass a BOOLEAN (or a CHAR) as an actual parameter. If X is a BOOLEAN which is to be passed, then the actual parameter will be written U(A: X), denoting an object of union type U which has an A alternative whose value is X.

3-3H. Tag Fields: T

ILS References: 5.5.4, 9.5.2

We point out that the select statement combines safety with efficiency in tag field discrimination.

3-3I. Definitions of Dynamic Types: T

ILS References: 5.1.4, 5.5.5

REDL's dynamic storage class and POINTER type generator provide for the definition of dynamic types.

3-3J. Constructor Operations: T

ILS References: 5.5.5.4

The procedure NEW is a constructor for objects of dynamic types. REDL provides an explicit FREE routine; this is not in conflict with the second sentence of 3-3J ("such elements shall remain allocated as long as there is an access path to them"), since an exception is raised when an attempt is made to FREE a dynamic object which is references on some other access path.

3-4A. Bit Strings: T

ILS References: 5.4.1, 5.5.1, 5.5.2, 5.6

The following example shows how the user might define a type for a BOOLEAN vector indexed by a CHAR in the range ^0^ through ^9^:

```
TYPE BOOL-VECT: ARRAY (^0^..^9^) OF GROUPED BOOLEAN;
```

3-4B. Bit String Operations: PT

ILS References: 5.6.6

The built-in type BITS comes equipped with the operations specified in 3-4B. These operations are not "automatically" provided for user-defined BOOLEAN vector types: the user must overload these operators if they are to be invocable on operands of such types.

3-5A. Encapsulated Definitions: T

ILS References: 6.0

The CAPSULE in REDL is the unit of encapsulation.

3-5B. Effect of Encapsulation: T

ILS References: 6.0

3-5C. Own Variables: T

ILS References: 6.3

3-5D. Operations between Types: P

ILS References: 6.0

Since representational details are not exportable from CAPSULEs, it will be necessary to define the set of types among which conversions are required in a single CAPSULE, instead of having each type defined in its own CAPSULE and the set of CAPSULEs further included in one encompassing CAPSULE.

4A. Form of Expressions: T

ILS References: 8.3

4B. Type of Expressions: PT

ILS References: 7.3, 7.4

A minor deviation from 4B is that REDL does not provide a means for specifying the type of an expression explicitly. (We note that the type of any expression is always determined by the types of the operands, and that there might be a compiler option which provides for a specified segment of the program, an annotation of the source listing with expression types explicit.)

4C. Side Effects: T

ILS References: 7.2.1, 7.3.2, 7.4

4D. Allowed Usage: T

ILS References: Syntax Diagrams

4E. Constant Valued Expressions: PT

ILS References: Syntax Diagrams and 14.2.3

The only deviation from 4E is that some of the directives in REDL require STRING literals (as opposed to permitting arbitrary compile-time evaluable STRING expressions). This was done primarily for clarity of exposition.

4F Operator Precedence Levels: PT

ILS References: 8.3

The seven precedence levels exceed the number suggested in 4F. Although a smaller number could have been achieved by using the Pascal convention of merging the BOOLEAN operators into the arithmetic hierarchy (i.e., placing AND and & with the multiplicative operators and OR, XOR, and ! with the additive operators), we chose to use the Algol 60 approach and to place the BOOLEAN operators at lower precedence. In making this choice, we adopted the advice of N. Wirth, the designer of Pascal. In a paper which assessed the Pascal Language, Wirth made the following statement [Wi75, p. 194]:

In the interest of simplicity and efficient translatability, Pascal aimed at a reasonably small number of operator precedence levels. Algol 60's hierarchy of 9 levels seemed clearly too baroque...In retrospect, however, the decision to break with a widely used tradition seems ill-advised, particularly with the growing significance of complicated Boolean expressions in connection with program verification.

4G. Effect of Parentheses: P

ILS References: 8.3

The first part of 4G ("explicit parentheses shall dictate the association of operands with operators") is met. The second part of 4G appears to contain an error. As it is stated, "explicit parentheses shall be required to resolve the operator-operand

associations whenever an expression has a nonassociative operator to the left of an operator of the same precedence...." The intention of this requirement is to make illegal such expressions as $A-B+C$ and $A-B*C+D$; the programmer would have to add parentheses to specify the intended right operand of the minus operation. However, as stated this requirement also makes illegal the expression $A/B + C/D$ (the nonassociative operator / is to the left of another occurrence of /), which should not require parentheses. The operator-operand associations are clear as is. Rather than attempt to place added restrictions on the rule stated in 4G so that $A-B*C+D$ would be illegal and $A/B + C/D$ legal, we chose a simple and widely-used left-to-right associativity convention. Under our rule, both expressions in the preceding sentence are legal, and $A-B*C+D$ is parsed in accordance with the conventions of most High Order Languages; viz., $(A - (B*C)) + D$.

An effect of the left-to-right associativity rule is that the last part of 4G ("explicit parentheses shall be required... wherever consecutive operators of an expression are of the same precedence but have different operand types") is not met. Again, the reason was that of language simplicity.

5A. Declarations of Constants: PT

ILS References: 5.1.4, 5.2, 14.2.1

The only deviation of REDL from 5A is that the types of compile-time constants are restricted to the built-in types. (This also implies a deviation from 3C.)

5B. Declarations of Variables: T

ILS References: 5.1.4, 5.2

5C. Scope of Declarations: T

ILS References: 4.2

The IMPORT directive, in conjunction with open and closed scopes and the Pervasive directive, satisfy 5C.

5D. Restrictions on Values: T

ILS References: 7.1.4, 7.3.1, 8.6, 9.3

5E. Initial Values: T

ILS References: 5.2.2

We note that the "?" form provided for initialization phrases is not a default initial value but rather an explicit indication that no initialization is desired.

5F. Operations on Variables: T

ILS References: 5.1.5.7

6A. Basic Control Facility: T

ILS References: 9.0

6B. Sequential Control: T

ILS References: 4.1.4, 4.2, 9.0

We note that in REDL the semicolon is used as a statement terminator (i.e., it is syntactically part of each statement).

6C. Conditional Control: PT

ILS References: 9.5

REDL deviates in a minor way from the last sentence in 6C ("only the selected branch shall be compiled when the selected case for a conditional statement is determinable at translation time"). The stated behavior is true for compile-time conditionals (14.3), but not necessarily for general conditional statements.

6D. Short Circuit Evaluation: T

ILS References: 5.4.1.3

We note that the short circuit forms are not restricted to the contexts specified in 6D.

6E. Iterative Control: PT

ILS References: 9.6

The only deviation is that in REDL one may not perform an iteration over a subrange of an unordered enumeration type (see also 3-2A above).

6F. Control Variables: T

ILS References: 4.2.4

Each structured control statement is a scope and thereby permits the declaration of local variables.

6G. Explicit Control Transfer: T

ILS References: 4.2.4, 9.7.3

7A. Function and Procedure Definitions: T

ILS References: 7.0, 8.6.3, 9.7.1

7B. Recursion and Nesting: T

ILS References: 7.1.8

Appendix A to this document shows why the restriction in ILS 7.1.8 (that a recursive routine may not contain the declaration of another recursive routine) is sufficient to enable an implementation to avoid the use of a display for recursion. As pointed out by the response to Ironman Clarification Question 66, however, displays may be necessary in the presence of parallel paths.

7C. Scope Rules: T

ILS References: 4.2.1, 11.2

We note that exception identifiers in REDL follow the same lexical scope rules as any other identifier.

7D. Function Declarations: PT

ILS References: 7.3.1, 7.4.3

A minor deviation from 7D is that the specification of associativity on an overloaded operator is optional.

7E. Restrictions on Functions: PT

ILS References: 7.2.1, 7.3.2

The only deviation from 7E is that REDL is more strict with respect to side effects in functions. The special case of assignment to a variable "own to an encapsulation that does not contain any call on the function" is not permitted. The result is a simplification of the rules; the expense is that some routines must be written as procedures instead of functions.

7F. Formal Parameter Classes: T

ILS References: 7.1.4

7G. Parameter Specifications: T

ILS References: 5.1.2, 5.6.4, 7.1.4

7H. Formal Array Parameters: T

ILS References: 5.6

We point out that 7H (in particular, the requirement that "determination of the subscript range for formal array parameters may be delayed until execution time and may vary from call to call") motivated much of the design of the parameterized type facility

in REDL.

7I. Restrictions to Prevent Aliasing: T

ILS References: 7.2.2

8A. Low-Level Input-Output Operations: P

ILS References: 10.1, 10.5, 13.0, 14.5

REDL does not provide a set of built-in low-level I/O operations; the wide range of device characteristics stymied our attempts to define a reasonably small set of I/O primitives. Instead, REDL contains a sufficiently powerful set of machine-dependent features and definitional facilities so that implementations can provide both low-level and application-level I/O as library routines. An example of these facilities is the FILE_COMPATIBLE function (14.5.3). Programming examples appear in ILS, Appendix A.

8B. Application-Level Input-Output Operations: PT

ILS References: 10.2, 10.3, 10.4

REDL defines the protocols for invoking standard library routines for application-level I/O. A minor deviation from 8B is that the number and types of actual parameters to the OPEN procedure (10.3.1) is dependent upon target machine characteristics (i.e., there is not simply a single standard library definition for this routine).

8C. Input Restrictions: T

ILS References: 10.2, 10.3

8D. Operating System Independence: PT

ILS References: 12.0

Since some of the Ironman requirements (especially in connection with parallel processing) imply the need for run-time support generally provided by an operating system, REDL falls

short of the operating system independence specified in 8D. However, since the routines required for run-time support can be written in REDL, the presence of an actual operating system is unnecessary.

A minor deviation from 8D is the form of the OPEN procedure, as described above in 8B.

8E. Configuration Control: P

ILS References: 10.1, 10.5, 13.0, 14.5

As with 8A, REDL provides facilities with which such low-level operations can be defined, instead of selecting a set to be built into the language.

9A. Parallel Control Structure: T

ILS References: 12.2

A scope-entry determined number of parallel control paths which rejoin at a single point can be achieved by embedding a FORK statement in a recursive routine.

9B. Parallel Path Implementation: T

ILS References: 12.2

9C. Mutual Exclusion and Synchronization: T

ILS References: 12.3, 12.4

The EVENT is the low-level synchronization primitive. Although EVENTS may also be used as a mutual exclusion primitive, REDL provides the MONITOR CAPSULE as a more secure facility for this purpose.

9D. Scheduling: T

ILS References: 12.2.4

9E. Real Time Clock: T

ILS References: 12.5

9F. Simulated Time Clock: T

ILS References: 12.2

REDL does not provide a special facility for the simulated time clock. Instead, the clock may be realized within the language by a shared integer data object declared in a MONITOR CAPSULE and incremented by one path of a FORK statement and read by the other (simulation) paths. The routine which increments the clock can maintain a vector of integers, one for each simulation path of the fork, reflecting the value of the simulated clock when the path is to be "awakened." An entry is inserted in the vector when the path calls a delay routine defined in the MONITOR CAPSULE. This delay routine causes the calling path to wait for an EVENT specific to the path. This EVENT will be sent by the clock incrementing routine when the simulated clock has reached the value posted for the path in the vector. When the clock incrementing routine detects that all the paths are waiting for a future simulated time, the clock may be advanced by awakening all the paths (i.e., by sending all the EVENTS).

9G. Catastrophic Failures: T

ILS References: 11.3, 12.2.2

A path may raise the X_TERMINATE exception in another path to obtain the effect of 9G.

10A. Exception Handling Facility: T

ILS References: 11.0

10B. Error Situations: T

ILS References: 5.1.2, 5.2.2, 5.5.2.3, 5.4.3.3, 5.5.4.2, 7.2.2, 9.5.2.2, 9.8, 11.1, 11.3, 12.2.2

REDL provides the X_SUBSCRIPT, X_RANGE, X_OVERFLOW, X_INIT, X_ALIAS, X_TAG, X_TERMINATE, and X_ASSERT exceptions for the purposes stated in 10B.

10C. Enabling Exceptions: T

ILS References: 11.3

RAISE statements are used for enabling exceptions, and ON statements are used for specifying handlers.

10D. Processing Exceptions: T

ILS References: 11.3.2, 11.5

The ON statement, X_ANY, and LAST_EXCEPTION supply the needed behavior. We note that REDL allows the user to specify system-defined handlers for exceptions.

10E. Order of Exceptions: T

ILS References: 8.4

10F. Assertions: T

ILS References: 9.8

We note that assertions cannot have side effects in REDL.

10G. Suppressing Exceptions: T

ILS References: 11.4

The OFF directive is used for suppressing exceptions within a scope.

11A. Data Representation: T

ILS References: 13.3.1

The machine-record type generator supplies the needed behavior.

11B. Multiple Representations: P

ILS References: 5.1.2, 5.6.4, 5.7, 5.8

REDL is fairly conservative in its approach to multiple representations, in the interest of simplicity and efficiency. The grouping attribute is supplied for use on components of aggregate types, and different objects of the same type may differ in grouping. This does not imply that they occupy different amounts of storage, rather that the ungrouped object is alone in the storage unit in which it resides and that the grouped object may be in the same storage unit as other grouped objects. If the user intends to define two arrays, one containing grouped components and the other containing ungrouped components, these arrays must be of different types, and explicit packing and unpacking routines must be defined to convert from one representation to the other. (If it were possible to define these arrays as having the same type, then assignments would mask the packing and/or unpacking code. The expense of such code was one of the factors influencing our decision to prevent such arrays from having the same type and instead to make the user explicitly aware that packing and/or unpacking is being performed.)

We note that range identifiers give the user the opportunity to associate identifiers with representations.

REDL completely meets the requirement that representations be identical for VAR (i.e., input-output) parameter passing.

11C. Translation Time Constants and Functions: P

ILS References: 13.2, 14.2.2

The main deviation from 11C is that in REDL a configuration specification is mandatory only when a program is machine-dependent. This specification is embodied in REDL's CONFIGURATION directive. An advantage to the REDL approach is that machine-dependent programs

are immediately visible as such because of the presence of the directive.

11D. Configuration Dependent Specifications: P

ILS References: 13.0

REDL provides a comprehensive but encapsulated set of machine-dependent features. The deviation from 11D is that the use of these features is permitted on the basis of the presence of a CONFIGURATION directive, and not on the basis of a conditional construct which discriminates on object machine characteristics. However, through the use of compile-time constant declarations and compile-time conditional commands, much of the required behavior could be achieved.

11E. Code Insertions: T

ILS References: 13.4

11F. Optimization Specifications: T

ILS References: 4.2.9

12A. Library Entries: PT

UKS References: 14.1

A minor deviation from 12A is that a machine configuration specification is not a separable library entry. That is, REDL allows only declared program elements as its units of access and promotion in separately compiled programs, and configuration directives do not fall into this class.

12B. Separately Compiled Segments: T

ILS References: 14.1

The PROMOTE and ACCESS directives fully support 12B.

12C. Restrictions on Separate Compilation: T

ILS References: 14.1

12D. Generic Definitions: T

ILS References: 14.4

REDL's compile-time procedures provide the behavior required by 12D.

3.0 SUMMARY OF KEY DESIGN DECISIONS

This chapter summarizes some of the major design decisions which shaped the structure of REDL. The organization of this chapter basically follows that of the ILS.

3.1 Program Structure and Scopes.

The design of namespaces in REDL was influenced heavily by Euclid [LHLM77]. A possibly controversial decision was to prohibit an outer declaration of a name from being overridden in a scope S if that declaration would otherwise be known in S. Some reasons for this decision appear in Chapter 2 in the discussion of Ironman 2F.

The REDL has two kinds of scopes - open and closed - because there is a natural difference between the rules for inheritance of outer declarations in the two cases. Because of the desired restriction on side effects and aliasing, declarations for variables should not be automatically inherited by routines. On the other hand, BEGIN statements, FOR statements and the like are scopes, and in these cases the programmer should not have to write long import lists to gain access to outer declarations. Thus, routines are closed scopes (names, unless pervasive, must be explicitly imported), whereas BEGIN and FOR statements are open scopes (no import is necessary).

The PERVERSIVE directive was included in REDL to allow types, constants and routines to be referenced in closed scopes without being explicitly imported. The decision to prohibit variables from being made pervasive was based on consideration of the rules for side effects and aliasing. In particular, the checking for dangerous aliasing is considerably complicated if variables may be pervasive.

Programs are prohibited from immediately containing statements, in order to avoid the interaction with separate compilation which would occur in defining the order of execution of statement sequences from separate programs. This is also why capsules are prohibited from immediately containing statements.

3.2 Data Types.

3.2.1 Type Naming.

A basic design decision underlying the type facility was to require that each type defined in a program be named. We were influenced heavily by SPL/I [Kos76] in making this decision. The main advantage of the rule adopted is that type identity is straightforward (for both the compiler and the programmer). This advantage should not be underestimated, since the rules for type identity can be exceedingly complex in languages which have powerful data definition facilities yet which allow unnamed types (e.g., ALGOL 68 [Pe71]). Also, the requirement that types be named implies that compilers can perform optimization more easily. In REDL, the declaration

```
TYPE T: ARRAY(1..10) OF T1;
```

implies that each object of type T has exactly 10 components, and there is no need to associate "dope vector" information with such objects.

3.2.2 Storage Classes.

REDL adopts the static and dynamic storage classes of Pascal. The decision to exclude PL/I-style-static was based on reliability and simplicity considerations. The appearance of PL/I-style-static data in a reentrant routine would lead to unprotected sharing, and special rules would have to be established barring the use of such data in functions. Moreover, there is no requirement in Ironman for PL/I-style-static, and most of the efficiency benefits of this storage class can be attained by declaring program-level data (possibly within a capsule).

3.2.3 Pointers.

REDL's pointer facility is based on ideas in both Pascal and LIS [IHRC74]. The fundamentals are derived from Pascal, and the distinction between pointers whose values are constants and those whose values are variables is from LIS. The collection scheme of Euclid was considered but was rejected on the basis of conceptual complexity. The central presence of a variable-like entity (the collection) in a type definition was judged likely to cause confusion.

3.2.4 Parameterized Types.

The parameterized type facility is basically in response to Ironman 7H; i.e., it allows routines to be applied to arrays which may differ in size. The question of how to extend Pascal so that this objective is met has evoked considerable interest among the programming language community, and REDL's approach is based on the recommendations of N. Wirth [Wi75, p.195].

[The limited applicability of static array types] can be overcome by allowing type definitions - in particular array types - to be parameterized....

A most sensible decision is to restrict parameterization to the index bounds of array types, and to allow for constants only as actual parameters (in variable declarations). This already solves the dilemma of array parameters. If dynamic arrays are to be allowed, the latter rule may be relaxed. I would caution, however, against any further generalization: allowing the component type of an array to be a parameter too, for example, would destroy many advantages of the Pascal type concept at once.

3.2.5 Grouping.

REDL's approach to packed aggregates differs slightly from Pascal's. In Pascal, a structured type may be prefixed with the "packed" specifier, whereas in REDL the "grouped" specifier is placed before the component type. It is important to recognize that the notions of packing and grouping are distinct, and that a language may choose either or both in the interest of giving the user some control over space/time trade-offs for data storage. The relationship between packing and grouping is that the components of a packed aggregate are grouped. It is misleading to say that each component will be packed, since in general a packed object occupies less space than an unpacked one. Instead, the fact that the whole object is packed implies that each storage unit occupied by the object may house several components, i.e., the components are grouped.

The decision of Pascal to use packing instead of grouping resulted in a number of subtle language issues. As an example, is it legitimate to pass a component of a packed object as a variable parameter? To allow this can lead to serious inefficiencies, but there is no simple way to explain how the Pascal language semantics would prohibit it.

In the design of REDL it became clear that a component of a packed object has a different attribute than a component of an unpacked object, even though both components might have the same type. Thus we decided to make the grouping attribute explicit by allowing the user to specify GROUPED aggregate components. Since a grouped component has a different value for the grouping attribute than a formal parameter to a routine, the prohibition of passing grouped components as VAL parameters is a direct consequence of REDL's standard rules on parameter passing.

The decision to provide grouping but not packing was made in the interest of simplicity and readability. Although greater flexibility in determining data representation would have resulted had

we allowed both grouping and packing, the complexity of the rules outweighed the benefits. If the programmer could specify both packing and grouping, then redundant or contradictory specifications would be possible, as in TYPE T1: PACKED ARRAY (1..10) OF GROUPED BOOLEAN; <*redundant*> or TYPE T2: PACKED ARRAY (1..10) OF UNGROUPED BOOLEAN; <*contradictory*>. There is also the question of whether specifying packed or a record of records applies at each level or only the first.

3.3 Capsules.

In considering data abstraction facilities, we investigated three specific language approaches: Simula, CLU and Modula. This section describes the facilities in each of these languages and discusses why their approaches were rejected or incorporated into the capsule mechanism of REDL.

Simula [DH72, Pal76] was designed as a general purpose simulation language. Although it predates the interest in data abstraction, it contains a feature, the "class", which makes it possible to construct data abstractions in a relatively simple manner.

A class, like a block, is a collection of declarations and procedures. Unlike a block, a class comes into existence as an object through explicit allocation on the heap by the NEW procedure. This procedure returns a pointer to the class object. By declaring several pointer variables and allocating the same class, the programmer obtains multiple objects of a given data abstraction. The objects are destroyed by automatic garbage collection when they are no longer referencable. Simula classes have a lifetime from the time of allocation to the time when no pointer-variable points to them.

Parameters to the class are used to initialize the newly allocated object. The local variables and procedures within the class are considered to be a Cartesian product which is the representation of the data abstraction. The variables and procedures are the abstract operations which can operate upon those local variables. A later revision of Simula introduced a HIDDEN attribute which could be attached to any declaration in a class to make its declared name inaccessible except to procedures internal to the class [Pal76].

At the time Simula was designed, it was assumed that most operations would be unary operations. That is, a single object is associated with a set of operations. If, for example, a complex number were declared in a class with a set of operations for complex numbers, it would not be possible to add together two complex numbers. The second complex number would be another object of the same class with its own set of operations. This is a severe drawback to the utility of data abstractions in Simula.

The class concept was rejected for several reasons:

(1) Operations are bound to the created object in a class, not to the object's "type". Therefore, operations which need access to the representation of two objects of the same type are impossible.

(2) Semantically, the Simula class is a block with "retention". This is a more complicated approach than the approach taken by REDL.

CLU [SA75] was designed expressly for structured programming. It contains a syntactic construct for data abstractions, a cluster. A cluster encapsulates a representational type and a set of operations. Multiple objects can be declared to be of this type. Only a single type can be exported from a cluster and this is done automatically. The lifetime of an instance of a cluster is tied to the lifetime of the variable declared to be of that cluster. When a variable of a cluster type is allocated, the parameters to the cluster type are used to initialize the local variables of the cluster through initialization code contained within the cluster. There are two unusual formal parameter specifications allowed within a cluster: REP and TYPE. REP allows a procedure to take parameters of the cluster type and act upon them within the procedure as the representational type. This mechanism means that, unlike SIMULA, CLU can have operations with multiple operands of the same type and have access to the representation of all these

operands. Types may be passed as parameters. This means that a stack may be defined as an array of X, where X is a parameter specified to be a type. Clusters, then, can act as type generators.

The CLU approach was rejected because:

- (1) Since only a single type can be exported from a cluster, it is not possible to define operations between two exported types (i.e., conversions).
- (2) Instances in CLU share the "retention" problem with SIMULA, even though the CLU clusters have their lifetime associated with a variable declaration rather than with explicit allocation.
- (3) Unlike Simula, CLU does not allow the straightforward definition of a data abstraction of a single object. It is only possible to define a type and then declare a variable of that type, requiring that an object be passed as a parameter every time, with the additional overhead of parameter passage.

Modula [Wi76] was developed by Wirth for modular multi-programming. It is derived from Pascal and contains a technique for data abstraction which is consistent with the objective of simple language design.

A data abstraction in Modula is known as a module. A module is a boundary around local variables and routines which limits access in and out. A module contains a define list which can include the names of types, variables and procedures which are exported from the module. If a type is exported, there is no way to discover the representational type; only the name is exported. The only operations that may be performed upon a variable of an exported type are the procedures exported from the same module as the type. If a variable is exported, it may not be changed, but is a read-only variable. The lifetime of local variables is the lifetime of the scope immediately containing the module.

A module contains a statement list following the procedure declarations. At entry to the scope immediately enclosing the module, this statement list is executed. The purpose of the statement list is the initialization of the local variables. Modula can be used to define multiple objects with the same data abstraction.

When a type is exported, its structure may not be examined. There is no initialization code possible when the operations are associated with a type rather than an object. Initialization of objects must take place through an exported operation.

Modula can also be used to define a single object. In this case, a local variable is defined and not exported and abstract operations are exported. In this case, the initialization code can be used to initialize the single object.

Operations are accessed outside modules by distinct names. In Modula, it is not possible to define overloaded operations.

The scheme chosen in REDL is basically that of Modula with some minor modifications.

(1) Variables cannot be exported from a capsule in REDL. Exporting a variable means a loss in representational transparency. Any future change to the representation must leave the variable untouched since that can be referenced in the body of the program. A variable which is exported from a monitor poses additional problems. All references to the variable outside the monitor could cause the locking and unlocking of the monitor. Alternatively, references to the variable could take place with no protection and the reference could potentially cause errors. Rather than allow variables to be exported from capsules and not from monitors, REDL has taken the more consistent approach of not allowing variables to be exported from either.

(2) No statement list appears within a capsule. Reasons for this decision are given in Section 3.1.

(3) REDL makes it possible to export overloaded operations.

3.4 Side Effects in Functions.

REDL goes further than Ironman 7E in prohibiting side effects from functions. If a function is declared within an encapsulation, Ironman permits "assignment to a variable that is non-local to the function... where the variable is own to an encapsulation that does not contain any call on the function." This kind of side effect can be considered to be not directly observable from the site of the function call, and therefore harmless. However, even this exception to the prohibition defeats many of the advantages of preventing side-effects. For instance, 7E would allow the creation of a capsule with an own variable X and two exported functions:

(1) ASSIGN_X: a function of one parameter, which assigns the parameter to X and returns the value.

(2) VALUE_X: a function of no parameters which returns the current value of X.

The use of ASSIGN_X and VALUE_X in an expression is equivalent to allowing assignment to appear within an expression. Such a usage is an obvious violation of the intent of 4C, which prohibits the changing within an expression of any variable which is accessible at the point at which the expression occurs.

There are four ways in which a function could cause a side effect (the raising of an exception is not considered a side effect):

- (1) modification of a parameter
- (2) assignment to a non-local variable
- (3) assignment to a dereferenced pointer
- (4) invocation of an outer routine which performs (2), (3), or (4).

The restrictions on side effects handle these four circumstances. Thus, a function may only have CONST parameters, may only import variables READ ONLY, may only use pointer dereferences in R-valued contexts, and may call an outer procedure only if the procedure is "safe". (A safe procedure must be so marked by the presence of a SAFE directive. The compiler will check that a safe procedure only imports variables READ ONLY, only used pointer dereferences in R-valued contexts, and only calls an outer procedure if the procedure is also safe.)

3.5 Input/Output.

REDL's philosophy regarding I/O is to provide a sufficient set of definitional facilities so that both low-level and application-level routines can be written in the language. The machine-dependent features and the compile-time facilities provide this capability, with no sacrifice of efficiency.

The standard library application-level routines are derived primarily from Pascal. Differences include the provision for random-access file organization and the inclusion of "status" result parameters for the I/O procedures.

The decision not to build in asynchronous I/O was based on considerations of consistency with the multipath facility. The fork-region is the basic parallel processing feature, and an asynchronous I/O call (i.e., a call which has the effect of spawning an I/O task and, in parallel, continuing execution at the point following the call) would have been in conflict with the fork's "one-in one-out" control flow philosophy. We point out that asynchronous I/O can be realized in the language through a fork statement, so that the decision not to build it in does not cause a loss of functionality. Alternatively, a CONNECT directive may be used to associate a programmer-supplied procedure with an asynchronous I/O interrupt.

The reason for providing an explicit "status" parameter on I/O calls, instead of having the I/O routines raise an exception, is that "abnormal" status values do not necessarily reflect error situations.

3.6 Exception Handling.

3.6.1 Scope of Exception Identifiers.

Ironman 7C implies that the language can choose dynamic scoping rules for exception identifiers. We rejected this approach and instead decided that exception identifiers follow the same static scoping conventions as any other identifier. The benefits of this decision are consistency (exception identifiers do not need special-case rules) and readability (dynamic scoping rules imply a need for the reader to be familiar with the call chain in order to understand the semantics of a routine which raises an exception).

3.6.2 ON Statements.

The form chosen for the ON statement is consistent with the "one-in one-out" control flow philosophy and also avoids a recursion problem encountered in some exception mechanisms. If an ON statement raises exception X in the course of handling X, then no special-case rules are required in REDL to prevent the same handler from being invoked.

3.7 Multipath Facilities.

3.7.1 Monitors.

A variety of facilities were considered in connection with the requirement for mutually exclusive accesses to shared data. Semaphores [Di68] were rejected because their use for mutual exclusion is highly error-prone. Critical regions [Bri72], also known as update blocks [Int74], were dismissed because the code to access shared data would have to be distributed among all the paths using the data. Moreover, critical regions would have been a new feature in the language. The approach to mutual exclusion adopted in REDL is the monitor capsule (or monitor for short), modeled after interface modules of Modula. Since monitors are a special kind of capsule, the number of new concepts is minimized. Moreover, the encapsulation in the monitor of the code which is used for accessing shared data is useful for program maintainability.

3.7.1 Events.

The EVENT is the language primitive for synchronization. It corresponds to a general (i.e., integer) semaphore and was chosen over the Modula SIGNAL because of potential difficult-to-trace errors inherent in the latter. As pointed out by Wirth [Wi77, p. 579], if a process in Modula sends a SIGNAL before a second process issues a wait, the second process will be indefinitely postponed since there is no trace of the SIGNAL's having been sent. This is not a problem with REDL's EVENTS, since the sending of an EVENT increments a count.

The reason for our choosing integer rather than binary semaphores as our model for EVENTS is that it is somewhat more convenient to define binary semaphores in terms of integer semaphores than vice versa. (This is due to the fact that EVENT is not a data type and hence EVENTS cannot be used as components of aggregates.)

REDL leaves up to the programmer whether the sending of or waiting for an **EVENT** should release a monitor. This is accomplished via an optional **RELEASE** phrase on these two statements.

3.8 Compile-Time Facilities.

3.8.1 Separate Compilation.

REDL's approach to separate compilation is based on HAL [Int 74] and CS-4 [Int 75]. The objective was to provide sufficient flexibility that large programming systems could be designed and developed without artificial constraints on the contents of separately compiled programs. An equally important objective was to retain REDL's strong type-checking across separate compilations. The PROMOTE and ACCESS directives of REDL satisfy these objectives. The example in ILS 14.1.4 illustrates the use of these directives in defining separately compiled programs containing mutually recursive routines.

3.8.2 Compile-Time Procedures.

REDL's compile-time procedures provide a facility for generic parameters as required by Ironman 12D. An alternative approach - the inclusion of a special "generic" binding class for procedures, functions, and capsules - was rejected on two grounds. First, it would require a specialized mechanism that would complicate the semantics of these facilities (capsules, for example, do not take parameters). Second, the similarity of forms for invocation of normal routines and invocation of routines taking generic parameters would be misleading in light of the semantic and efficiency differences between the two. The REDL approach solves the first problem by providing a separate facility - the compile-time procedure - which is semantically independent of routines and capsules. It solves the second problem by requiring that compile-time procedures be instantiated via a distinct form - the compile-time call command.

It may be noted that there is a potential interaction between compile-time procedures and side effects in functions. It might appear that by declaring a function within a compile-time procedure such that a statement formal parameter to the compile-time procedure is referenced within the function, the user could obtain a side effect. However, this is not the case; the restrictions against side effects are enforced when the function declaration is produced by a compile-time invocation of the compile-time procedure.

4.0 ANALYSIS OF IMPLEMENTATION FEASIBILITY

In this chapter we consider four specific aspects of REDL's implementability: formal semantic specification (4.1), documentation requirements (4.2), implementation design (4.3), and language support facilities (4.4).

4.1 Formal Language Definition.

4.1.1 Introduction.

The formal definition of a programming language is valuable in many domains. During the language design process, a formal definition provides a succinct mode of communication for the design team. Difficulties encountered in defining the formal semantics of a language construct often reveal design problems or unexpected feature interactions.

The developers of different implementations for the language employ the definition as a touchstone. It acts as a formal specification that particular translators must satisfy. This enhances the portability of machine-independent programs, since each translator conforms to the single, machine-independent definition. It also contributes to reliability--a program compiled under different translators for the same target machine--will exhibit a single behavior.

The formal semantic definition is the final arbiter of all questions concerning the meaning of programs written in the language. While not being appropriate in itself as a programmer's guide, it serves as the foundation of language reference manuals and primers.

Finally, it serves as the basis set of axioms and inference rules for proving the formal correctness of programs. The axioms for language statements act as predicate transformers in the proof process. Although the application of manual and automatic proving techniques to practical programs is still in its infancy, the potential benefits to system correctness and reliability of a successful approach make it incumbent upon designers to provide usable formal semantic definitions for new languages.

Our conclusion is that a language should facilitate formal semantic specification where other design goals are not compromised.

This implies that designers should always be cognizant of the possible impacts each language decision may have on the formal specifiability of the language, and compare alternative language features with respect to their formal semantic aspects.

A second goal is that the formal semantic definition be simple. This is not so much a product of the defining process as it is a result of simplicity in the language. A language having complexly interacting features and a profusion of special cases breeds a complicated formal definition, if one is possible at all. Conversely, a clean language with orthogonal features and unifying concepts bears fruit in the form of simplified semantics.

During the design of REDL, the objectives of simplicity and formal specifiability were carefully weighed. In several instances, decisions made on formal semantic grounds were found to be justified on the basis of other goals, particularly simplicity and understandability. For instance, Ironman 7E calls for permitting some kinds of side effects in functions. However, we feel that the prohibition of all side effects is valid from the standpoints of formal specifiability, consistency, and reliability. Our design choices concerning such features as aliasing, monitors, and assert statements were also influenced by the examination of formal semantic implications. The remainder of this section contains an abbreviated presentation of the formal semantic definition of several aspects of REDL, and a discussion of some areas such as real-time parallel processing for which complete formalization techniques have not been developed.

4.1.2 Formal Definition of REDL.

There are two aspects of a programming language which must be supplied in a formal specification: the syntax and the semantics. The formal definition of REDL is achieved through

the use of a BNF grammar and an attribute grammar [Kn68]. The complete BNF syntax, expressed as a LALR(1) grammar in the Informal Language Specification, specifies the context-free aspects of REDL, and will be employed unchanged in the test translator. The attribute grammar serves to define context-sensitivities of the grammar and to indicate the semantics (both static and dynamic) of syntactic units. The attributes specify context-sensitivities such as "The identifier following a closing bracket (e.g., END WHILE L) must be the same as the identifier associated with the matching opening bracket (e.g., L: WHILE...)" ; "static" semantics such as type checking; and "dynamic" semantics such as the rule of inference for a construct.

We present the following example, one variety of if-statement, in order to demonstrate the application of this definitional technique to REDL.

Syntax: `<if-statement> ::= IF <expression> THEN <body> END IF`

Context-sensitivities:

- (1) $\text{error}(<\text{if statement}>) = \text{error}(<\text{expression}>) \vee \text{error}(<\text{body}>) \vee (\text{type}(<\text{expression}>) \neq \underline{\text{boolean}})$
- (2) $\text{environ}(<\text{expression}>) = \text{environ}(<\text{if statement}>)$
- (3) $\text{environ}(<\text{body}>) = \text{environ}(<\text{if statement}>)$

Semantics: `inference_rule(<if statement>) =`

$$\underline{P \wedge (<\text{expression}> \{<\text{body}>\} Q, P \wedge \sim <\text{expression}> \Rightarrow Q)} \\ P \{<\text{if statement}>\} Q$$

We now explain the notation.

Syntax. This is a standard BNF production with `<if statement>`, `<expression>`, and `<body>` as "syntactic categories" (or "non-terminal" symbols), and IF, THEN, and END as "terminal symbols."

Context sensitivities. Restrictions which are not captured by the BNF syntax are established by the "attributes" displayed in

rules (1), (2), and (3). Rule (1) defines the value of the error attribute for <if statement> to be "synthesized" from the error attributes of the non-terminal constituents on the right-side of the BNF production. For example, if error (<expression>) is true (this could occur if an identifier appearing in the <expression> had not been declared), then the error attribute for the <if statement> would also be true. Rules (2) and (3) define environ as an "inherited" attribute; i.e., the value of environ for the non-terminals on the right side of the BNF production is determined by the value of the environ attribute of the left side. In general, environ values will be synthesized during the application of BNF productions for declarative statements, then inherited during the application of productions for executable statements. We note that the condition for program legality can be stated fairly simply using the attribute notation; viz., that error (<program>) is false, where <program> is the starting non-terminal for the BNF production.

Semantics. The semantics for <if statement> is embodied in the "inference rule" displayed above. We read this rule as follows, where P and Q are arbitrary logical formulas:

Assume that these assertions are true:

- (1) If, before the execution of <body>, both P and <expression> are true, then, after the execution of <body>, Q is true.
- (2) If P is true and <expression> is false, then Q is true.

Then the following assertion is also true:

If, before execution of the <if statement>, P is true,
then Q is true after execution.

The most difficult group of attributes to develop for formal specification are the inference rules and assertions defining

dynamic semantics. The remainder of this discussion is devoted to exhibiting a portion of the formal semantics of REDL.

4.1.3 Axiomatic Semantics.

For this presentation we choose the axiomatic approach for defining the semantics of REDL. Hoare describes this method in [Hoa 69], with which the reader is assumed to be familiar. The notation employed will be:

- (1) $P\{S\}Q$ is an assertion that the if the predicate P is true before statement S , and if the execution of S completes, then Q is true.
- (2) $\frac{H_1, \dots, H_n}{H_{n+1}}$ is a rule of inference that whenever assertions H_1, \dots, H_n are true, then it is valid to treat H_{n+1} as a true assertion.
- (3) P_y^x represents the predicate obtained by uniformly replacing all free instances of x in P by y .

4.1.4 Relation to Pascal

REDL has a great deal in common with Pascal, as is discussed in the Informal Language Specification. Instead of exhibiting the complete axiomatic semantics of REDL, we will compare it with Pascal's formal semantics, as defined in [HW73]. Various differences will be described.

Note that [HW73] states that the formal semantics of Pascal apply only when functions do not have side effects. Since REDL prohibits such side effects, the formal semantics apply to all programs.

4.1.5 Data Types.

Enumeration types are similar in both languages, though REDL has both ordered and unordered enumeration types, and several other

functions. Additional axioms are:

TYPE T: ORDERED ENUM(c_0, c_1, \dots, c_n)

- 1) $c_0 = \text{LOW}(x), c_n = \text{HIGH}(x)$
- 2) $i = \text{FIXED}(c_i)$ for $i=0..n$
- 3) $c_i = T(i)$ for $i=0..n$
- 4) $u < v \equiv \text{FIXED}(u) < \text{FIXED}(v)$

If T is unordered, then only the equality - inequality axiom holds, as no other functions or comparison are defined on elements of T.

The BOOLEAN, CHAR, STRING, array, record, file and pointer types have essentially the same semantic specification in both languages. FIXED is a considerable generalization of Pascal's integer, and its semantics is described in the Informal Language Specification. A formal treatment is not provided here, not due to definitional difficulties, since FIXED is based on exact arithmetic, but rather in the interest of brevity.

FLOAT presents the problem common to all language definitions of trying to describe inexact arithmetic within a formal system. This difficulty is eased somewhat in REDL because the precision and range of FLOAT are defined in a relatively machine-independent manner. In most other languages, these properties are heavily implementation-dependent, making complete formal specification impossible.

REDL UNIONS replace the mixed concept of Pascal variant records.

TYPE T: UNION($s_1:T_1; \dots; s_n:T_n$)

Let x_i be an element of T_i .

- 1) $T(s_i:x_i)$ is an element of T.
- 2) These are the only elements of T.

3) $T(s_i : x_i) \$ s_i = x_i$ for $i = 1..n$

The BITS type, an array of booleans, is included instead of sets.

Parameterized types in REDL have no analog in Pascal. They are provided to allow for the specification of array bounds at run-time, particularly the bounds on formal parameters. The only effect on the semantics is that the bounds on the parameterized array type T are associated with each element of T, instead of with T itself.

4.1.6 Declarations.

As in Pascal, declarations serve to associate names with sets of properties. In a <body> consisting of declarations D followed by statements S, the assertions created by the declarations are used in defining the effect of the statements. However, since the names are not known outside the <body>, they do not appear in the assertion describing the <body>. As stated in [HW73], if H is the set of assertions about D, then

$$\frac{H \vdash P\{S\}Q}{P\{D;S\}Q}$$

is a rule of inference, where P and Q do not contain names of D.

Functions and operators have virtually the same semantics in REDL; we will here consider only function declarations.

```
FUNCTION f(F)RESULT r:T;
  :IMPORT I;
  S;
END FUNCTION f;
```

Let x be the set of formal parameters in F, and y be the set of imported (and pervasive) identifiers in the import list I. If $P\{S\}Q$ is an assertion with r not free in P and x not free in Q, then

1) $P \Rightarrow Q_f(\underline{x}, \underline{y})$

is a valid implication.

A procedure can be viewed as a function that returns multiple results. Thus a procedure declaration

PROCEDURE p(f); !IMPORT I;S;END PROCEDURE p;
establishes the implication

2) $P \Rightarrow Q_{f_1}(\underline{x}, \underline{y}), \dots, f_m(\underline{x}, \underline{y}), g_1(\underline{x}, \underline{y}), \dots, g_n(\underline{x}, \underline{y})$

where f_i is the function computed for the VAR or RESULT parameter x_i , and g_i is the function computed for the non-READONLY imported variable y_j , by P. $P\{S\}Q$ is an assertion as stated above.

4.1.7 Statements.

The standard assignment axiom holds in REDL, since functions have no side effects:

1) $P_x^y\{x:=y\}P$

Similarly, the effect of a procedure call is described by:

2) $P_f, (\underline{x}, \underline{y}), \dots, f_m(\underline{x}, \underline{y}), g_1(\underline{x}, \underline{y}), \dots, g_n(\underline{x}, \underline{y}) \quad \{\text{CALL } p(\underline{x})\}P$

The simultaneous, independent substitutions are well-defined in REDL, because aliasing is prohibited. (This does not hold for all pointers, where storage sharing is an implicit property. It does hold, however, for POINTER(CONST T) types, since the values in shared locations cannot change.) Otherwise, the order in which substitutions occurred would affect the axiom's meaning.

The BEGIN, IF, SELECT, WHILE, REPEAT, and FOR statements in REDL have their corresponding analogs in Pascal. The axioms are similar to those in [HW73].

The unordered FOR statement has slightly different semantics, since the separate iterations are independent and do not

employ induction.

$$3) P \& i \in [a..b] \{S\} Q$$

$$\underline{P \{ \text{FOR } i \text{ FROM } a..b \text{ DO } S; \text{ END FOR} \} Q}$$

The ASSERT statement has the axiom

$$4) P \{ \text{ASSERT } Q \} P \wedge Q$$

because the X_ASSERT exception will be raised during execution if Q is false, and execution will not continue at the following statement. Also note that the prohibition of side effects in functions implies that the execution of ASSERT Q cannot effect the truth of P.

Labeled statements exhibit more complex formal semantics due to their interaction with GOTO and EXIT.

$$5) P \{ S \} Q; P \{ S \} \vdash (P \{ \text{GOTO } L \}, Q \{ \text{EXIT } L \})$$

$$\underline{P \{ L:S \} Q}$$

This rule of inference expresses that if S transforms P into Q, and if P true before S proves that P is true before all instances of "GOTO L," and Q is true before each "EXIT L," then P{L:S}Q is a valid assertion. Note that applying this rule with GOTOS is difficult, because they do not correspond to the program structure. EXITS, on the other hand, are better suited to the formal semantics because their scope is lexically contained within the labeled statement.

The ON statement can be described in a somewhat similar manner.

$$6) P \{ S_1 \} Q, R \{ S_2 \} Q; P \{ S_1 \} \vdash (\forall i \quad R \{ \text{RAISE } x_i \})$$

$$\underline{P \{ \text{ON } X \text{ IN } S_1 \text{ DO } S_2 \text{ END ON} \} Q}$$

The x_i are exception names in the list X. The formal semantic difficulty is that "RAISE x_i " may be executed by any routine

invoked by S_1 , making it difficult to statically specify each "RAISE x_i ." In the simple case where every "RAISE x_i " appears lexically within S_1 , the ON statement is equivalent to

```
success: BEGIN  
exception: BEGIN  
    S1;  
    EXIT success;  
END BEGIN;  
  
S2;  
END BEGIN;
```

where every "RAISE x_i " is changed to "EXIT exception." The execution of such statements will cause the block labeled "exception" to complete and S_2 to begin. Otherwise S_1 will complete normally and the block labeled "success" will be exited, avoiding S_2 .

4.1.8 Parallel Processing

The semantic specification of the effect of statements in the presence of concurrent execution sequences has always been an extremely difficult task. The meaning of a program depends on what constitutes the basic, uninterruptable unit of computation, whether multiprocessing (several processors) or only multiprogramming (several programs sharing a processor) is provided, the operation of a scheduler, if any, etc. The problem comes in attempting to define the status of shared data, which may be changing due to operations in several processes while being accessed in another.

One valid rule of inference in REDL is

$$1) \quad (P_i \{S_i\} Q_i)$$

$$\frac{\bigwedge_{i=1}^n P_i \text{ (FORK PATH; } S_1; \text{ END PATH; } \\ ! \\ \text{ PATH; } S_n; \text{ END PATH; END FORK}) \bigwedge_{i=1}^n Q_i}{}$$

where the free VAR names in P_i , S_i , and Q_i do not appear in any P_j , S_j or Q_j , with $j \neq i$. This exhibits the case where the parallel paths are independent. Note that shared CONST names can appear in any number of paths, and that variables declared within a path are known only within that path and thus behave as in a uniprocess environment.

The major construct provided by REDL furnishing safe and simple exclusive access to shared data is the monitor. The objects declared in a monitor can be used only by routines of the monitor, and at most one path is "holding" the monitor at a time. The standard axioms of sequential programs are applicable to monitor objects when processing statements in the monitor.

Clearly, devices other than monitors are needed for achieving cooperating sequential paths. Some helpful axioms can be derived from properties of EVENTS, the devices through which synchronization is achieved. Let $|WAIT e|$ be the number of "WAIT e" statements executed, and $|SEND e|$ be the number of sends plus the initial value of the event e . Then the following axioms describe some properties of e .

- 2) $|SEND e| \geq SENDER(e) \geq 0$
 $|WAIT e| \geq WAITERS(e) \geq 0$
- 3) $SENDER(e) > 0 \Rightarrow WAITERS(e) = 0$
 $WAITERS(e) > 0 \Rightarrow SENDER(e) = 0$
- 4) $|SEND e| - SENDER(e) = |WAIT e| - WAITERS(e)$

As for the synchronizing primitives, in a multipath context outside of monitors, the general axiom

5) $P_E\{S\}P_E$

holds, where P_E is a predicate restricted to objects which are exclusive to the path, and S is a send, wait, or pause statement.

Due to the mutual exclusion property of monitors the stronger axiom

6) $P_m\{S\}P_m$

is valid, where P_m is restricted to monitor objects (which may be shared), and S is a synchronization statement without the release clause.

Monitors also permit the application of a technique for proving monitor invariants, predicates true of monitor objects while no path is holding the monitor. This compensates for the loss of exclusive control of the monitor caused by release clauses of sends and waits. A predicate P is a monitor invariant if

- a) basis: P is true when the monitor is created,
- b) induction: (P on monitor entry and P after each SEND-RELEASE and WAIT-RELEASE) implies (P on monitor exit, and before each SEND-RELEASE and WAIT-RELEASE),

or symbolically

P initially; $P\{M\}$, $\{\text{SEND-RELEASE}\}P$, $\{\text{WAIT-RELEASE}\}P$
 $\{M\}P, \{\{\text{SEND-RELEASE}\}, P\{\text{WAIT-RELEASE}\}$

P is a monitor invariant

The formal semantic definition must also specify items such as expressing legal state transitions of paths from waiting to running, etc., the influence of priorities on scheduling, that processors will not be idle if there are paths eligible to be

run, etc. Axiomatic methods do poorly in describing such issues; approaches that are more algorithmic and interpreter-oriented will be needed.

4.2 Documentation Requirements.

As a production language, REDL will require support documentation which will include a Language Reference Manual, a Language User's Manual, and a Language User's Guide. What follows is a brief discussion of the organization and contents of each of these documents.

4.2.1. Language Reference Manual.

This document is characterized by its organization--it is organized for efficient accessing of information, rather than for the presentation and development of language concepts. Related language features and the rules for using them are grouped together, so that, for instance, all the language defined types and the rules for creating new types will be presented in a single chapter. Although the manual's primary purpose is reference, not teaching, the reader who already understands the fundamental principles of programming languages will not have trouble learning REDL from it. It provides clear answers to such questions as "What statements are in the language?", "Can I use dynamic arrays?", and "What can I pass as an actual parameter to a formal parameter of the RESULT binding class?". In order to resolve more concept oriented questions, such as "How do I write a factorial program in REDL?", the reader will have to follow forward and backward references from chapter to chapter.

The traditional organization for a Language Reference Manual has been to put "value set" information in the type section and information on the operations in the language in the expression section (for example [Nau60]). However, many recent language reference manuals have followed Hoare's [Hoa72] description of types as a combination of both value sets and the operations which are legal on those value sets, and have put the information

on operations into the type section (for example [JK 75]). It is this latter approach which the REDL Language Reference Manual will follow.

The language will be described by a combination of syntax and semantic rules. The syntax rules will be described by "syntax diagrams" in order to maximize readability. These diagrams allow an overall view of a construct which would otherwise require many lines of BNF to express. The semantic rules will be presented in English. A model for the organization and style of the document is [Kos 76].

4.2.2 Language User's Manual.

A Language User's Manual has a very different organization. As a teaching document, it is designed to be read in order. Examples will be presented from the beginning and will be shown as complete programs so that the reader will always be able to write executable test programs based on the examples. The description will start with only a few concepts which will be added to in a manner that relates concepts functionally rather than syntactically. For example, whereas all types would be grouped together in one chapter in the Language Reference Manual, in the Language User's Manual arrays will be discussed in the section on the for-statement and unions will be discussed in the section on the select-statement.

A Language User's Manual can be designed for a variety of audiences. It can be designed for the non-programmer as an introduction to both programming and to the language, or for the experienced programmer to teach the concepts which differentiate this language from other languages. This latter approach will be adopted in the Language User's Manual. It is assumed that most readers of this manual will be either experienced Pascal programmers or experienced FORTRAN programmers. By identifying

the audience, we can describe many language concepts by simply relating them to similar concepts already known to the reader. For example, the concept of assignment is present in REDL, FORTRAN and Pascal. To teach assignment in REDL, one need only describe the semantics unique to REDL. Block structure is present in REDL and Pascal. For a Pascal audience, block structure can be presented in terms of its similarities and differences from traditional block structure, but for a FORTRAN audience one needs to teach block structure.

The Language User's Manual will be designed to also be a manual of programming style, in that it will include discussions of the safety of various features and will describe clean and "structured" approaches to the use of the features. All examples will be designed to encourage good programming style. An outline of the Language User's Manual follows.

1.0 INTRODUCTION

Purpose and scope of the document; overview of the language.

2.0 BASIC CONCEPTS

Manual notation; lexical information; program declaration; declaration and statement sequences; simple I/O and assignment; simple variable declaration; comments; comparison with Pascal and FORTRAN.

3.0 ARITHMETIC TYPES

Fixed and float types; variable declarations (initialization); range declarations for simplification; arithmetic operations, assignment and explicit conversions.

4.0 TRANSFER OF CONTROL

IF..THEN..ELSE; boolean type; variable declaration (initialization); boolean operations and assignment; open scopes; IF..THEN..ELSEIF; exit statement; SELECT; enumeration

type and declaration; variable declaration (initialization); enumeration operations and assignment; unconditional transfer of control.

5.0 LOOPS

WHILE (zero or more); REPEAT (one or more); FOR; array type and declaration; dynamic arrays - parameterized types; variable declaration (initialization); array operations and assignment (subscripting and slicing).

6.0 RECORDS AND UNIONS

Record type and declaration; variable declaration (initialization); record operations and assignment (selection); union type and declaration; variable declaration (initialization); SELECT; union operations and assignment.

7.0 ROUTINES

Procedure delcaration; formal parameter binding classes; import directive; closed scope; CALL statement; actual parameters; function and operator declaration; local result variable; exit statement; no side effects (restrictions on parameters); function and operator evaluation; overloading.

8.0 CAPSULES

Motivation; capsule declaration; export directive; restrictions on exported variables; opacity of exported types.

9.0 EXCEPTION HANDLING

Exceptions; establishment and suppression of exceptions; raising of exceptions; handling of exceptions; system-defined exceptions.

10.0 PARALLEL PROCESSING AND REAL-TIME FACILITIES

FORK statement; path priorities; interactions between paths (termination); synchronization (signals); mutual exclusion (monitors); real-time clock.

11.0 MODULE LINKAGE

Compilation modules; access and promote directives; building libraries.

12.0 SYSTEMS PROGRAMMING

Motivation; pointer type; variable declaration; pointer operations and assignment (dereferencing); bitstring type declaration; variable declaration; bit operations and assignment; machine records; assembly language routines.

13.0 COMPILE-TIME FACILITIES

Motivation; configuration directive; compile-time declarations and expressions; type, representation and generator predicates; compile-time conditional statement; inline directive; optimization directive.

4.2.3 Language User's Guide.

The Language User's Guide will provide the implementation-dependent information necessary for a specific implementation of REDL. It will contain information about the environment of the compiler or interpreter and show how to run a program in REDL. It will not describe the language itself other than to list implementation-specific restrictions, if any (such as the maximum number of nesting levels). It will also contain the error messages produced by the compiler or interpreter.

4.3 REDL Implementation.

This section describes the design of the Test Translator to be implemented during Phase 2 of the contract. The Test Translator will be developed on a PDP-10 computer running the TOPS-20 operating system. The computer selected will be attached to the ARPANET. The Test Translator will be written in the Simula language. The version of Simula used will include the HIDDEN PROTECTED specification. The Test Translator will be an interpreter rather than a compiler. (In the following discussion we shall use the term "REDL interpreter", or simply "interpreter", instead of "Test Translator".) It is not intended that the machine-dependent features or the separate compilation facilities be supported in the interpreter.

4.3.1 Design Goals.

This section discusses several important goals that influence the design of the REDL interpreter.

4.3.1.1 Modularity. The interpreter will be designed with an emphasis on modularity. This aids the distribution of programming assignments during implementation, provides a framework for verification and simplifies maintenance. The most important features of Simula for the higher modular levels are procedures (for creating abstract operations) and classes (for creating abstract data structures). The top levels of the design decomposition will use both of these facilities of Simula extensively.

One important technique for achieving modularity is to design language-dependent operators to be table driven. Both the lexical analysis and parsing algorithms will be table driven. In addition, the semantic and interpreter execution parts of the interpreter will also be table driven.

4.3.1.2 Memory Utilization. The entire interpreter together with its tables will be resident in main memory. This simplifies the interpreter design, makes implementation easier, and makes the resulting interpreter faster. The main disadvantage will be the large amount of memory required to run the interpreter. The interpreter will, however, be written so that overlaying of the code will be possible. Overlays will reduce memory requirements without having any significant disadvantage.

4.3.1.3 Optimization. The interpreter will not perform optimization. This will make the interpreter much easier to design and implement. The lack of optimization has no disadvantage, since the purpose for the interpreter is to provide for testing the language as opposed to demonstrating run-time efficiency.

4.3.1.4 Error Diagnosis. Since considerable testing of programs written by people with no prior knowledge of the language is planned, the interpreter will include extensive error diagnosis and recovery facilities. Errors will be reported both at "compile-time" and "run-time" by both an error number and a line of descriptive English text.

4.3.1.5 Statistics Collection. The interpreter will include a provision for collecting statistical information about the features of programs being run. Statistics about errors in the programs will also be included. This information will be useful in improving the language design, and in locating troublesome language features or language feature descriptions.

4.3.2 Design of REDL Interpreter.

Figure 4-1 shows the top level modules in the REDL interpreter. Figure 4-2 is a key to the notation used in the figures of this section. In Figure 4-1, data flow and control flow connections to SYMTAB and ERROR are not shown since they are connected to almost all of the other modules. Each of the modules in Figure 4-1 will now be briefly discussed.

(1) LEXER (a class) - This module performs lexical analysis. It reads the source program and produces a stream of tokens (lexical units). It also enters names into SYMTAB. LEXER is discussed in more detail in 4.3.2.1.

(2) PARSER (a class) - This module parses the programs. It uses the tokens produced by LEXER and produces a reduction stream which consists of tokens and reduction numbers. PARSER is discussed in more detail in 4.3.2.2.

(3) TREE (a class) - This is a data structure that holds the parsed program. Its value is constructed initially by BUILDER and takes the form of a pointer-connected parse tree. PREPASS manipulates this initial tree to produce a modified tree. This modified tree is then the primary input to RUN.

(4) BUILDER (a procedure) - This procedure uses the reduction stream produced by PARSER to construct the initial value of TREE. BUILDER is discussed in more detail in 4.3.2.3.

(5) PREPASS (a procedure) - This procedure takes the initial value of TREE and produces a modified value. Its main purposes are the processing of compile-time language facilities and implementation of name-scope rules. It also collects information on name usage in XREF. PREPASS is discussed in more detail in 4.3.2.4.

(6) XREF (a class) - This data structure contains information about usage of names. The initial value of XREF is created by PREPASS. This initial value is used by NAMEANAL.

(7) NAMEANAL (a procedure) - This module examines the name reference information in XREF to check the validity of any safe, recursive and reentrant directives in the program. It also produces a name cross reference listing. NAMEANAL is discussed in more detail in 4.3.2.5.

(8) SYMTAB (a class) - This is a data structure that contains information about all names. Names are initially entered by LEXER. Type information is added by PREPASS. PREPASS also changes all the name references in TREE to their corresponding SYMTAB index. SYMTAB is discussed further in 4.3.2.7.

(9) RUN (a procedure) - This procedure executes the program contained in the modified value of TREE. RUN is discussed in more detail in 4.3.2.6.

(10) REDL (a procedure) - This is the main procedure of the interpreter. It calls, in order, BUILDER, PREPASS, NAMEANAL, and RUN.

(11) ERROR (a procedure) - This procedure establishes a uniform way of reporting all errors encountered during "compilation" and "execution".

It must be realized that the design presented here is initial. Further refinement will require minor modifications; however, the overall structure will be basically unchanged.

4.3.2.1 LEXER. LEXER and its submodules are shown in Figure 4-3. LEXER includes three classes that produce streams: LINES, CHARS, and LEX. Each stream-producing class includes three operations:

- (1) INIT - called initially to open the stream
- (2) TERM - called at the end to close the stream
- (3) NEXT - called to obtain the next item in the stream.
NEXT must be called after INIT and before TERM.

Each of the modules of LEXER will now be discussed.

(1) LINES (a class) - This class produces a line stream. NEXT returns the next line of the source program. LINES also produces the source program listing.

(2) CHARS (a class) - This class produces the character stream. NEXT returns the next character in the source program together with its character class. Character classes are given in Appendix B of the Informal Language Specification.

(3) LEX (a class) - This class produces the token stream. NEXT returns the next lexical token in the input stream. Tokens in general have two parts: a terminal number (see Figure 4-4) and a symbol table index. All tokens include the terminal number. Some tokens are completely identified by their terminal number. Others indicated by names enclosed by "<>" in Figure 4-4 are entered into the symbol table and their symbol table indices are also included to provide unique identification. LEX is implemented by means of a finite state machine (FSM) recognizer. This FSM is encoded as tables that are used to drive LEX. The FSM to be used is derived from the Lexical Diagram in the Informal Language Specification.

(4) RESERVED and CT_RESERVED (procedures) - These procedures are used to look up reserved names and compile-time reserved names (reserved names that begin with "%") respectively. These procedures are automatically generated from tables produced by the LALR analyzer (see 4.3.2.2).

4.3.2.2 PARSER. PARSER and its submodules are shown in Figure 4-5.

(1) PARSE (a class) - This class produces the reduction stream (see 4.3.2.1 for a discussion of stream producing classes). NEXT returns either the next token or reduction. A reduction is the number of the reduction that is to be applied to the preceding part of the reduction stream. Reduction numbers appear in the LALR(1) grammar displays in Appendix D of the Informal Language Specification. PARSE is a LALR(1) parser that is

table driven. The tables are produced by a grammar-driven LALR parser generator.

(2) PARSE_STACK (a class) - This class implements the parse stack used during the LALR analysis. Its operations are:

INIT - initializes the stack to empty
PUSH
POP

(3) RECOVER (a procedure) - This procedure is called when a parsing error occurs. It reports the error (via ERROR) and attempts to make an intelligent recovery.

4.3.2.3 BUILDER. BUILDER and its submodules are shown in Figure 4-6.

(1) BUILD_STACK (a class) - This class implements a stack used in building the initial parse-tree. Elements of the stack are either tokens or tree indices. Its operations are:

INIT
PUSH
POP

(2) BUILD (a procedure) - This procedure builds the initial parse tree from the reduction stream. It gets items in the reduction stream and takes the following actions. If the item is a token it is pushed onto the BUILD_STACK. If the item is a reduction, then let n be the number of items on the right-hand-side of the reduction. If n is greater than one, the top n items are popped from the parse stack and are used to build a new tree node. The index of this new tree node is then pushed on the BUILD_STACK. If n is equal to one, then no action is taken. If the right-hand-side of the reduction includes terminals with no associated symbol table indices (see 4.3.5.1) then these terminals are not included in the built node.

4.3.2.4 PREPASS. PREPASS and its submodules are shown in Figure 4-7.

(1) PREP (a procedure) - PREP takes as input the initial value of TREE and produces a modified value of TREE. The following modifications are performed.

(a) Name references are replaced by symbol table indices. A specific line for each overloaded name reference is selected.

(b) Compile-time constant declarations are evaluated and then deleted.

(c) Compile time statements and call statements are expanded.

(d) Namescope directives are deleted.

The following information is also collected by PREPASS.

(a) Type information for each declaration is recorded in the symbol table.

(b) Values of compile time constant expressions are recorded in the symbol table.

(c) Name reference information is recorded in XREF.

The following checking is also performed.

(a) Ensure that every name is declared.

(b) Check for multiple declarations of names.

(c) Ensure that sequences of directives, declarations, and statements are in the right order.

(d) Check that each kind of directive appears only in permitted scopes.

(e) Each routine call is checked to make sure that the arguments are of permitted types (and representations). PREPASS will be implemented as a set of mutually recursive tree-walking procedures. Note that some subtrees may be walked several times.

(2) NAME_STACK (a class) - This is a stack-like data structure used in implementing name scope rules. Its operations are:

INIT - initializes the stack to empty
PUSH (declaration, name) - associates name with declaration. Name is a symbol table index and declaration is a tree index.
PUSHSCOPE(type) - indicates that a new scope is being entered. Type is either "open", "closed", or "capsule".
POPSCOPE - indicates that a scope is being left.
IMPORT(name) - indicates that the name is to be imported.
EXPORT(name) - indicates that the name is to be exported.
LOOKUP(name) - returns either the current declaration for name or an indication that there is currently no valid declaration.

4.3.2.5 NAMEANAL. NAMEANAL and its submodules are shown in Figure 4-8.

(1) ANALYZE (a procedure) - ANALYZE calls (in order) CLOSURE, NAMESREPORT, and CROSS.

(2) CLOSURE (a procedure) - CLOSURE computes in XREF the transitive closure of names referenced in each routine, path, and monitor.

(3) NAMESREPORT (a procedure) - NAMESREPORT issues error messages for

(a) Violations of the safe, recursive, or reentrant directives.
(b) Illegal monitor recursions.

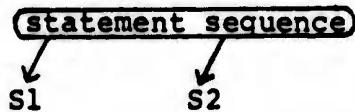
(4) CROSS (a procedure) - CROSS produces a cross reference listing of all names in the program. Type information is included with each name.

4.3.2.6 RUN. RUN and its submodules are shown in Figure 4-9. EXEC is the part of the interpreter that carries out "execution." The program to be "executed" is contained in the modified parse tree in TREE. Since multiprocessing (fork statements) will be supported, EXEC is not implemented as a recursive tree walk. Note that backing out of the recursion would be necessary during a process "switch." Instead of using recursion, EXEC uses continuations. During execution of a process, the continuation for that process contains an indication of the part of the program still to be executed by that process. Some examples may clarify the notion of continuation.

Example 1. Consider two statements

S1 ; S2 ;

which might have the following parse tree:



while S1 is being processed its continuation is S2 (followed by the continuation for the pair S1;S2;).

Example 2. Consider a call

CALL P; S1;

The continuation of the execution of the body of P is S1 (followed by the continuation for CALL P;S1;).

In these examples, it can be seen that continuations function as a kind of stack. This is not always the case however. When a goto statement is executed, the current continuation is thrown away and replaced by a continuation which is the "value" of the target label. Exit and raise statements operate similarly.

Note, too, that goto, exit, and raise statements may cause one or more scopes to be terminated. When a scope is terminated, its storage should be released. This release is handled by

attaching the release as a part of the exiting continuation. A similar approach works to unlock monitors under abnormal exit.

The fork statement will result in the creation of separate continuations for each path. These continuations all end back at the continuation of the fork statement. Process "switching" now can be seen to involve no more than a change to a different current continuation.

The submodules of RUN are now discussed.

1. EXEC (a procedure)--EXEC is the major component of the interpreter for program "execution." It will be factored into a set of subprocedures for each kind of statement, declaration, expression, etc. EXEC also includes a scheduler for switching between processes (that is, between their continuations).
2. CONTINUATIONS (a class)--This data structure holds the continuations for all active processes.
3. PROGIO (a class)--This class supports input/output commands in "executing" programs.
4. PROGDATA (a class)--This class contains storage for all program variables and other data needed during program "execution." It is organized as a "cactus stack" with separate arms for each path. Data for each scope are organized as a stack frame. Offsets into the stack frame are kept in SYMTAB. Variable sized data are represented in each frame by a pointer to the data and to their attributes.

4.3.2.7 SYMTAB. SYMTAB is a class which serves as the interpreter's symbol table. It includes the following information.

1. A string representation for each name in the program (created by LEX).

2. A value for each program literal (created by LEX).
3. An entry for each declaration of each name. This entry includes the type and representation of the declared name. Routine attributes (other than name scope attributes) are also recorded here (created by PREPASS).
4. Stack frame offsets for each variable and other program data (created by PREPASS).

4.3.3 Relationship of Test Translator to Production Compiler.

A substantial part of the code in the test translator can be reused as part of a future production compiler. In particular, all modules, except RUN, which would be replaced by a code generator module, could be used with only minor changes.

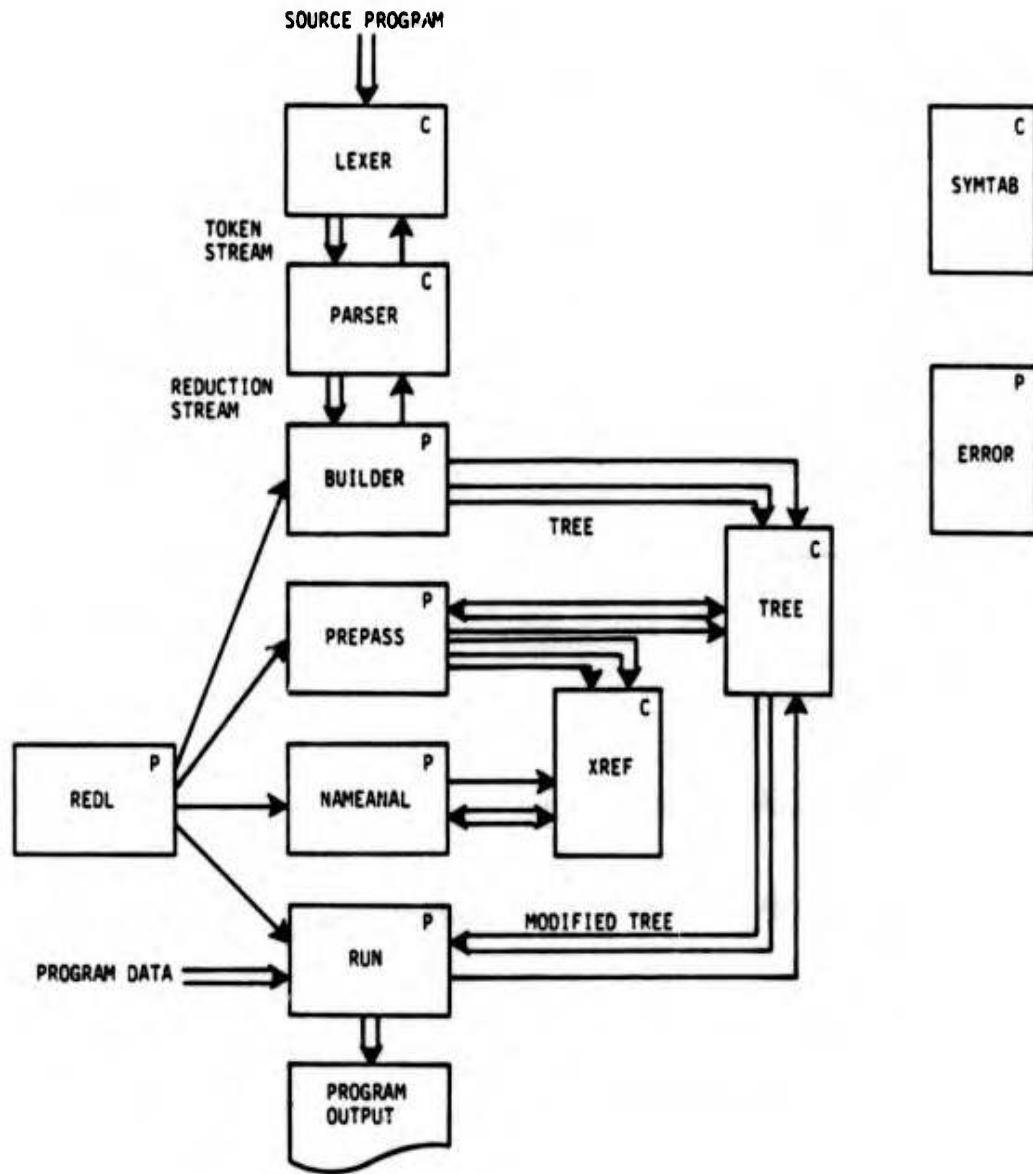


FIGURE 4-1: Major Modules of REDL Interpreter

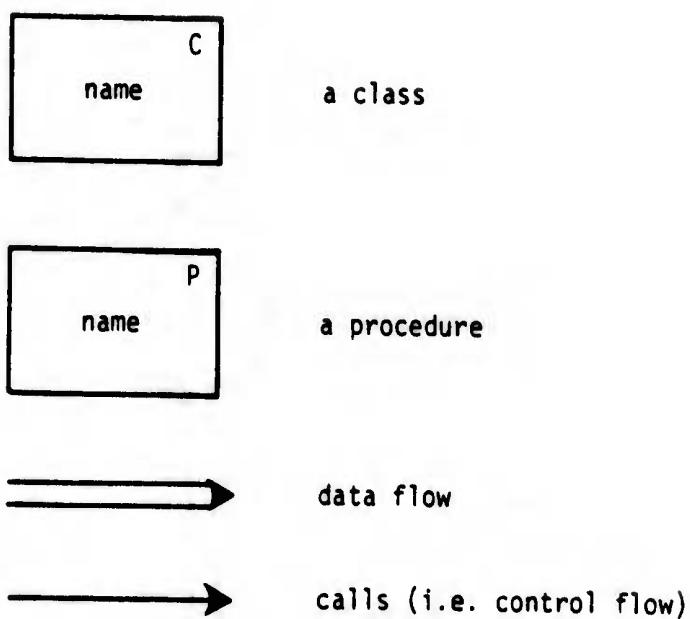


FIGURE 4-2: Notation

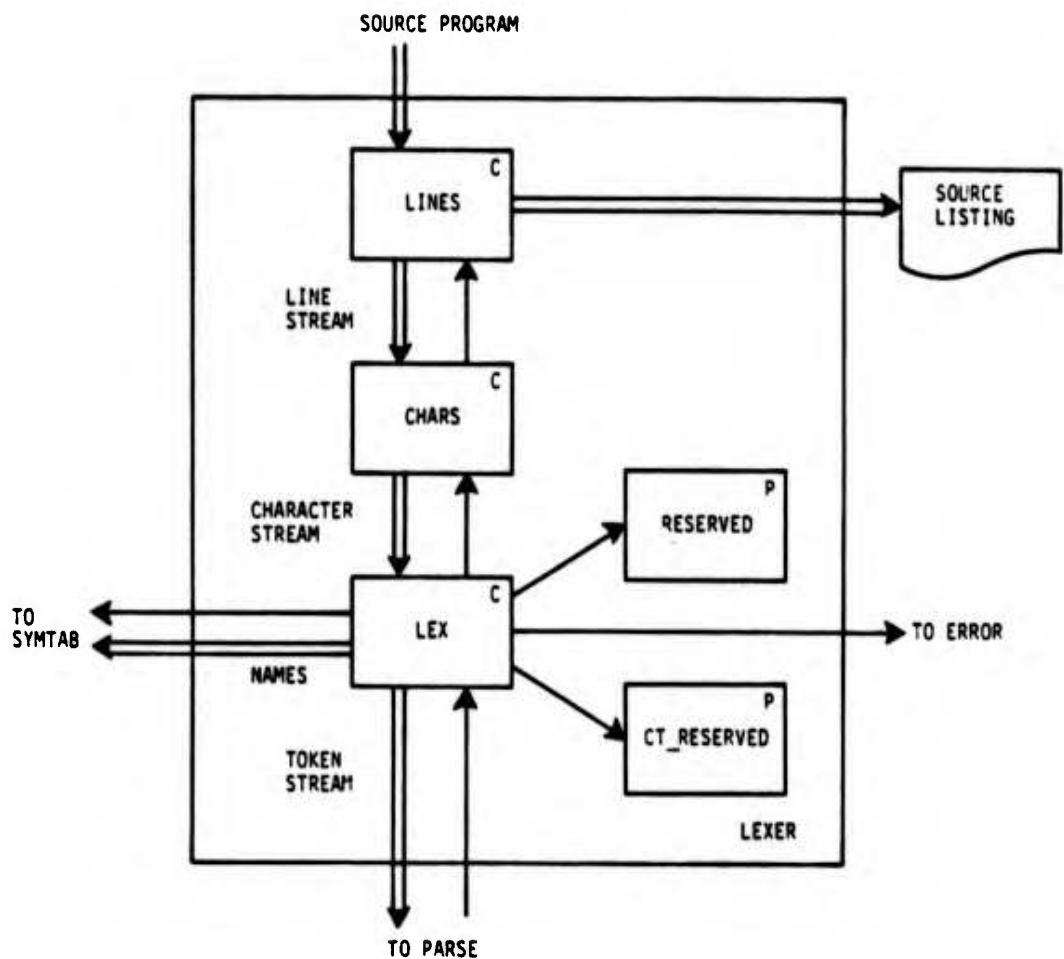


FIGURE 4-3: LEXER

1 .	53 ENUM	113 MRECORD
2 <	54 EXIT	114 ORDERED
3 (55 FILE	115 POINTER
4 +	56 FORK	116 PRCGRAM
5 \$	57 FRCM	117 PROMOTE
6 !	58 GOTO	118 RELEASE
7 %	59 INIT	119 SECONDS
8 *	60 MAIN	120 <STRING>
9)	61 PATH	121 %DECLARE
10 ;	62 SAFE	122 EXTERNAL
11 -	63 SEND	123 FUNCTION
12 /	64 SIZE	124 OPERATOR
13 ,	65 THEN	125 OPTIMIZE
14 >	66 TIME	126 OVERLOAD
15 ?	67 TYPE	127 PRIORITY
16 :	68 UPTO	128 READONLY
17 #	69 WAIT	129 EXCEPTION
18 \$	70 <BIT>	130 OTHERWISE
19 =	71 XCALL	131 PROCEDURE
20 {	72 XELSE	132 RECURSIVE
21 }	73 XTHEN	133 REDECLARE
22 ..	74 ALIGN	134 REENTRANT
23 <=	75 ARRAY	135 %PROCEDURE
24 **	76 BEGIN	136 ASSOCIATIVE
25 >=	77 CONST	137 COMMUTATIVE
26 :=	78 EVENT	138 CONFIGURATION
27 =>	79 PAUSE	139 MACHINE_CODE
28 AT	80 RAISE	140 MILLISECONDS
29 DO	81 RANGE	141 CALLER_ASSERT
30 IF	82 SPACE	142 <SP ENUM SYMB>
31 IN	83 STORE	
32 OF	84 TICKS	
33 ON	85 UNION	
34 OR	86 UNTIL	
35 TO	87 USING	
36 %IF	88 WHILE	
37 _	89 %CONST	
38 AND	90 ACCESS	
39 CAT	91 ASSERT	
40 DIV	92 DOWNTO	
41 END	93 ELSEIF	
42 FCR	94 EXPORT	
43 MOD	95 IMFCRT	
44 NOT	96 INLINE	
45 OFF	97 OFFSET	
46 VAR	98 RECORD	
47 XOR	99 REPEAT	
48 %END	100 RESULT	
49 BITS	101 SELECT	
50 CALL	102 SYSTEM	
51 CASE	103 <EMPTY>	
52 ELSE	104 <FIXED>	
	105 <FLOAT>	
	106 <IDENT>	
	107 XELSEIF	
	108 CAPSULE	
	109 CONNECT	
	110 GROUPED	
	111 LINKAGE	
	112 MONITOR	

FIGURE 4-4: Terminal Numbers

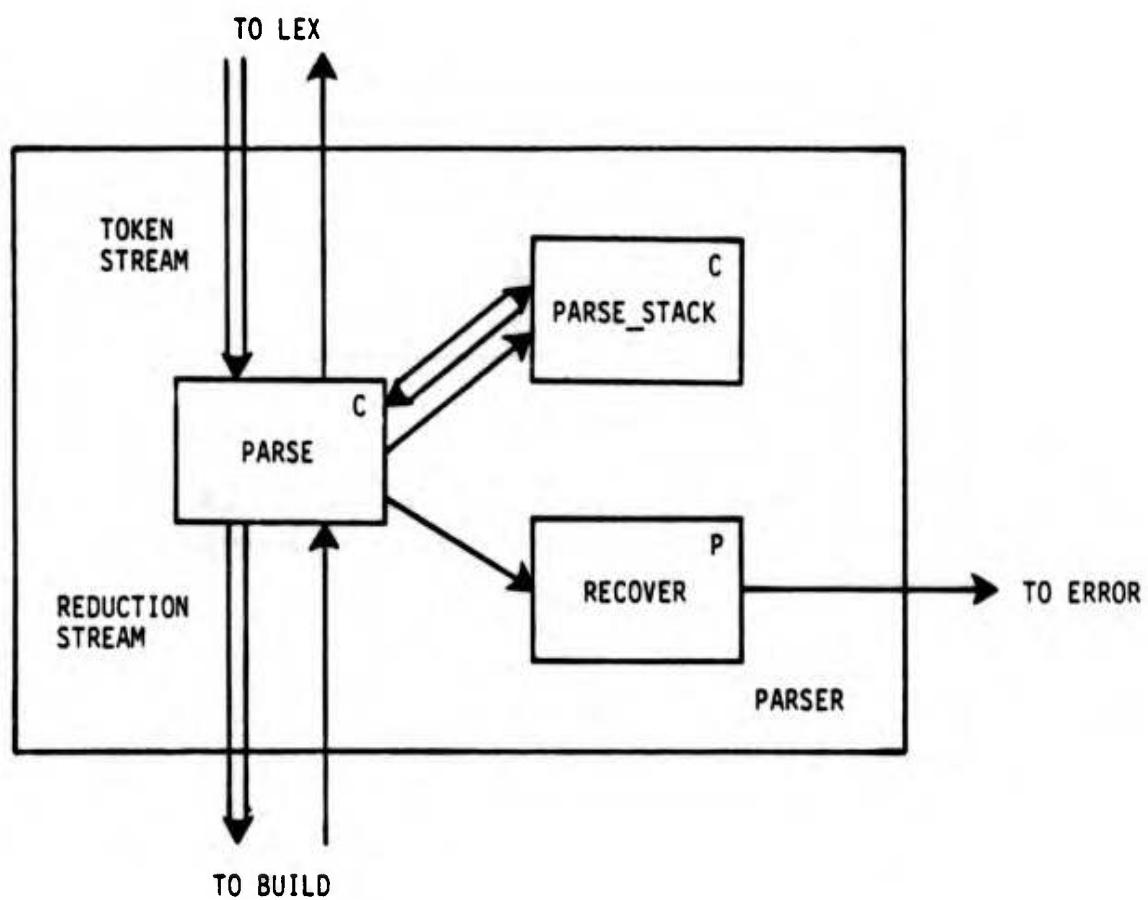


FIGURE 4-5: PARSER

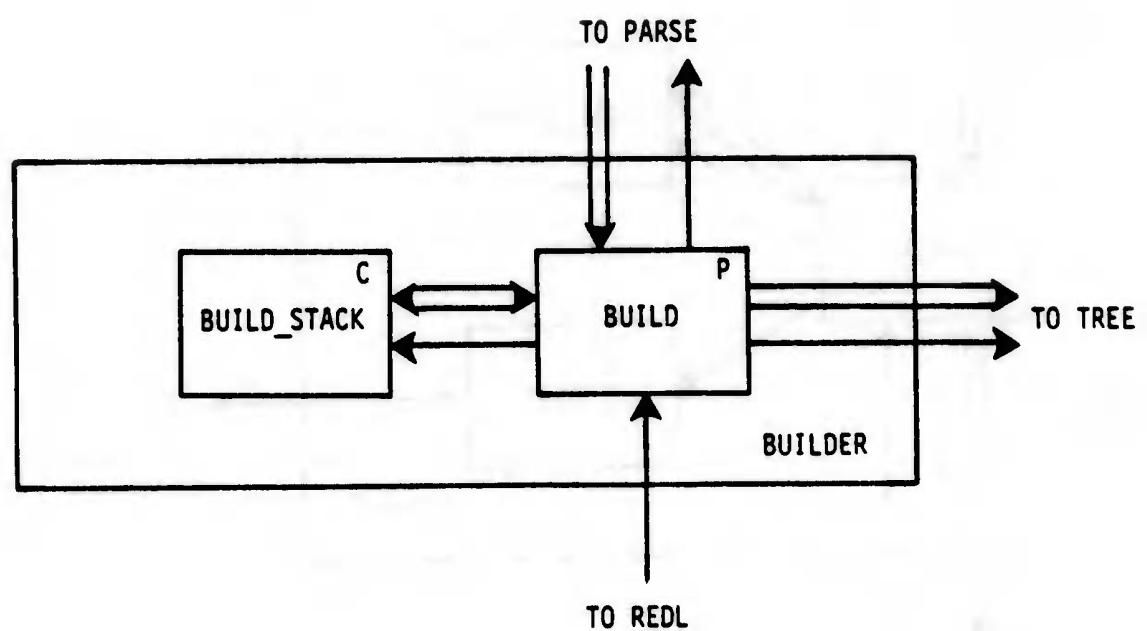


FIGURE 4-6: BUILDER

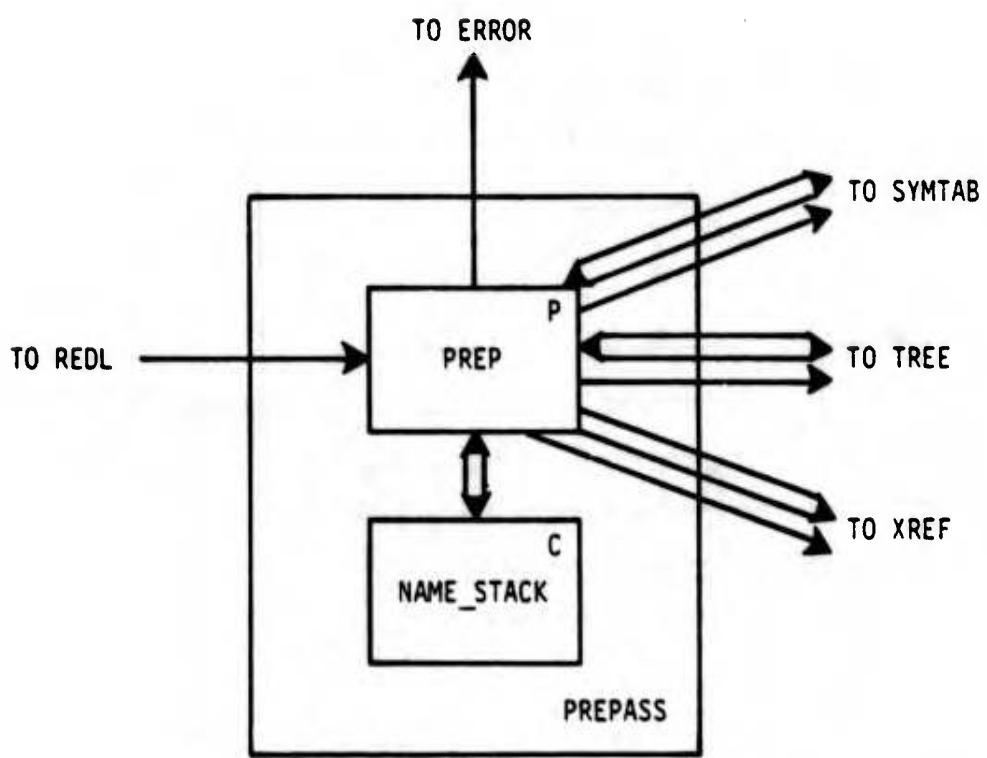


FIGURE 4-7: PREPASS

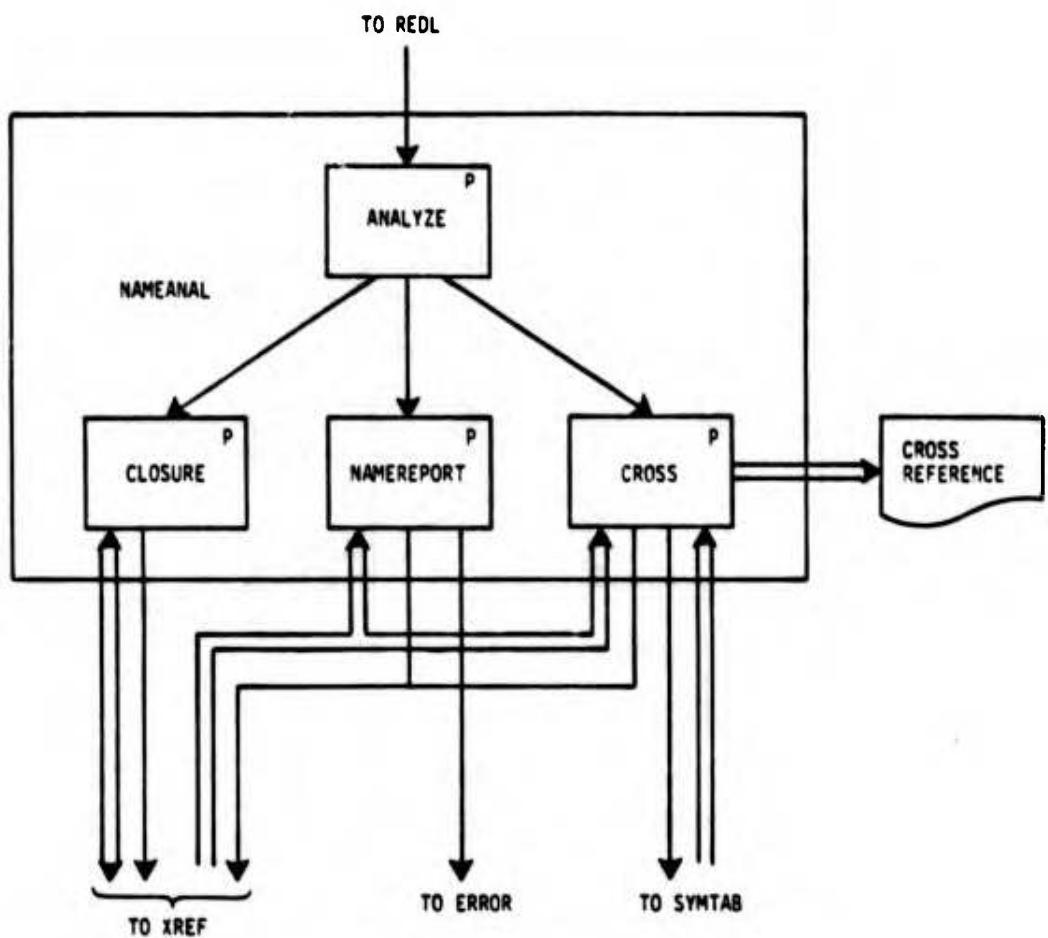


FIGURE 4-8: NAMEANAL

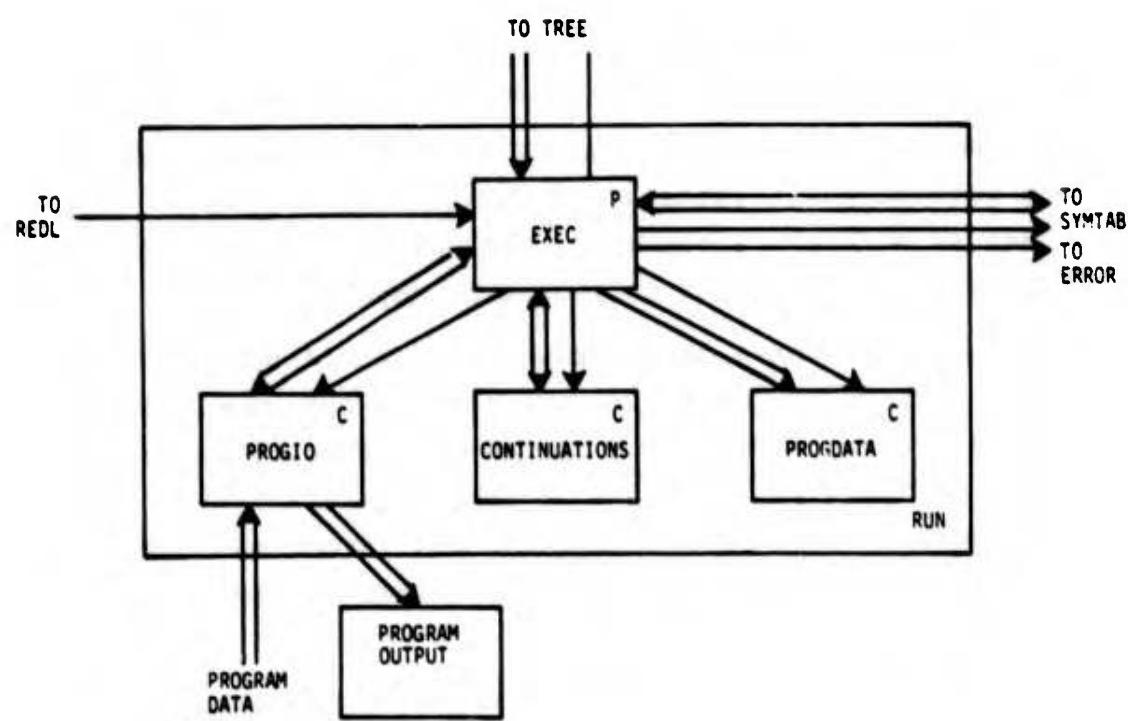


FIGURE 4-9: RUN

4.4 The REDL Language Support Facilities

4.4.1 Introduction.

An optimal environment for software production contains an integrated set of software tools which facilitate the various activities of programmers and project managers. The proper design of such an environment bears as importantly on the life-cycle costs of the software being developed as does the proper design of the principal programming language to be used.

If programmers and project managers are to move about between projects and between installations, the availability of a consistent set of software tools and a consistent interface to them at different installations increases in importance.

This section contains a description of a set of software tools based on REDL as the principal programming language. The tools will be discussed in relation to use with REDL programs. They will also be discussed in terms of implementability in REDL. It is assumed that the host machine and target machine sometimes differ, and most of the tools described are to be run as commands on the host machine.

The software tools to be described form a consistent whole and constitute a very good programming environment. It is not claimed, however, that all desirable tools have been described, nor that all the tools described are of universal applicability. They do, however, present a solid basis upon which to examine the features of the REDL language.

4.4.2 Overview.

4.4.2.1 Categorization of Tools. The tools for software production may be divided into three classes. Except as

otherwise noted, each is a command on the host operating system.

(1) Tools for source program preparation

- (a) editors, source file formatters
- (b) compilers
- (c) linkers, binders

(2) Tools for program testing and debugging

- (a) interpreters
- (b) simulators
- (c) symbolic debuggers (for host machine), debuggers for target machine, core-dump analyzers (for target machine)
- (d) timing and other statistics tools (run-time statistics)

(3) Tools for project management

- (a) automated documentation aids
- (b) global cross-referencer (source-level statistics)
- (c) library maintenance tools
- (d) program analyzers (object-level statistics)
- (e) project management data tools

In addition to the commands listed above, the REDL language support facilities include the following:

(1) Libraries (of source programs, object modules, and interfaces)

- (a) standard system libraries
- (b) applications and project libraries
- (c) user libraries

(2) Project management data

- (a) cross-references and source-level statistics
- (b) program analyses and object-level statistics
- (c) results of simulation and run-time statistics
- (d) documentation
- (e) other information on project progress and status, programmer statistics, project complexity, etc.

4.4.2.2 Assumptions about the Operating Environment. Embedded computer applications constitute a very broad class of programs,

and it is expected that REDL will be used in a variety of operating environments. Although it will sometimes be the case that the host system, supporting the compiler and other development tools, will also be the target (or object) system, it will also be the case that the host system is providing for a smaller and different target machine, a machine having at best a rudimentary operating system, not designed to be used in program check-out.

For this reason, program check-out will sometimes have to be done in an interpretive or simulation fashion on the host (or another large) system. Experience with simulation of large real-time programs for embedded applications suggests that valuable information about program timing, as well as program testing, can be accomplished through a suitable simulation facility.

Wherever a number of distinct target object machines exist for which REDL compilers are to be provided, a useful software development strategy is to develop language-oriented debuggers and simulators on a single machine so that the greater part of compilation for the several target machines can be accomplished with common software (i.e., the common "front-ends" of the compilers). Similarly, in this way the greater part of debugging or simulation can be done with a single set of software tools. A decision to proceed in this fashion will depend on the availability of a common software development system for all of the projects involved.

4.4.3 Libraries and Library Maintenance Tools.

Libraries are valuable in any programming environment. The REDL language facilitates the use of libraries and therefore the REDL compiler, as well as the linker, should be integrated with the library design. In this section, both the structure of a library facility and the integration of that facility with the REDL compiler will be discussed.

4.4.3.1 Library Structure.

Although usable libraries can be implemented directly on the file systems of many operating systems, even systems with quite flexible file systems (such as MULTICS) have chosen to implement additional library facilities beyond those provided by the operating system. On less flexible systems, such extensions would be even more attractive. We therefore define a library file as a single file containing one or more library entries, each library entry being a source file or object module. A library maintenance tool, implemented as a system command, is provided which provides for creating and destroying library files and for adding, deleting, and updating library entries. Library files will also record such information, of interest to project managers, as:

- (1) name of last user to extract (copy) a source file; date of extraction
- (2) name of last user to update a source or object file; date of update
- (3) list of users, programs, projects, or applications which have linked to (or compiled from) a given object module, with dates.

Utility programs to search within library files and to extract object modules from them must, in addition, be provided to the REDL compiler and to the linker.

One of the most interesting features of the library file described above is the management information which is provided. A library which is able to tell a user that a source file has been extracted (copied) for update, but not yet replaced, is of use even on small projects. A library able to inform project management 'who used what and when' is of enormous importance on a large project, permitting the monitoring of project complexity, allowing automated advisories on changes and improvements, specifying necessary recoding or recompilations due to altered interfaces, and so on. Access control mechanisms can, with little additional difficulty, be designed into the library file and

its maintenance tools.

4.4.3.2 Use of the Library by REDL.

REDL promotes modular programming by allowing for "interchangeable parts." There are many reasons why several different versions of a program may exist:

- (1) program stubs are needed during early program development
- (2) differing versions of a program are needed corresponding to differing stages of implementation, completeness, or reliability
- (3) sometimes it is necessary to try out an entirely different algorithm or approach in a program
- (4) differing object modules for the same source program may be desired, corresponding to different object machine configurations, different space/time optimization strategies, etc.

What all the elements in a set of interchangeable parts have in common is an "interface specification" which, in REDL, is called a "template." A REDL program, wishing to use an item promoted from another program, obtains a "template" for that item. At link time, it is required that the templates for items passed between programs agree exactly.

A program may be written which accesses all of the promoted items of the programs in a library and then promotes them all. This single program can be given a preferred position in the library and thus made available to a compiler compiling a program which imports "from the library." Such a program (which might well be called a library program) may without changes be associated with each library in a set of interchangeable libraries.

A REDL library has a list of the templates promoted by the object modules which it contains. A program wishing to use a promoted item may access it with an access directive specifying only the name of the library program which is to be the source

of the access. This frees the accessing program from the need to know how the promoted items of a library are distributed among the other programs in that library. It also frees the accessing program from any dependency on a particular distribution. If whole libraries are to become interchangeable, as happens in the case of "releases" of major software, then it is necessary that programs compiled with ACCESSes from one library be linkable to the object modules of another equivalent library. REDL permits this without difficulty. In most respects, a library as described above is very simple structure. Once designed, a library is easily implemented and independent of REDL except in its collection of "templates."

4.4.4 Editors and Source File Formatters.

Standard conventions for formating source programs are advantageous in large projects. These conventions include code layout conventions (e.g., rules on indenting and on one-line versus multi-line forms for compound statements) and conventions on comments (e.g., conventions on form and content of per-line, per-block, per-routine, and per-program comments).

Editors can be supplied with commands to help in producing standard formated code. The editors may have, for example, commands to facilitate proper indentation and may have commands for creating and properly positioning comments.

Separate programs to format source files properly may also be provided. There is nothing in REDL to mandate combining the editing and formating functions, and a stand-alone formatter may complement an existing editor.

Because of the syntax of its compound statements and declarations, REDL lends itself to the establishment of a standard formating convention. A parser for REDL can, without great

difficulty, be used as the front-end of a stand-alone formating program. Provision of such a separate parser is a reasonable and inexpensive extension of a compiler or interpreter implementation project. Addition of a formater as such to an interactive editor can be difficult depending on the functionality of the editor. A line-number oriented editor capable of moving backward as well as forward over the program text could not easily be made to format its text as well. On the other hand, the provision of some formating aids in an interactive editor might be both easy and attractive. For example, commands to the editor to change and maintain the current level of indentation would be easily implemented, and a command to begin a comment in the conventional position would also be helpful and readily implemented.

4.4.5 Linkers and Binders.

Linkers may differ in a number of respects according to their functionalities. The basic linker links together separately compiled object modules into a loadable or executable core image. In some cases it is desirable to provide for the linking of several object modules to produce another object module which may in turn be linked or bound; this kind of linking is called binding. A binder is useful in cases where large programs are being re-linked repeatedly after recompilation of only a few of the component object modules. The unchanged part need be bound only once, thus greatly reducing linking times.

Linkers may be designed to support various other features. For example, linkers may support an overlay facility and may have features to aid in debugging. Linkers for virtual memory machines may contain features to allow user specification of where the routines will be placed in memory, permitting the production of linked programs with better locality properties.

The linker functions which are specifically required by the REDL language will now be discussed. First, a linker must examine the template-identifiers (i.e., the "unique names" of declarations used by more than one separately compiled program) recorded in all of the object modules being linked to make sure that there is no "version skew" among them (i.e. mismatches of essential interfaces). The design of the PROMOTE/ACCESS facility of REDL is such that the essential work of comparing declarations for version skew is done by the compiler.

Second, the linker must examine the "call graphs" of the object modules being linked and deduce from them, in effect, a "master" call graph for the entire linked program. Each call graph shows which routines might be invoked, directly or indirectly, from which other routines. The call graph analysis serves

- (1) to make sure that each routine which may be invoked recursively was declared with the RECURSIVE directive
- (2) to make sure that no "loops" exist between MONITORS; that is, that if a routine of one MONITOR may call one of another MONITOR, then no routine of the latter may call any routine of the former MONITOR. This is a REDL restriction which prevents MONITOR-induced deadlock.
- (3) to make sure that each routine which is reentrant was declared with the REENTRANT directive

Third, the linker must examine all the object machine configuration directives made in the separate object modules and make sure that they all may be met, and that the linking of the modules is permissible. Fourth, the linker must search libraries of object modules (if appropriate) to find object modules supplying items accessed by other object modules being linked. The rules for searching libraries are not specified in the REDL language. It is expected that these rules will conform to the usages of the

host operating system, its file system, and any further library facility provided (see 4.4.3 above).

Normal linker technology may be used for a REDL linker except for techniques relating to ACCESS/PROMOTE and to call graphs. As noted above, the checking for version skew is straightforward for the linker. Call graph analysis is quite similar to the call graph analysis performed by the compiler. Thus, the cost of extending these functions to the linker will not be great.

4.4.6 Symbolic Run-Time Debuggers.

The term "run-time debugger" refers to a program which is loaded with the program to be debugged, which is capable of setting break-points (or otherwise gaining control at specifiable points or times), and which provides information about the values of program variables, routine parameters, the sequence of routine invocations, and so forth.

It is worth noting that REDL provides, through its exception handling mechanism, a useful debugging aid within the language. Handlers for various errors can be provided in programs during the program check-out phase and later removed, together with the code that checks for the error, decreasing the program's run-time space and time requirements.

"Symbolic" run-time debuggers refer to debuggers capable of interactive dialog with the user at debug-time (i.e., at run-time, when the debugger is executing) which can accept symbolic requests for information about named variables and routines. A symbolic debugger thus requires the availability of a run-time symbol table and mechanisms for discovering the association between data items and their storage in the activation records of routines.

A debugger should be able to assign to, and to display at

debug-time, named variables of language-defined and user-defined types. One way to accomplish this is to permit calls on arbitrary routines in the program from the debugger: the user can then provide routines in the program being debugged which display or assign to user-defined types.

The debugger can determine the lexical environment and activation record of the user-programs' routine whose execution the debugger is interrupting; and can provide a backwards trace of the calls to that routine, showing the current values of their parameters and local variables. Where FORK statements have been executed, the debugger will identify the path presently executing the routine being examined.

Lastly, the debugger can be used to declare temporary debug-time variables of language-defined and user-defined types, and to set, modify, and examine their values by calls on user-provided routines as described above.

A symbolic run-time debugger of this form can be built as a REDL program which runs with the programs being tested on an operating system which supports REDL and an interactive terminal. Implementations for object machines lacking such an interactive capability may have to look into providing a debugger with an interpreter or simulator for REDL running on an alternative host machine.

The implementation of such a debugger must make use of a number of machine-dependent features of REDL, since many of the debugger functions are outside the normal REDL language. For example, the ability to call an arbitrary user-defined routine on the basis of its (ASCII) name implies the need not only to do a run-time symbol table lookup but also to call a routine by use of its address and perhaps to construct activation records for the routine and its dynamic descendants. The debugger must

also be able to handle any exception that may be raised by any routine so called.

4.4.7 Memory Dump Analyzer.

A memory dump analyzer takes as inputs a representation of the variable parts of the memory image of a program (with appropriate memory mapping information), information about the state of the program when it stopped, and a run-time symbol table. The analyzer is then able to display the values of all data at the time that the program stopped, associating symbolic data names with appropriately represented displays of their values.

Such an analyzer, like the debuggers described in 4.4.6, needs to allow the user to specify routines to be called to display data of user-defined types.

In the case of the dump analyzer, both the display routines and the run-time symbol table need not be part of the loaded core-image of the program. They may exist as separate entities not to be used until the memory dump analyzer is run.

The debugger of 4.4.6 and the dump analyzer of 4.4.7 share considerable functionality. Modular design of the shared portions should allow a reduction of costs if a given implementation decides to provide both of these facilities. In implementations for alternate target machines of varying size and availability (as with small, airborne computers with larger ground based counterparts), both interactive debuggers and dump analyzers may be desired.

4.4.8 Global Cross-Referencer.

The output of a global cross-reference program is a listing which shows all (or most) of the references to names and operators within a set of separately compiled programs. Each use of a name

or operator is identified by program name, page number, and line number, using the page and line number from either the compilation listing or from the source file of the program in which the reference appears. The defining scopes of the names are identified. Program level names are identified by name and by program name of origin.

Some names may be suppressed from the cross-reference, such as built-in names. Similarly, some operators, such as the built-in operators, may be suppressed.

The listing of references to data should distinguish three classes of references: read references, such as normal references within expressions and uses of data as CONST parameters to routines; write references, such as targets of assignment or as RESULT parameters to routines, and read/write references, which are the VAR parameters to routines.

References to pointers should be cross-referenced as are other data references, but dereferenced pointer references should be cross-referenced as references to the data pointed to by the pointer.

A cross-reference program is normally an option during compilation, and makes the cross-reference listing as a part of the compilation listing. A global cross-reference program is then made as a merging operating on a set of per-program cross-reference listings.

It is sometimes desired to be able to produce a global cross-reference listing from a set of source programs, name references being identified by source program page and line numbers. In this case it is necessary to build a program capable of parsing the source programs and performing sufficient semantic analysis to correctly display the declaring scope of names, the uses of data references, and to suppress unwanted items.

A cross-reference facility in the compiler makes a global cross-reference facility a simple merging operation. A cross-reference facility for the compiler, if designed with the compiler, is not a major addition.

4.4.9 Simulators and Run-Time Statistics.

4.4.9.1 Basic Properties. In many situations it will be useful to be able to test and debug programs by interpretation or simulation on a machine which is not the eventual target machine. This could be due to, for example, the small size, lack of peripherals, or unavailability of the target machine, or due to the need to test the programs with simulated real-time inputs. In addition, simulators can be useful for metering run-time statistics, such as the number of times various routines or statements are executed.

Simulation or interpretation can be accomplished in a number of ways. An interpreter may interpret source code. A simulator may interpret either target object code, an intermediate representation, or code for the simulating machine. This last possibility requires producing a compiler for the simulating machine as well as for the target machine.

Simulators may be very useful for timing studies. The execution times of linear parts of code (that is, sequences which execute a fixed and known set of instruction and/or addressing operations, without calls or branches) can be determined by the compiler for the target machine if it is provided a set of instruction timings and provided to the simulator. The simulator can then count the number of executions of the linear parts, multiply by the target-machine timings, and so obtain target machine execution timings of a high degree of accuracy via simulation. Or the simulator could be used to count the number of executions

of routines, statements, and loops, which is often useful for choosing code to be further optimized.

The complexity of a simulator thus depends on the class of statistics to be gathered and on the mechanism chosen for simulating real-time events. When simulation includes priority driven parallel processing, timings, and simulated real-time external events, the production of a simulator can become a difficult task. In comparison, the simulation of single CPU systems is much simpler.

4.4.9.2 Feasibility of Simulators. Simulators of the kind described above are within today's state of the art. Both the interpretation of source programs and the simulation of code compiled for the target machine require, however, a large amount of simulation time. For instance, evaluation of a program may take 500 to 1000 times as long as the running of the same program on the same machine. The time spent in simulation depends on the amount of simulation overhead involved. Thus, even where a compiler is built to compile directly for the simulation machine, the simulation runs may still take from two or three times longer than the real-time runs being simulated, assuming machines of equal speed.

The above discussion assumes that the simulation of the environment is carried out in hardware or software external to the simulation of the program being tested. Some simulations must be run at approximately real-time speed--for example a "man in the loop" system, in which the amount of time a person has to react to the program's output is an integral part of the system. In such cases emulators cannot be used unless they run fast enough to execute 500-1000 emulation instructions in the time it takes for the object machine to run the single instruction being emulated.

In another type of simulation, the environment is simulated within the simulation system. A single system both simulates the program and the environment. In such cases, the time to simulate obviously depends in an irreducible way on the time that must be spent in the simulation of the environment.

The choice of simulation strategy must take into account the total amount of real-time periods to be simulated over the lifetime of the simulation system, simulation/real-time timing ratios, and simulation machine costs, and then determine whether an interpreter, a target-code-emulator, a compiler for the simulation-machine, or something else is preferable.

In part, this is a trade off between the capital and maintenance costs of compilers (code generators), interpreters, or object-code emulators, etc., and the costs of using these tools over the lifetime of the simulation system.

In any case it will be more economical to design any simulation systems at the same time as the compilers, so that simulation features of the compilers can be an integral part of their design.

4.4.9.3 The Relationship of REDL to Simulation. The major difficulty in simulation arises in handling parallel processing and multiprocessing in response to real-time events. This situation characterises many embedded applications.

REDL provides a rich set of facilities related to scheduling and synchronization between parallel processes. REDL's FORK statement and priority scheme allow for the well delimited beginning and ending of paths (processes) and for their relative scheduling. REDL's PAUSE, SEND, and WAIT statements and the CONNECT directive together with REDL's monitor facility provide a number of clear points where a simulation system may usefully

provide "breakpoints." The implementation of these primitives on the simulation machine become part of the simulator, thus making the simulator independent of the object machine operating system (if any) or instruction-set provisions for scheduling, interrupt handling, etc.

4.4.10 Automated Documentation Tools.

Automated documentation tools refer to programs which generate documents of technical or management interest directly from source programs or object code produced by the compiler. Such tools might, for example:

- (1) produce a collection of program descriptions by extracting comments which immediately follow the program headers of the set of programs of a system, and might take note of additions, deletions, and changes in this set of descriptions.
- (2) produce a set of descriptions of all routines, also from comments extracted from the source text.
- (3) produce a report on the meanings of all program-level declarations from comments directly following these declarations in the source programs.
- (4) collect information on which routines call one another directly or through intermediate calls.
- (5) integrate information on the meanings of symbols with the information produced by the cross-referencer.

Documentation tools which use source program inputs are of two kinds: those which make use of comments and those which do not. A global cross-referencer (described in 4.4.8) is an example of the first kind, being driven entirely by the compilable text of the source programs. Most of the tools to be discussed below are of the second kind, making heavy use of well-formatted and commented programs.

Many of the tools described above depend on the establishment of standard formats for program commenting. Most of these tools

require a REDL parser which would identify constructs of interest and find the associated comments.

An ambitious system might establish a database to contain these reports and so allow comparison for the purpose of detecting additions, deletions, and changes to these descriptions.

A tool which documents which routines are called from which other routines could use source programs as input and could parse the source text itself, but it would be much simpler to let such a tool use the call graphs associated with object modules as input. The call graphs must be provided, in any case, if the linker is to check the validity of the RECURSIVE and REENTRANT directives across separate compilations.

By using several of the tools described above, a system could produce a cross-reference table which was complete with extensive descriptive and analytical information. In the past, information of this kind was either unavailable or was only available as secondary documentation, laboriously produced by hand. Second-order tools, such as tools to measure total program complexity, or tools to measure the possible effects on data of calls to routines which, though not themselves accessing the data might invoke routines that do, were not implementable. Technology now exists to permit the development of improved tools for development of management information.

5.0 REDL COMPLIANCE WITH DOD CRITERIA

This chapter summarizes the compliance of REDL with the criteria listed in the HOLWG's Analyses Plan for the Preliminary Designs. Except for Ironman Compatibility and Formal Definition, which are discussed in Chapter 2 and Section 4.1, the criteria are considered in this chapter in the order in which they appear in the Analyses Plan.

5.1 Military Applications

The design of REDL has taken into account the variety of ways in which it may be used. The DoD has both embedded system and general purpose computational applications. While many of the requirements of embedded systems are the same as those for general purpose applications, embedded systems have some special needs. REDL has been designed to meet these special needs as well as to satisfy general purpose requirements.

Some concerns of embedded systems which are common to most applications are those of program clarity, self-documentation, and maintainability. Those features of REDL which promote these general goals are described elsewhere in this chapter. The structure of REDL encourages the modular, top-down development of programs. REDL requires as part of this program development process, the complete specification of interfaces between the routines within the program. These are major concerns in the development of embedded systems.

The majority of coding for an embedded or general purpose application can be satisfied by a machine-independent High-Order Language. REDL provides a powerful set of machine independent features to satisfy this need. Embedded systems, however, operate in an environment where machine dependencies are inevitable. They must respond to real-time sensors and drive real-time devices. These sensors and devices often require a precise, machine-dependent format to specify the information which flows between them and the central processor.

Thus REDL's machine-dependent features allow the specification of data records in any format, and the complete bit-by-bit specification of the fields if desired. Once defined, these machine-dependent records may be manipulated by the normal operations within the language. Because REDL provides bitstrings and a full set of common bit-manipulation primitives, operations upon machine-depend records can be defined within the HOL.

In the programming of embedded systems it is occasionally necessary to go outside the HOL. REDL allows transitions into and out of machine code, with no overhead, and with the machine code clearly partitioned from the rest of the program.

Embedded systems also require a full range of real-time and parallel processing facilities: for instance, the ability to respond to a real-time or simulated clock; to respond simultaneously to a number of real-time sensors; or to drive a number of real-time displays simultaneously. These input and output devices must be able to guide the control flow of the program's execution asynchronously. REDL's parallel processing and synchronization mechanisms were designed with this capability in mind.

A project may be required to preserve some programs previously written in another language while adding new programs written in REDL. REDL programs may be interfaced with programs in other languages. If a REDL program is to call a routine in another language, the routine may be declared with the EXTERNAL directive, which then allows the user to specify the call interface which the foreign routine expects.

Examples of REDL programs for a variety of embedded applications appear in the Informal Language Specification, Appendix A.

5.2 Readable, Modifiable, Maintainable Programs.

Readability, modifiability and maintainability were primary goals in the design of REDL. Some examples of design decisions which help to meet these goals are the following.

5.2.1 Control Structure.

The control statements in REDL were designed with the "one-in one-out" philosophy; that is, there should be a single entry point and a single exit point. In REDL, this philosophy applies not only to the basic control statements such as IF and WHILE, but also to multipath (FORK) and exception handling (ON) statements.

5.2.2 Namescopes.

As pointed out in the discussion of Ironman 2F, the restrictions on overriding declarations were in specific response to the goals of readability and maintainability. The descriptive power given by the ability to redeclare, say, the built-in type BOOLEAN was more than outweighed by the confusion such a redeclaration would cause to a reader of the program.

5.2.3 Pointers.

The explicit specification of CONST or VAR in the declaration of a pointer type reveals the programmer's intended use of the type.

5.2.4 Separate Compilation.

REDL's approach to separate compilation allows a variety of program structuring conventions, an important consideration in the development of maintainable systems.

5.2.5 Constructors.

The constructor forms for records, unions, and machine-records require the explicit specification of field names, aiding readability.

5.2.6 Capsules.

The data encapsulation facility contributes heavily toward maintainability, since it allows the programmer to hide representation-dependent details (which are susceptible to change).

5.3 Rigid Standards.

REDL has been designed so that the rigid standards, necessary in the military environment, can be established and enforced for the language's definition and the certification of its translations.

One requirement of the language design was to produce a language of sufficient simplicity that there would be no need for subsetting it. To accomplish this, the number of basic statement types was minimized, and only a small number of language-defined data types was provided. This was done without limiting the power and usefulness of the language, because easy-to-use facilities exist to let users define their own data types and routines.

Moreover, in REDL there is no difference in the way that built-in routines and user-defined routines are handled. Because of this, REDL need only define a minimum number of built-in routines with additional routines provided by the user as needed. Consequently, each group of users can tailor a programming environment to its needs, while the language retains a minimal cost of implementation.

Another way of maintaining rigid standards is to ensure that, in all implementations, a program will keep its exact meaning. REDL accomplishes this by providing a complete description of a program's behavior and by minimizing and encapsulating implementation dependencies.

5.4 Efficient Execution.

REDL incorporates many features to make fast execution of programs possible, and to make efficient use of storage. A partial list follows.

The array mechanism permits, through the existence of parameterized bounds, the passage of arrays of different sizes to the same routine. This is very useful for the construction of general array manipulation routines, and is a flexibility which is absent, for example, in Pascal. However, this added power does not introduce any additional cost when it is not used; i.e., there is no need for array dope information for arrays which have fixed bounds. This is an example of how flexibility and power were added to the language so that any additional cost is incurred only when the feature is used.

The definitions of procedures and functions were developed in a way which allows the compiler to make optimizing decisions in the way it translates them, to maximize efficiency. For instance, the compiler may translate a routine call either as a branch to closed code or by inline substitution of the routine body without affecting the semantics, and is therefore free to make the choice on the basis of efficiency. Similarly, an input parameter to a called routine (a CONST parameter) may, in many cases, be realized either by copying or by passing a "pointer", at the compiler's discretion.

There are inevitably some cases in which hand-coded machine code will be more efficient than compiled code. When this situation occurs, REDL permits the programmer to write an inline machine dependent routine with a machine code body. In this way a needed speed improvement may be obtained without reverting to assembly language for the entire program.

The combination of extensive compile-time evaluation of constant expressions and conditional compilation enables a large range of operations which are traditionally done at run-time to be performed at compile-time.

The variable which represents the return value of a function is named in the header of the function, rather than in a RETURN statement. This means that there is only one return value variable for any function. The compiler can therefore use the return value slot on the stack for that variable throughout the execution of the called function. In languages which have return statements of the form

RETURN expression;

the expression must be copied into the return value slot at the time of the return. REDL avoids this extra copy, since the return variable never exists in separate storage, but only in the return value slot itself.

Another feature which improves efficiency is that the size of the return value must be known at compile time. Thus a variety of highly efficient calling protocols may be used, depending on the architecture of the target machine.

REDL's compiler directives enable the programmer to choose which generated code tradeoffs should be considered the most important by the compiler. It is possible to specify, on any namescope, whether to optimize storage requirements or execution time. Within a program, the user may also specify in an individual declaration whether access time or storage space should be optimized, through the use of the grouping attribute.

Other compiler directives allow further optimizations to be made. A routine which is used either recursively or reentrantly must be explicitly denoted as such by the presence of the appropriate compiler directive. This is checked by the compiler or linker. Thus efficiencies in the case of nonrecursive or nonreentrant routines may be realized without sacrificing safety.

The INLINE directive lets the programmer recommend that calls to a routine be expanded inline when this will likely improve efficiency.

Since it is impossible in a non-machine-dependent program to access the storage of a variable in any way except by the explicit use of its name, a variety of optimizations are therefore possible.

The compiler can, with knowledge of the control flow of the program, determine the value of a variable at compile-time if its last value came from a constant, and not have to generate code to look up the value. Alternatively, the compiler can determine that it may use the value of a second variable instead, if access to it is cheaper. It can also determine (at compile-time) that a variable is unnecessary because its value is always represented by other variables or constants, and eliminate all use of the variable in the code, not even producing storage for the superfluous variable. Another optimization which can be made because there is only one possible access path to a named variable is that two variables may be given the same storage, if the compiler can ascertain that their lifetimes are disjoint.

Compile-time facilities can be used to maximize the amount of processing done at compile-time and to reduce the amount left for run-time. They can also preserve the efficiency of inline code while obtaining the readability advantages of having a single instance of a particular code sequence. The compile-time procedure facility in REDL provides these features to an extend beyond what can be done with **INLINE** routines.

The way that a **SELECT** statement discriminates among alternative tag fields was designed to maximize run-time efficiency without loss of safety.

The parallel path facility of the language is an efficient way of realizing multitasking capabilities. Although it is powerful and flexible enough to model any multitasking algorithm, in most cases the number of paths is known at compile-time, allowing the compiler to make storage provisions for the paths when it provides storage for the rest of the program. The compiler may also set up the control blocks for the paths at compile-time, thereby reducing the run time overhead of multitasking. This would be impossible if the paths were dynamically created.

5.5 Simplicity.

It is the interaction of different aspects of a language which most often give rise to unnecessary complexities. It is therefore difficult to point to specific aspects of a language as examples of simplicity. Some design decisions which had particular impact in reducing the number of underlying concepts in the language are discussed below.

The rules for the type mechanism in REDL were kept simple. All types in REDL are opaque. Any two distinct type names represent two distinct types. This reduces the number of underlying concepts, and makes type identity extremely simple: two objects are of the same type if and only if their type names are the same.

To keep REDL simple, the introduction of new constructs to the language was generally avoided when the capability provided by the construct could be obtained through the use of features already in the language. For example, the treatment of a monitor as a capsule avoids the necessity for specialized syntax and semantics.

An objective of the design of REDL was to have the language rules apply with consistency. For example, the semantics of CONST and VAR are the same for data declarations, formal parameters, and "pointed to" types. As another example, the rules for the bracketing of syntactic units (in particular, the conventions pertaining to the end delimiter) are consistent, independent of whether the construct is a declaration or a statement.

5.6 Reliable, Provable Programs.

These goals emerge in many of the Ironman requirements and thus are met in REDL by virtue of REDL's compliance with Ironman. The following examples point out cases where REDL has gone beyond the Ironman in attempting to meet these goals.

(1) Side Effects. REDL does not permit the Ironman-allowed side effect of a function which modifies a variable "own" to an encapsulation.

(2) Aliasing. Some kinds of aliasing are regarded as an error by REDL but not by Ironman; for example, passing an object as a VAR and RESULT parameter. The reason REDL prohibits this is that such a practice is highly susceptible to error: any change made to the VAR parameter during execution of the procedure will be overridden by the copy out to the RESULT parameter.

5.7 Ease of Teaching and Learning.

To a large extent these goals overlap with simplicity (Section 5.5). A difference is that ease of teaching and learning are dependent on the quality of language documentation, whereas simplicity is dependent on the language itself. The Informal Language Specification provides a basis upon which a judgment of REDL's learnability can be made.

5.8 Machine Independence.

REDL has been designed so that machine dependencies are minimized and encapsulated. As with many of the other language goals, machine independence is captured in the Ironman requirements and is thus achieved through REDL's compliance with Ironman. An instance is the requirement that each arithmetic variable be declared with a specific range constraint on its value space, and that floating-point variables be declared with specific decimal precisions.

REDL defines the interface between the program and the run-time environment in a machine-independent (and operating-system-independent) fashion. This is true, for example, for exception handling, input/output, and parallel processing. A set of exceptions is defined within the language, and whether the exception is detected by a trap on the machine or by additional compiled code in the program does not change the program's behavior. The I/O mechanism is defined so that it is free from any effects of the particular environment in which the program is running. The behavior of any program using the parallel processing facilities is completely defined by the language, so that under any operating system the program will behave in the same way.

A feature of REDL showing the machine independence of parallel processing programs is the PRIORITY directive. This feature allows the programmer to completely specify the interaction of parallel paths in their contention for a shared resource, whether it be data or the CPU itself. In this way the programmer can stipulate the way the program will behave, independent of the operating system's scheduling algorithm. The program will thus behave the same on any implementation.

The ability to specify machine dependent properties in a CONFIGURATION directive is another aid to transportability. This compile-time facility allows two sets of machine-dependent portions of the program to be present, one set for the large host and one set for the target machine to be selected by interrogating machine-dependent properties. The program can be debugged on the large host and transported, changing only the CONFIGURATION directive, to the target machine, at which point only the machine-dependent portion, which should be very short, need be tested.

5.9 Practical, Implementable.

Implementability was a key concern during the design of REDL, and an attempt was made in the Informal Language Specification to provide a level of design detail sufficient to judge the degree to which this goal is met. Some specific examples follows:

(1) Side Effects and Aliasing. Unless care is taken, the rules for prohibiting side effects and dangerous aliasing can demand sophisticated implementation techniques. In REDL this problem was avoided. The requirement for use of the SAFE directive makes the checking for side effects in functions straightforward (in fact, there is no need to trace through arbitrary call graphs; an investigation of "directly reachable" procedures is sufficient). With respect to aliasing, the prohibition against pervasive variables eases the task of the implementation.

(2) Type Declarations. The requirement that each type have a distinct name greatly eases the task of determining type identity. Moreover, it facilitates the work of an optimizing compiler, since if a type T is declared as an ARRAY (1..10) of T1, then each object of type T is guaranteed to contain 10 components and there is thus never any need to store "dope vector" information with objects of type T.

5.10 Addresses the Difficult Issues.

Although not intended as a complete description of REDL, the Informal Language Specification was designed to confront the "difficult issues" implicit in the Ironman. This section presents a brief description of some of these issues.

Parallel processing presents some difficult problems. What is the program behavior when an unhandled exception is raised in a path? What is the meaningful lifetime of a path? What happens when there is an attempt to terminate a path which no longer exists? What happens to data "locked" by a path which terminated? What paths may legitimately terminate what other paths? What is the initial state of an event? What is the precise meaning of sending events or waiting for events? How is data accessed which is shared between parallel paths? How is deadlock prevented? All of these questions are resolved in the sections on multi-path facilities.

The area of real-time control raises some additional issues. It shows the need in the language for machine-dependent records, for a capability allowing response to real-time interrupts, for scheduling real-time responses, and for run-time determined amounts of storage to be allocated. REDL addresses these requirements of real-time control.

A comprehensive exception-handling mechanism requires a language, among other things, to provide a definition of exceptions which does not change among target machines; to deal with what happens when an exception is raised while processing another exception; to provide the ability to enable or disable the detection of exceptions; to define what storage may be accessed by the exception handler itself; to define the semantics of return from a handler; to define what happens to routine parameters when the routine is exited directly from an exception handler. REDL has expressly addressed all these issues, among others raised by exception handling.

Among the other difficult areas which are documented in the Informal Language Specification are those relating to configuration specification, conditional compilation, input-output facilities, generic definitions, library interfaces, encapsulations, dynamic types, side effects, aliasing.

Interactions between features have been considered in the design of REDL and documented in the Informal Language Specification. Examples are exception handling and multipath facilities; overloading, namescope rules, and separate compilation; pointers and side effects in functions; the PERVASIVE directive and aliasing.

APPENDIX A: RECURSION AND DISPLAYS

A.1 Background

Ironman 7B requires that the language provide for recursive routines and requires that restrictions be placed so that an execution-time display (or equivalent mechanism) is not needed. The term "display" refers to a vector of stack pointers which duplicate the lexical chain; if a variable is allocated at offset p from the stackframe corresponding to lexical level n , then its address is given by $\text{DISPLAY}(n)+p$. In the absence of an explicit display, this address can only (in general) be determined by tracing back through the lexical links k times, where $k = \text{current level} - n$. Further details can be found in [RR64,pp65ff].

One approach to satisfying 7B, illustrated by the language C [RKL75], is to prohibit routines from importing non-local data objects, unless such objects are declared at program level. The reason a display is unnecessary (even in the presence of recursion) is that references to local data are given by offsets from a stack pointer, and references to program-level data are given by actual addresses. Although this approach is sufficient, it is possible for less stringent restrictions to be imposed without requiring the use of a display. The following sections outline the solution adopted by REDL and show how it might be implemented.

A.2 Avoidance of Display in the Non-Recursive Case

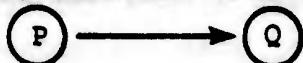
Let us first consider the situation if there were no recursion. Is it possible to get by with no display? There are two problems: first, length-unresolved arrays defer knowledge of storage requirements until run-time; and second, the same routine can be called from different lexical levels, implying that its data will not always be in the same place

relative to the origin of the entire stack.

The first problem can be handled by treating a length-unresolved array as a fixed amount of storage (possible dope information) including a pointer to the actual data housed in the heap or on an auxiliary stack.

The second problem can be handled by inducing a partial order (representable as a directed acyclic graph) on name-scopes in the following fashion:

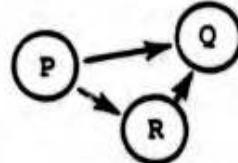
an arc is directed from P to Q



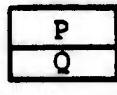
if and only if P contains in its immediate body an invocation of the routine (or block) Q. For example, consider:

```
PROCEDURE P;  
    PROCEDURE Q; ... END PROCEDURE Q;  
    PROCEDURE R; ... CALL Q; ... END PROCEDURE R;  
    CALL Q;  
    CALL R;  
END PROCEDURE P;
```

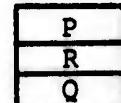
We represent the call structure as:



Under ordinary stack conventions, the possible dynamic environments are:

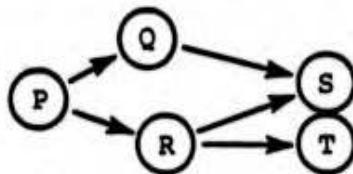


and



Thus, if Q contains variable declarations and routine declarations which reference such declarations, then there is no way for such a reference to be resolvable without a pointer back to Q's origin, since Q's data will be at different positions depending on which call on Q is active. However, under the assumption of no recursion (and thus

no cycles in the graph), it is always possible to lay out "stack" frames at compile time in accordance with the maximum requirements at any one time. In the above example, the compiler would always allocate Q's data after the storage required for R, even though R might not be active. Since Q can't reference R's data, there is no danger to doing this; the only sacrifice is that more storage might be reserved at a given time than is actually needed. As another example, from the following directed acyclic graph:



the compiler could lay out run-time storage as follows:

P
max of Q,R
max of S,T

The point of this discussion is that, in the absence of recursion, it is possible for the compiler to organize storage in such a way that displays (or lexical links) are unnecessary; i.e., each data reference can be compiled into an actual address (possibly an offset with respect to an index register holding the absolute address of the origin of data storage for the entire program).

The above technique is applicable in the presence of multiple paths, but we point out that the time savings in avoiding a display is at the expense of data space expansion. The storage for the parameters and local data of any reentrant routine reachable from a path must be allocated with the storage for the path's data.

The next section shows how to extend the technique described above to the case of recursive routines. (In view of the response to Ironman Clarification Question 66, we do not attempt to account for the cases where recursion

interacts with multiple paths.)

A.3 Accounting for Recursion

What happens if we allow recursion? To simplify the arguments, let us assume that no routine may be defined within a recursive routine (this was the restriction posited in the DoD's "Tinman" requirements, which predated the Ironman). There are thus only two issues which arise concerning data references:

(1) References from non-recursive routines (perhaps from separate compilations) invoked in recursive routines; e.g.,:

```
VAR X: ...  
PROCEDURE P; ... X ... END PROCEDURE P;  
PROCEDURE Q; ... IF cond THEN CALL Q; ELSE CALL P; END IF;...  
END PROCEDURE Q;  
<* NON-RECURSIVE P IS CALLED FROM RECURSIVE Q *>  
  
CALL Q;  
..
```

But resolving the reference to X in P is no problem; the data storage layout for the main program and P is simply:

main program (including X)
P

and the address for X is the same regardless of the depth of recursion on Q.

(2) References within recursive routines. There are two kinds of such references:

(a) References to parameters or to data declared within the recursive routine. To resolve such a reference,

we require a dynamic pointer to the recursive routine's stackframe, since there may be several incarnations of the data.

(b) References to imported data (i.e., data declared in an enclosing scope). Since, by our assumption at the start of this section, the enclosing scope cannot itself be (or be contained within) a recursive routine, it follows that the data of the scope can be laid out in accordance with the principles of Section A.2 above. Thus any reference to an imported data object can be resolved to an actual address.

The implication of (1), (2), and the techniques described in Section A.2, is that the prohibition against defining routines within recursive routines is sufficient to render the display unnecessary. (We point out again that this was exactly the claim made in the Tinman.) Actually, we need not make the restriction so severe; all that is necessary is that if a routine is recursive, then no routine defined inside it can import data (parameters, constants, variables) local to the recursive routine. In fact, even this can be relaxed. If the inner routine is non-recursive, its data storage can be laid out "statically" and it can import data from the recursive routine and access it via an offset from the dynamic stack pointer for the recursive routine's stack frame. It is only when the inner routine is itself recursive that we must prohibit import of the outer recursive routine's data, for in this situation the dynamic stack pointer will reference the stack frame from the inner routine; we would have to trace a lexical link to determine the stack frame in which the outer routine's data appear. To simplify the statement of the restriction, REDL adopts the following: prohibit the definition of recursive routines inside other recursive routines. (This is slightly more strict than necessary,

but considerably simpler than a specification of the exact circumstances under which a display is unnecessary.)

A.4 Summary of Recommendations

- (1) Allow routines to import data declared in surrounding scopes.
- (2) Prohibit the definition of recursive routines inside other recursive routines.
- (3) The implementation technique which avoids the display (or lexical links) is to partition storage into a fixed size region reflecting the maximal requirements for the non-recursive routines (length-unresolved arrays being represented by "pointers" to the actual arrays), and a stack used for the data of recursive routines.

BIBLIOGRAPHY

- [AF76] Air Force Systems Command, Rome Air Development Center. JOVIAL J73/I Specification. July 1976.
- [Bri72] Brinch Hansen, P., "Structured Multiprogramming," Comm. ACM, Vol. 15, No. 7, July 1972.
- [DH72] Dahl, O.J. and Hoare, C.A.R. "Hierarchical Program Structures" in Structured Programming, Academic Press, 1972.
- [Di68] Dijkstra, E.W., "Cooperating Sequential Processes," in Genuys, F. (ed.), Programming Languages, Academic Press, 1968.
- [FW77] Fisher, D.A. and Wetherall, P.R. Institute for Defense Analyses. "Fixed Point and Floating Point Requirements" (Draft), Memorandum for HOL Design Contractors, Dec. 5, 1977.
- [Hoa69] Hoare, C.A.R., "An Axiomatic Basis for Computer Programming," Comm. ACM, Vol. 12, No. 10, Oct. 1969, pp. 576-583.
- [Hoa72] Hoare, C.A.R. "Notes on Data Structuring," in Structured Programming, Academic Press, 1972.
- [HW73] Hoare, C.A.R., and Wirth, N. "An Axiomatic Definition of the Programming Language PASCAL," Acta Informatica, Vol. 2, pp. 335-355, 1973.
- [IHRC74] Ichbiah, J.D., Heliard, J.C., Rissen, J.P., and Cousot, P. The System Implementation Language LIS, Technical Report 4549 E/En, CII, December 1974.
- [Int74] Intermetrics, Inc., HAL/S Language Specification, November 1974.
- [Int75] Intermetrics, Inc., CS-4 Language Reference Manual and Operating System Interface. October 1975.
- [JW76] Jensen, K., and Wirth, N., PASCAL User Manual and Report, 2nd ed., Springer-Verlag, New York, 1976.
- [Kn68] Knuth, D.E., "Semantics of Context-Free Languages," Math. Systems Theory, Vol. 2, No. 2, June 1968, pp. 127-145.
- [Kos76] Kosinski, M., SPL/I Language Reference Manual, Intermetrics Inc., IR-115-2, July 1976.
- [La70] Lalonde, W.R., An Efficient LALR Parser Generator, Master's Thesis, Univ. of Toronto, 1970.

- [LHLMP77] Lampson, B.W., Horning, J.J., London, R.L., Mitchell, J.G., and Popek, G.L., Report on the Programming Language Euclid. SIGPLAN Notices, Vol. 12, No. 2, February 1977.
- [Nau60] Naur, P. et al. "Revised Report on the Algorithmic Language ALGOL 60", CACM, March 1960.
- [Pal76] Palme, J. "New Feature for Module Protection in SIMULA", SIGPLAN Notices, May 1976.
- [Pe71] Peck, J.E.L. (ed.), ALGOL 68 Implementation, North Holland Publishing Company, Amsterdam, 1971.
- [RKL75] Ritchie, D.M., Kerrigan, B.W., and Lesk, M.E., "The C Programming Language," Computing Science Technical Report #31, Bell Laboratories, Murray Hill, N.J., 1975.
- [RR64] Randell, B. and Russell, L.J., ALGOL 60 Implementation. Academic Press, 1964.
- [SA75] Snyder, A., and Atkinson, R., Preliminary CLU Reference Manual, CLU Design Note 39, M.I.T., January 1975.
- [Wi75] Wirth, N., "An Assessment of the Programming Language Pascal," IEEE Trans. on Software Engineering, Vol. SE-1, No. 2, June 1975, pp. 192-198.
- [Wi76] Wirth, N., "MODULA: A Language for Modular Multiprogramming," Institute fur Informatik, Report No. 18, Eidgenosische Technische Hochschule, Zurich, 1976.
- [Wi77] Wirth, N. "Toward a Discipline of Real-Time Programming," Comm. ACM. Vol. 20, No. 8, August 1977.
- [X75] American National Standards Technical Committee X3J1 and European Computer Manufacturers Association Technical Committee TC10. Draft Proposed American National Standard Programming Language PL/I: BASIS/1-12, Document BSR X3.53; February 1975.

INFORMAL LANGUAGE SPECIFICATION

**CDRL Item 0002AC
Contract MDA903-77-C-0330**

TABLE OF CONTENTS

	page
1.0 INTRODUCTION	1-1
1.1 Purpose of Document	1-1
1.2 Notation for Structural Description	1-2
1.2.1 Lexical Diagrams	1-2
1.2.2 Syntax Diagrams	1-4
1.3 Overview of Document	1-6
1.4 Terminology	1-7
2.0 LANGUAGE SUMMARY	2-1
2.1 Program Structure and Scopes	2-1
2.2 Data Types	2-1
2.3 Data Encapsulation	2-2
2.4 Routines and Expressions	2-2
2.5 Statements	2-3
2.6 Input/Output	2-3
2.7 Exception Handling	2-4
2.8 Multi-Path and Real-Time Facilities	2-4
2.9 Machine-Dependent Facilities	2-4
2.10 Compile-Time Facilities	2-5
3.0 LEXICAL STRUCTURE	3-1
3.1 Character Set	3-1
3.2 Lexical Units	3-2
3.3 Token	3-3
3.3.1 Name	3-3
3.3.2 Syntactic Trigger	3-4
3.3.3 Special Enumeration Symbol	3-6
3.3.4 Literal	3-6
3.3.5 Special Symbol	3-10
3.3.6 Compile-Time Symbol	3-10
3.4 Token Separator	3-11

	page
4.0 PROGRAM STRUCTURE AND SCOPES	4-1
4.1 Program Structure	4-1
4.1.1 Directives	4-3
4.1.2 Declarations	4-5
4.1.3 Compile-Time Commands	4-7
4.1.4 Statements	4-8
4.1.5 Terminology	4-9
4.2 Namescopes	4-11
4.2.1 Introduction	4-11
4.2.2 Pervasive Directive	4-13
4.2.3 Import Directive	4-15
4.2.5 Readonly Directive	4-19
4.2.6 Reuse of Names	4-20
4.2.7 Example	4-21
4.2.8 Forward References	4-22
4.2.9 Optimize Directive	4-23
5.0 DATA TYPES	5-1
5.1 Basic Concepts	5-1
5.1.1 Attributes and Representation	5-1
5.1.2 Assignment and Parameter Passing	5-3
5.1.3 Type Naming	5-4
5.1.4 Storage Classes	5-5
5.1.5 Comparison with Pascal	5-6
5.2 The Declaration of Constants and Variables	5-8
5.2.1 The Concepts of "Constant" and "Variable"	5-8
5.2.2 Initialization	5-9
5.3 The Declaration of Types	5-10
5.3.1 Examples	5-11
5.3.2 Simple and Parameterized Type Declarations	5-12
5.3.3 Type Identity	5-12

	page
5.4 Scalar Types	5-13
5.4.1 BOOLEAN	5-13
5.4.2 CHAR	5-15
5.4.3 FIXED	5-17
5.4.4 FLOAT	5-24
5.5 Type Generators	5-27
5.5.1 Enumeration	5-27
5.5.2 Array	5-30
5.5.3 Record	5-38
5.5.4 Union	5-41
5.5.5 Pointer	5-45
5.6 Parameterized Types	5-50
5.6.1 The Declaration of Parameterized Types	5-50
5.6.2 The Creation of Objects Having Parameterized Types	5-51
5.6.3 Unresolved Type References	5-52
5.6.4 Assignment and Parameter Passing	5-53
5.6.5 STRING	5-55
5.6.6 BITS	5-56
5.7 The Declaration of Ranges	5-58
5.8 The Grouping Attribute	5-59
5.9 Type Opacity, "Lifting" and "Lowering"	5-61
6.0 CAPSULES	6-1
6.1 Capsule Declarations	6-1
6.2 Export Directive	6-3
6.3 The Lifetime of Capsule Data	6-4
6.4 Capsule Example	6-5
7.0 ROUTINES	7-1
7.1 Introduction	7-1

	page
7.1.1 Comparison with Pascal	7-2
7.1.2 Overloading Routines	7-2
7.1.3 Invocation and Return	7-3
7.1.4 Formal Parameters and Actual Parameters	7-4
7.1.5 Routine Directives	7-6
7.1.6 Inline Directive	7-7
7.1.7 Reachability	7-8
7.1.8 Recursion	7-9
7.1.9 Reentrancy	7-10
7.2 Procedures	7-11
7.2.1 Safe Directive	7-12
7.2.2 Aliasing	7-14
7.2.3 Main Directive	7-16
7.3 Functions	7-17
7.3.1 Function Result Variable	7-18
7.3.2 Prevention of Side Effects	7-18
7.4 Operators	7-19
7.4.1 Contrast between Functions and Operators	7-20
7.4.2 Commutative Directive	7-21
7.4.3 Associative Directive	7-22
8.0 EXPRESSIONS	8-1
8.1 Introduction	8-1
8.2 Comparison with Pascal	8-1
8.3 The Parse of an Expression	8-2
8.4 Order of Evaluation	8-5
8.5 Diagrams for Expression Syntax	8-6
8.6 Primary	8-9
8.6.1 Object Component	8-11
8.6.2 Object Constructor	8-12
8.6.3 Function Invocation	8-13

	page
9.0 BASIC STATEMENTS	9-1
9.1 Comparison with Pascal	9-2
9.2 Empty Statement	9-3
9.3 Assignment Statement	9-3
9.4 Begin Statement	9-5
9.5 Conditional Statement	9-6
9.5.1 If Statement	9-7
9.5.2 Select Statement	9-9
9.6 Loop Statements	9-13
9.6.1 While Statement	9-14
9.6.2 Repeat Statement	9-15
9.6.3 For Statement	9-16
9.7 Control Transfer Statements	9-19
9.7.1 Call Statement	9-20
9.7.2 Exit Statement	9-21
9.7.3 Goto Statement	9-22
9.8 Assert Statement	9-23
10.0 INPUT/OUTPUT	10-1
10.1 Basic Properties	10-1
10.2 Files	10-2
10.3 File-Record I/O Routines	10-4
10.3.1 OPEN and CLOSE (Procedures)	10-4
10.3.2 READ (Procedure)	10-5
10.3.3 WRITE (Procedure)	10-5
10.3.4 EOF (Function)	10-5
10.3.5 REWIND (Procedure)	10-5
10.3.6 GET_POSITION (Procedure)	10-5
10.3.7 SET_POSITION (Procedure)	10-6
10.3.8 OVERWRITE (Procedure)	10-6

	page
10.4 Textfile I/O	10-7
10.4.1 The TEXT_FILE Type	10-7
10.4.2 Line-Oriented I/O Routines	10-7
10.4.3 The Standard Files INPUT and OUTPUT	10-8
10.4.4 Formatted I/O	10-8
10.5 Library Definition Facilities	10-10
11.0 EXCEPTION HANDLING FACILITIES	11-1
11.1 Introduction	11-1
11.2 Exception Declarations	11-2
11.3 Exception Statement	11-3
11.3.1 Raising of Exceptions: The Raise Statement	11-3
11.3.2 Handling of Exceptions: The On Statement	11-4
11.4 Suppression of Exception Checking: The Off Directive	11-7
11.5 Determination of Raised Exception	11-8
12.0 MULTI-PATH AND REAL-TIME FACILITIES	12-1
12.1 Introduction	12-1
12.2 Paths	12-2
12.2.1 Fork Statement	12-3
12.2.2 Execution of a Path Clause	12-4
12.2.3 Path States	12-5
12.2.4 Priorities of Paths	12-7
12.3 Monitor Capsules	12-9
12.4 Events	12-11
12.4.1 Initialization	12-12
12.4.2 Send Statement	12-12
12.4.3 Wait Statement	12-13
12.4.4 SENDERs and WAITERs	12-14

	page
12.5 Real-Time Facilities	12-15
12.5.1 Pause Statement	12-15
12.5.2 Cumulative Processing Time	12-15
13.0 MACHINE-DEPENDENT FACILITIES	13-1
13.1 Introduction	13-1
13.2 Configuration Directive	13-3
13.3 Data-Oriented Machine Dependencies	13-4
13.3.1 Machine Record	13-4
13.3.2 Absolute Addresses and Alignment	13-7
13.3.3 The Connect Directive	13-10
13.4 Routine-Oriented Machine Dependencies	13-11
13.4.1 Machine Routines	13-11
13.4.2 External Routines	13-14
14.0 COMPILE-TIME FACILITIES	14-1
14.1 Separate Compilation	14-1
14.1.1 Promote Directive	14-1
14.1.2 Access Directive	14-2
14.1.3 Linking Separately-Compiled Programs	14-3
14.1.4 Example	14-4
14.2 Compile-Time Constants and Expressions	14-6
14.2.1 Compile-Time Constant Declarations	14-6
14.2.2 Language-Defined Compile-Time Constants	14-7
14.2.3 Compile-Time Expressions	14-8
14.3 Conditional Compilation	14-9
14.4 Compile-Time Procedures	14-11
14.5 Type-Unresolved Parameters	14-17
14.5.1 ANY	14-17
14.5.2 Caller Assertions	14-18
14.5.3 Type Comparison Functions	14-19
14.5.4 Redeclare Directive	14-20

	page
APPENDIX A: REDL EXAMPLES	A-1
A.1 I/O for Particular Machines	A-1
A.1.1 Illustrates a channel-I/O program for the AN/UYK-20	A-1
A.1.2 Illustrates an Interrupt-driven I/O Program for the PDP-11	A-3
A.2 Methods for Construction of Standard I/O Library Procedures	A-6
A.3 Parallel Processing Examples	A-8
A.3.1 Illustrates use of MONITORS for Synchronization in a standard parallel processing problem	A-8
A.3.2 Illustrates Parallel Processing Where One Job is a Real-time Job Driven at 100-millisecond Intervals	A-10
A.4 Examples from Graphics and Communications Applications	A-12
A.4.1 Graphics Example	A-12
A.4.2 Communications Software Example	A-15
APPENDIX B: ASCII CHARACTERS	B-1
B.1 Character Classes	B-1
B.2 CHAR Literals	B-2
APPENDIX C: DIAGRAM CROSS-REFERENCE	C-1
C.1 Lexical Diagrams	C-2
C.2 Syntax Diagrams	C-3
C.3 Lexical Diagrams (Cross-Reference)	C-7
C.4 Syntax Diagrams (Cross-Reference)	C-8
APPENDIX D: LALR(1) GRAMMAR FOR REDL	D-1

1.0 INTRODUCTION

1.1 Purpose of Document.

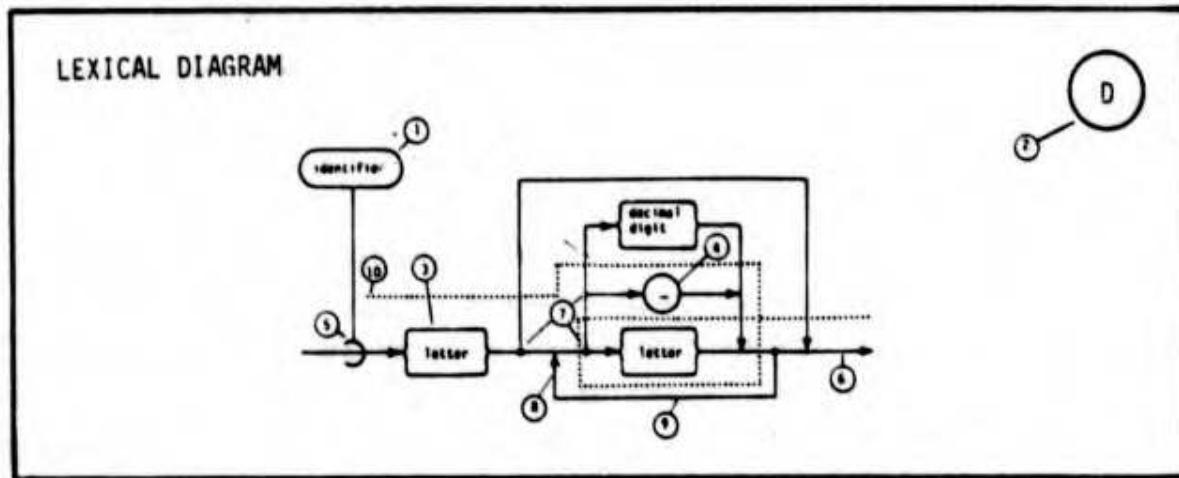
The purpose of this document is to provide an informal specification of REDL, a language based on Pascal and designed in accordance with the Department of Defense's Revised Ironman requirements (July 1977). The name "REDL", short for "Red Language", is derived from the color code established for this document by DoD.

The syntax of REDL is presented formally, through the use of syntax diagrams; the semantics of the language is described informally, in English. The informality of the semantic specification is intentional, since a complete and formal definition of the language was not required (indeed, would not have been possible) for this phase of the contract. The description in this document is intended to be sufficiently complete that the reader can judge the compliance of REDL with the Ironman and with the other evaluation criteria established by the DoD's High Order Language Working Group. To facilitate the description, we frequently employ programming examples and comparisons with Pascal. Moreover, if it is useful in describing a feature to explain why that feature is present (e.g., a specific requirement in Ironman), then we will on occasion supply motivational remarks.

1.2 Notation for Structural Description.

REDL, like Pascal, uses diagrams to express the lexical and syntactic structure of the language. Each lexical diagram specifies the structure of one or more lexical units. Each syntax diagram specifies the structure of one or more syntactic categories. By tracing a path through a diagram, the reader can produce an instance of the lexical unit or syntactic category represented by that diagram.

1.2.1 Lexical Diagrams.



Each diagram defines the set of legitimate instances for a particular kind of lexical unit. The name of the lexical unit being defined appears in the rounded box (1) at the upper left of the diagram.

The diagram index, the letter shown at (2), is associated with this specific diagram. Appendix C contains a cross-referenced index of diagram indices.

Boxes with braced sides, such as (3), represent character classes as defined in Appendix B.

Boxes with rounded sides, such as ④, represent single characters, sequences of contiguous characters, or lexical units. Above each rounded box representing a lexical unit is a cross-reference to the definition of that lexical unit.

To generate instances of the lexical unit, the reader follows the diagram from left to right, from box to box, starting at the point of junction of the definition box ⑤, and ending when the end of the path ⑥ is reached.

If a diagram does not fit horizontally, it is broken into segments indicated by three dots at the end of one line, and three dots at the beginning of the next. (This is not illustrated in the lexical diagram above.)

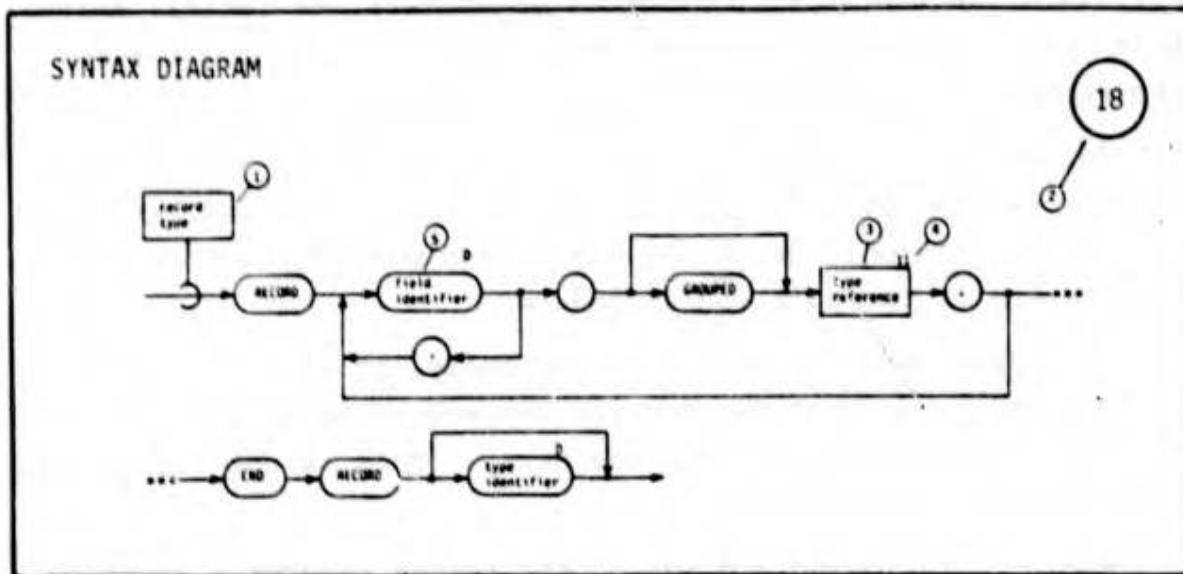
When, in following a diagram, the reader reaches a junction point ⑦, any of the paths leaving the point may be taken.

It is not legal to "back up" along a convergent path ⑧.

Potentially infinite loops such as ⑨ may sometimes be encountered.

Every time a box is encountered, a character it represents is added rightmost to the sequence of characters generated by moving along the path. For example, moving along the path parallel to the dotted line ⑩ generates the sequence "letter underscore letter" (e.g., A_B or Q_Q).

1.2.2 Syntax Diagrams.



The rules for syntax diagrams are similar to those for lexical diagrams, with the following changes.

The name of the syntactic category being defined appears in a rectangular box (1). The illustrated example defines the <record type> syntactic category.

As the preceding paragraph illustrates, the name of a syntactic category is enclosed in angle brackets when it is referenced in the text.

The diagram index for syntax diagrams, shown at (2), consists of integer values.

Rectangular boxes such as (3) represent syntactic categories defined in other syntax diagrams. The number above the box (4) is the diagram index associated with that box.

Boxes with rounded sides, such as (5), represent single characters, sequences of contiguous characters, or lexical units known as tokens (see Section 3.2). For ease of exposition, the

tokens "name" and "identifier" are frequently prefixed by a descriptive term reflecting the usage of those tokens in the context of the syntax diagram. In the diagram shown above, the terms "field identifier" and "type identifier" both refer to the token called "identifier".

Following a path through a syntax diagram produces instances of the category. Each instance is a sequence of tokens. Between any two tokens in the sequence, a token separator (see Section 3.3.7) may appear. If the juxtaposition of two tokens in the sequence would produce a sequence of characters which make up a longer token, then a token separator must appear between these tokens. In the diagram shown above, a token separator must appear between END and RECORD, and may optionally appear between the field identifier and the comma or semicolon.

An example of an instance of the syntactic category <record type> is the following:

```
RECORD REC EXAMPLE:  
  XFLAG: BOOLEAN;  
  S: STRING(20);  
END RECORD
```

1.3 Overview of Document.

This document is divided into fourteen chapters (organized to correspond roughly to Ironman), four appendices, and an index. Chapter 2 contains a summary of the language. Chapter 3 treats REDL's lexical properties. Chapter 4 describes the structure of REDL programs and the namescoping conventions. Chapter 5 discusses data objects and data types, and Chapter 6 presents REDL's data encapsulation facility. Chapter 7 describes routines (i.e., procedures, functions, and operators). Chapter 8 presents the rules for forming expressions. Chapter 9 treats the basic statements provided by REDL. Chapter 10 discusses I/O, Chapter 11 treats exception handling, and Chapter 12 describes the multi-path and real-time facilities. Chapter 13 presents the machine-dependent features, and Chapter 14 treats the compile-time facilities.

Appendix A contains a set of practical programming examples drawn from a variety of embedded applications. The programs are intended both to illustrate various features in REDL, and to demonstrate REDL's applicability to embedded systems. Appendix B contains information relevant to REDL's lexical properties. Appendix C comprises alphabetized and cross-referenced lists of the lexical and syntax diagrams. Appendix D contains a "BNF" syntax for REDL, equivalent to the syntax diagrams but suitable for direct use in a table-driven parser. (Formally, the syntax is LALR(1), implying that "bottom-up" parsing can be employed with one symbol look-ahead and no backup.)

1.4 Terminology.

In this document, the terms "must" and "may not" occasionally appear. Both of these terms are ambiguous in English. For example, to state "Condition C must be satisfied" could imply either that condition C is a tautology (and thus that it is impossible not to satisfy C), or that it is incumbent upon an agent to see to it that condition C is met (i.e., that it will be an error if the condition is not satisfied). Unless otherwise specified by the context, the latter interpretation will be assumed. (The "agent" is the programmer; the error will be detected by the compiler unless otherwise noted.)

There is a similar ambiguity in the term "may not". To state "Condition C may not be satisfied" could imply either that it is possible for C not to be met, or that it is imperative (with an error arising otherwise) for an agent to guarantee that C is not met. Again, the second interpretation is the one adopted in this document.

2.0 LANGUAGE SUMMARY

The material in this chapter provides an overview of the basic features of REDL.

2.1 Program Structure and Scopes.

A REDL program consists of a sequence of declarations (which associate names with program elements) and directives (which provide the translator with supplementary information). The actions to be performed are embodied in statements, which appear in a program through their containment in the declarations of routines (procedures, functions, and operators).

REDL is a block-structured language. In contrast with Pascal, a name declared in one scope is not automatically known in all inner scopes. If such behavior is desired, the name must be made pervasive. Otherwise, external names will only be known in open scopes and in those scopes into which they are explicitly imported.

2.2 Data Types.

REDL provides a rich set of built-in types and type definition facilities. The built-in types are BOOLEAN, CHAR, arithmetic types FIXED and FLOAT, character and bit string types STRING and BITS, and the machine-dependent type ADDRESS. Type generators are provided to permit the definition of enumeration, array, record, machine-record, union, pointer, and file types. Array, record, and union types may be parameterized, enabling routines to be written taking arguments whose sizes differ from one invocation to another. Thus, different objects of the same parameterized type may have a different number of components. The built-in types STRING and BITS are parameterized.

The rules for type identity in REDL are simple. Each type must have a name, and types are opaque in that types with different names are distinct types.

Each type in REDL, whether built-in or user-defined, possesses a set of attributes established by its declaration. For example, PRECISION is an attribute of the FLOAT type. Different objects of the same type may possess different values for an attribute of the type. The representation of a data object, established when the object is created, is the set of values possessed by the object for each of its type's attributes. The rules for assignment and parameter passage in REDL make use of the above concepts.

2.3 Data Encapsulation.

The unit of data encapsulation in REDL is the capsule, a namescope mechanism which allows the user to specify a protective boundary around the elements declared within. A capsule is a closed scope, implying that externally declared names must either be explicitly imported or be pervasive in order to be referenced inside. An element declared in a capsule must be explicitly exported if it is to be referenced outside. Thus the user may export selectively those capsule elements which represent an intended abstraction, and may hide those elements that are considered as details of implementation.

2.4 Routines and Expressions.

Procedures, functions, and operators are referred to as routines. Routines may take formal parameters, with three kinds of binding classes: CONST ("input"), VAR ("by reference"), and RESULT ("copy out"). Routines may be overloaded; i.e., the same name may be associated with several routine declarations, with the number and/or types of parameters determining the appropriate choice for any invocation.

Functions and operators may not have side effects. REDL imposes restrictions on the binding classes (only CONST is permitted) and on the use of pointers and external data and routines from within a function or operator.

Procedures are prohibited from performing aliasing of the kind which interferes with program reliability and formal semantic specification. For example, it is illegal to pass as a VAR parameter to a procedure a data object which is referenced via some other identifier inside the procedure.

REDL provides a set of built-in operators which can be extended by the user through overloading. There is no facility for defining new operator symbols or for performing other kinds of syntax extension. The syntax for expressions in REDL uses conventional precedence and associativity rules for the various operators.

2.5 Statements.

REDL contains a basic repertoire of statements including assignment, the begin statement (corresponding to the "block" of block-structured languages), conditional statements, loop statements, control-transfer statements, and the assert statement. The select statement, one of the conditional statements, provides a safe and efficient means for discriminating upon the "tag" of a union object.

2.6 Input/Output.

REDL provides the file type generator and a set of library routines for general application-level I/O, and a set of definitional facilities so that low-level and/or specialized I/O routines can be defined within the language. The library routines are based on Pascal, but are defined so that they return an encoding of the success or failure of the routine in an explicit "status" parameter. The definitional facilities include a set of compile-time features for type interrogation.

2.7 Exception Handling.

REDL's exception handling facility allows the user to provide explicit control over program behavior in the event of error situations. Exceptions may be either pre-defined or user-declared. The raising of an exception results in a search for a handler; REDL's on-statement provides a mechanism whereby the user can specify a handler. A directive is supplied for suppressing compiler-generated checking code for pre-defined exceptions, for use in those environments where efficiency considerations outweigh the benefits of the checking.

2.8 Multi-Path and Real-Time Facilities.

REDL contains a facility for defining parallel control paths (the fork statement), for allowing paths to cooperate in mutually exclusive accesses to shared data (the monitor capsule), for permitting paths to synchronize their execution, either time-independently (via events) or time-dependently (via the pause statement); and for dealing with failures on any control path (the X_TERMINATE exception).

2.9 Machine-Dependent Facilities.

REDL's machine-dependent facilities are encapsulated in that a program intending to use any of them must make this intention explicit in a configuration directive. The machine dependencies supported by REDL include a type generator (machine-record) which can be used to define the storage layout for data objects, an ADDRESS type, a LOC function, directives for aligning or storing data and for connecting programmer-supplied procedures to asynchronous interrupts, and facilities for defining assembly-language routines or for interfacing with routines written in other languages.

2.10 Compile-Time Facilities.

The compile-time facilities of REDL include features for separate compilation (the promote and access directives) which permit general strategies for program decomposition without sacrifice of compile-time checking; a facility for declaring compile-time constants and using them in compile-time-evaluable expressions; a feature for conditional compilation and for "semantic macros" (compile-time procedures); and facilities for declaring and using routine parameters whose types are "unresolved".

3.0 LEXICAL STRUCTURE

3.1 Character Set.

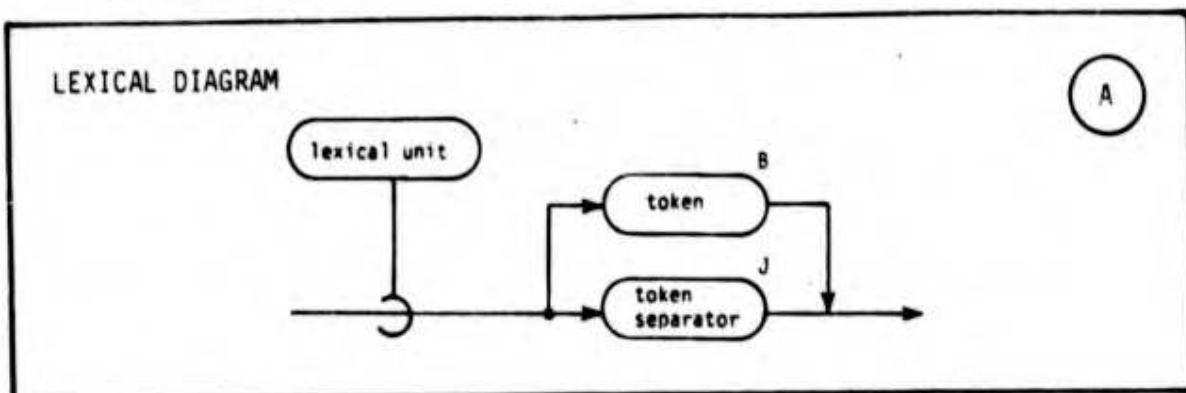
The program text consists of a sequence of characters taken from an implementation-defined character set. REDL assumes that this set includes the digits 0 through 9, the upper case letters A through Z, the space character, and the following symbols:

! # \$ % & ' () * + , - . / : ; < = > ? ^ _ @ []

We note that each of the above characters is in the 64-character ASCII subset (see Ironman 2A); the two symbols " and \ from this subset are not required by any REDL construct.

Although REDL does not require the presence of any character outside of the sixty-two specified above, such characters are permitted in contexts such as comments and string literals. As a result, for the purposes of this chapter it is useful to assume a specific character set in the definition of REDL's lexical properties. We have chosen full (i.e., 128-character) ASCII. Although the use of characters outside the 64-character ASCII subset may prevent a source <program> from being directly transported to another installation, we point out that routines which convert from one character set to another can be written in REDL. Thus, as an example, an installation using the EBCDIC character set can provide an ASCII-to-EBCDIC conversion routine so that <program>s using full ASCII can be transported.

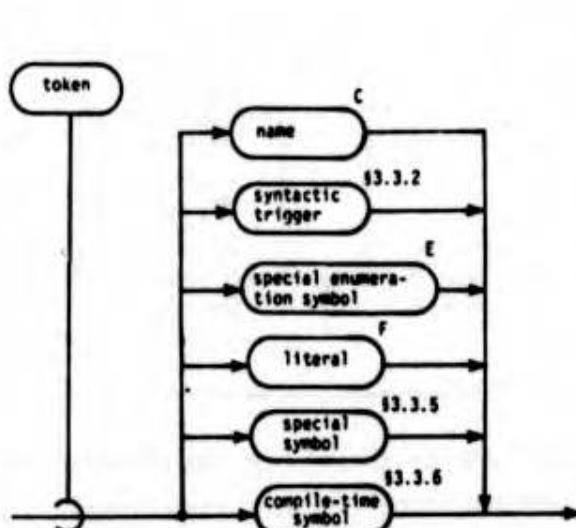
3.2 Lexical Units.



The sequence of characters comprising the program text is partitioned into lexical units which are either tokens or token-separators. The syntactic and semantic correctness of a <program> is based on the sequence of tokens which it contains; token separators serve as documentation or (as their name indicates) as lexical separators between contiguous tokens.

3.3 Token.

LEXICAL DIAGRAM

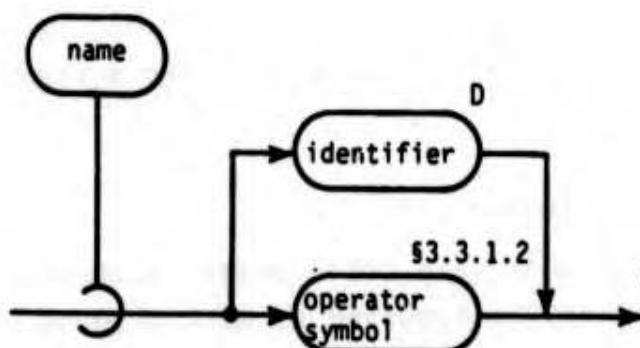


B

3.3.1 Name.

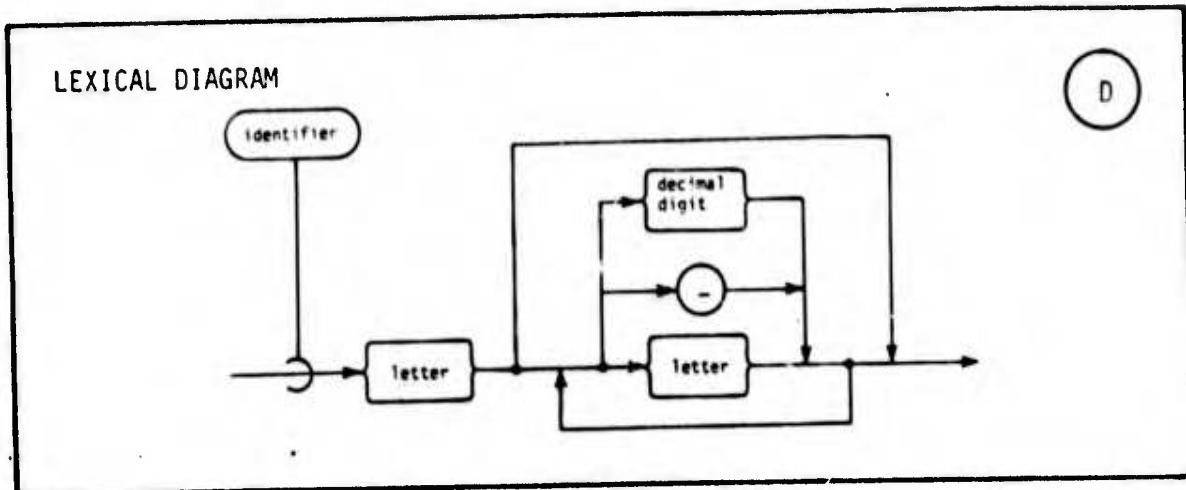
LEXICAL DIAGRAM

C



D

3.3.1.1 Identifier.



Identifiers are names denoting declared program elements (e.g., constants, variables, types, routines). The underscore is provided as a "break" character. If the character set includes lower case letters, these are not distinguished from upper case. Thus ABC and abc are regarded as the same identifier.

3.3.1.2 Operator Symbol. An operator symbol is a token from the following set:

+ - * / ** CAT DIV MOD AND OR XOR NOT & : # < <= > =

Although the operator symbols CAT, DIV, MOD, AND, OR, XOR, and NOT have the form of identifiers, they may only be used in a <program> in the contexts established by the Syntax Diagrams where they appear.

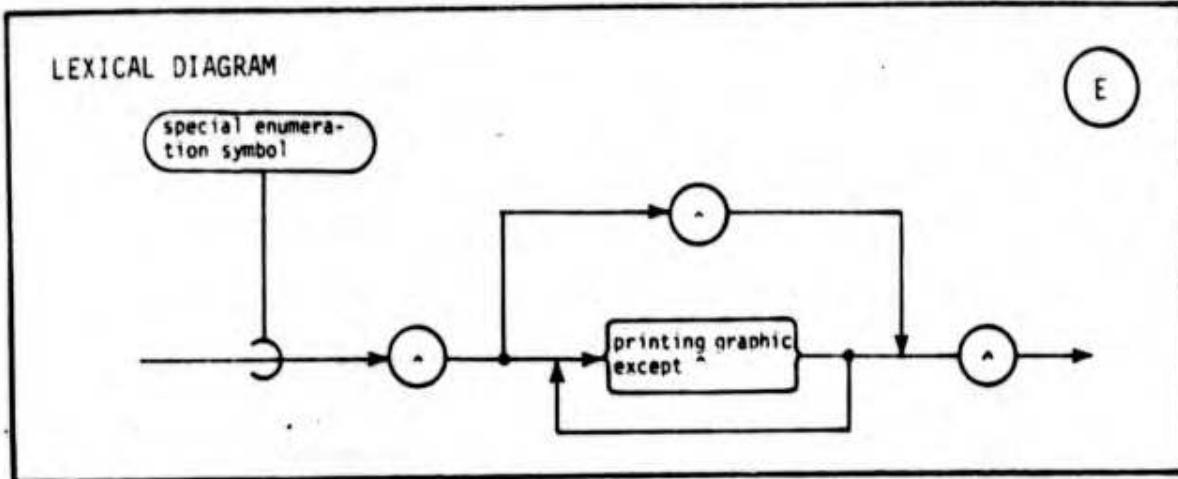
3.3.2 Syntactic Trigger.

A syntactic trigger is a token which, although having the form of an identifier, is reserved by REDL because of its syntactic role. Table 3-1 enumerates the syntactic triggers in REDL.

ARRAY	MRECORD
ASSERT	OF
AT	ON
BEGIN	OPERATOR
CALL	ORDERED
CAPSULE	OTHERWISE
CASE	OVERLOAD
CONST	PATH
DO	PAUSE
DOWNTO	POINTER
ELSE	PROCEDURE
ELSEIF	PROGRAM
END	RAISE
ENUM	RANGE
EVENT	RECORD
EXCEPTION	REPEAT
EXIT	RESULT
FILE	SELECT
FOR	SEND
FORK	SYSTEM
FROM	THEN
FUNCTION	TYPE
GOTO	UNION
IF	UNTIL
IN	UPTO
INIT	VAR
MACHINE_CODE	WAIT
MONITOR	WHILE

Table 3-1. Syntactic Triggers in REDL.

3.3.3 Special Enumeration Symbol.

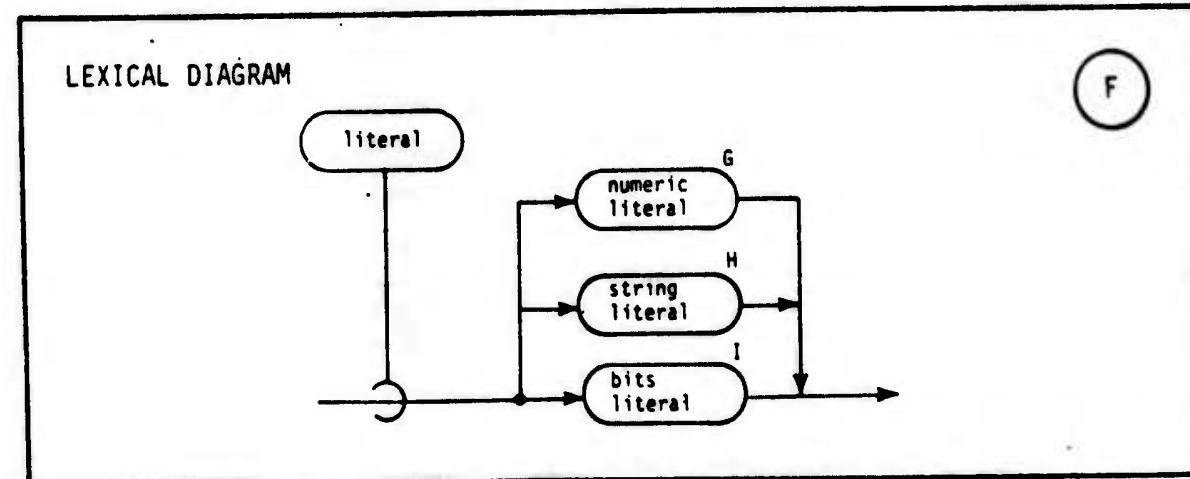


As will be seen in Section 5.5.1, an element of an enumeration type may be either an identifier or a special enumeration symbol. The latter form allows convenient representations of character sets as enumeration types.

Examples:

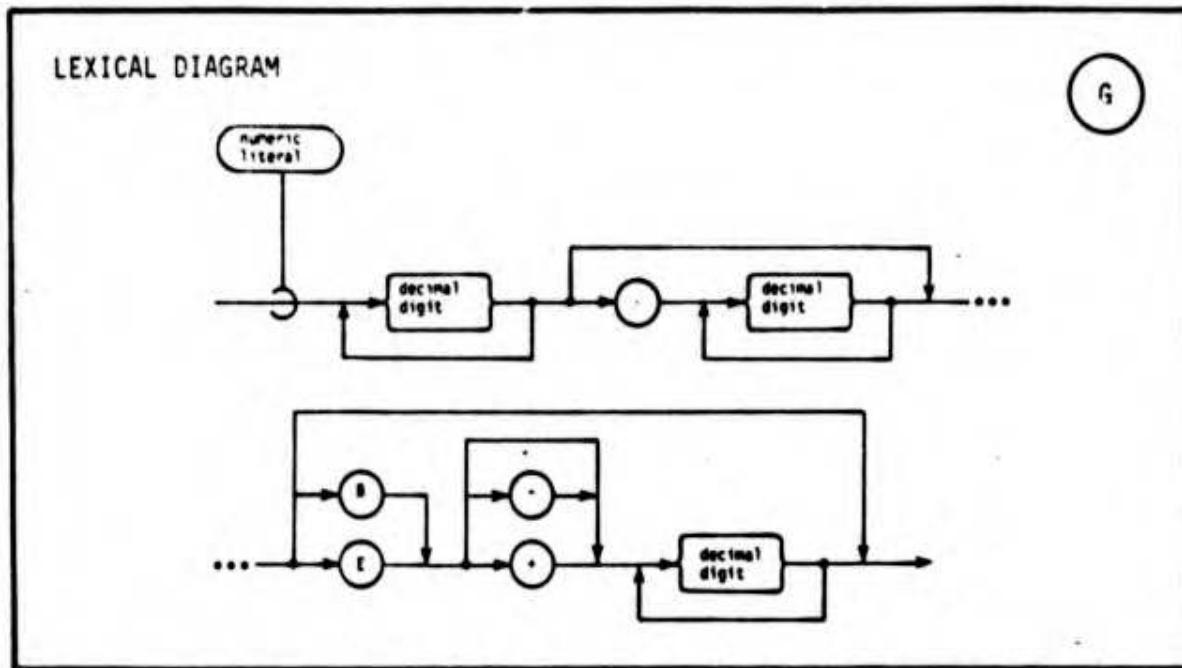
^CR^ ^\$^ ^^^

3.3.4 Literal.



Literals are tokens which represent constant values.

3.3.4.1 Numeric Literal.



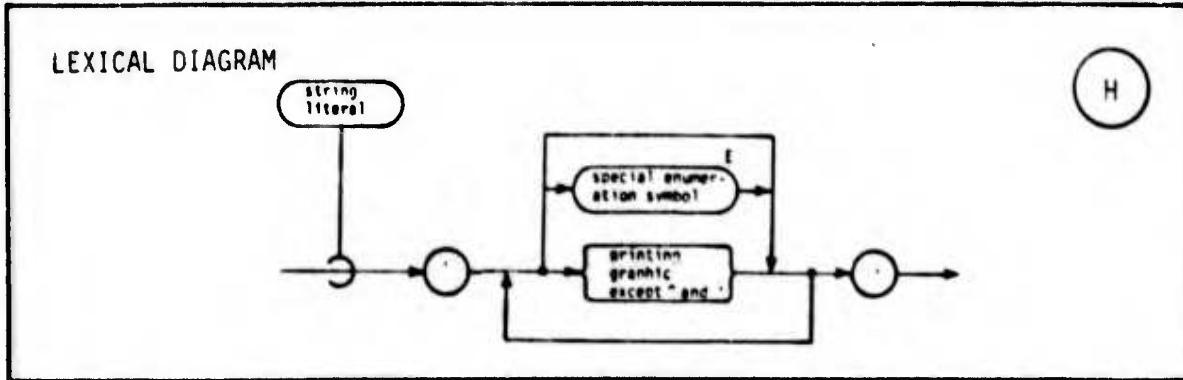
Numeric literals represent values of type FIXED or FLOAT.

Examples:

0 123 0.3E-5 5B7 6E-1

The interpretation of numeric literals is given in Sections 5.4.3.2 and 5.4.4.2.

3.3.4.2 String Literal.



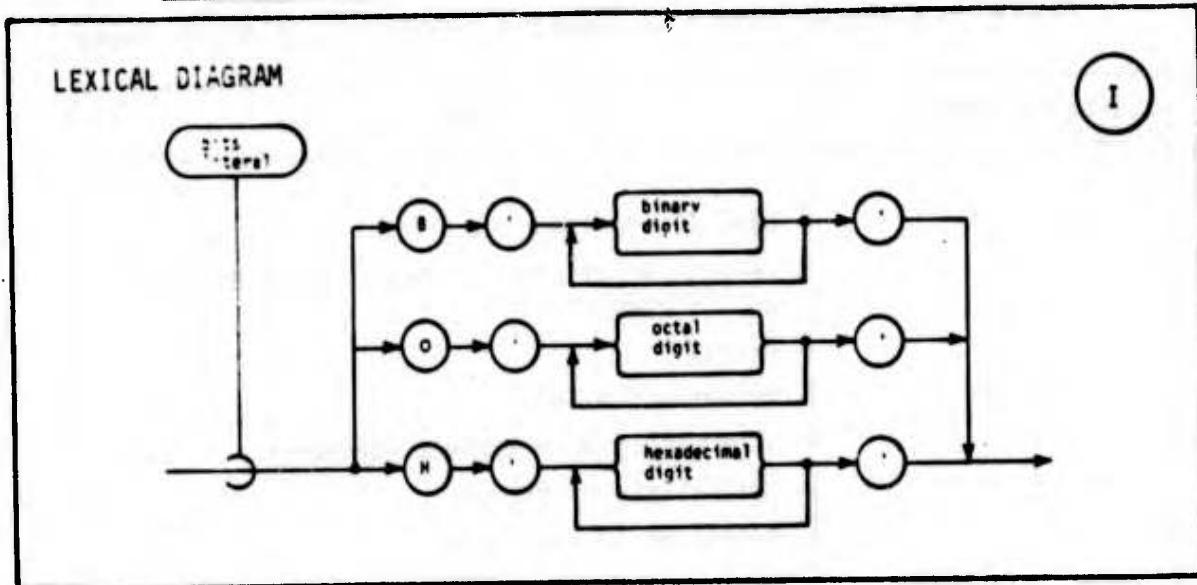
String literals represent values of type STRING.

Examples:

'ABC' '' 'DON^'^T GO NEAR THE WATER' 'IJKL^CR^^LF^'

The interpretation of string literals is given in Section
5.6.5.2.

3.3.4.3 Bits Literal.



Bits literals represent values of type BITS.

Examples:

B'01001' O'7760' H'FF'

The interpretation of bits literals is given in Section
5.6.6.2.

3.3.5 Special Symbol.

Table 3-2 enumerates the special symbols and their uses.

SPECIAL SYMBOL	USE
,	General separator in lists
;	Terminator of <directive>s, <declaration>s, and <statement>s
..	Ranges
(and)	Parenthesization
=>	Alternative of a <select statement>
?	Initialization Phrase
!	Prefix of <directive>s
:	Separator in <declaration>s
:=	Assignment
.	Record qualification
\$	Union selection
@	<Pointer dereference>
{ and }	Array Construction

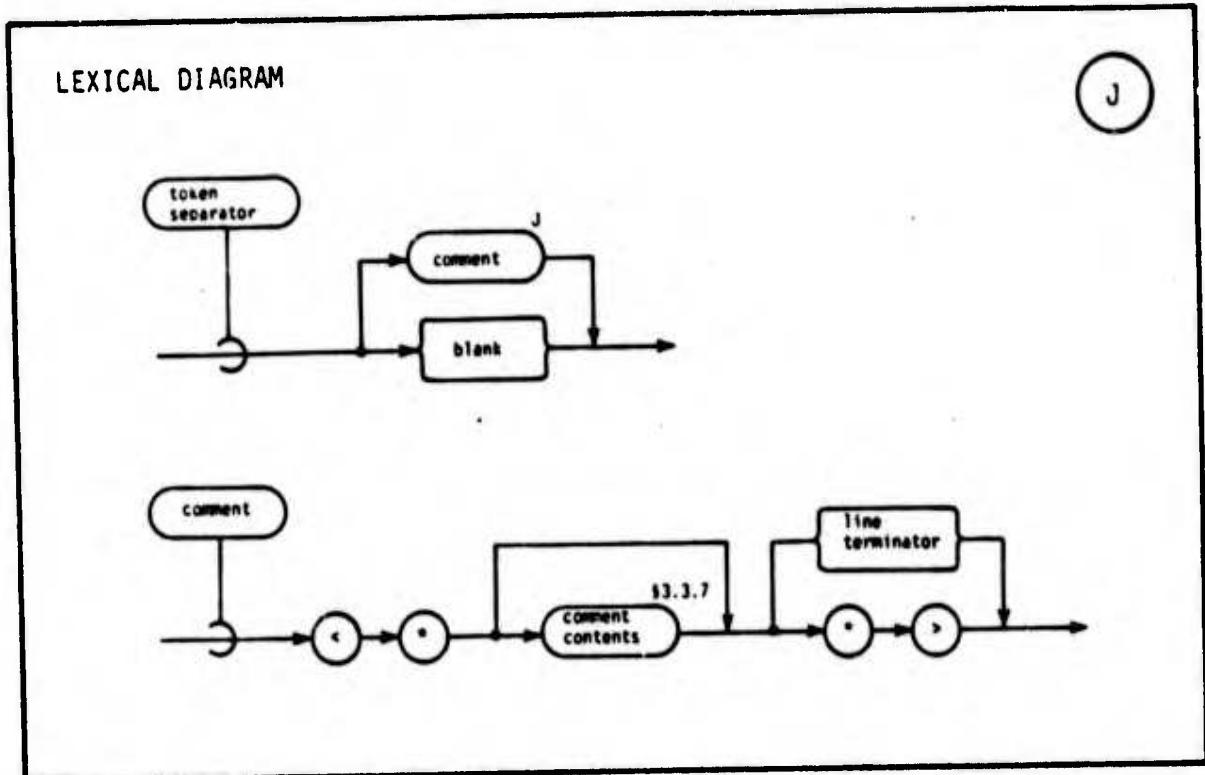
TABLE 3-2: Special Symbols and Their Uses

3.3.6 Compile-Time Symbol.

The compile-time symbols in REDL are the following:

%IF %THEN %ELSEIF %ELSE %CONST %PROCEDURE %DECLARE %END %CALL

3.4 Token Separator.

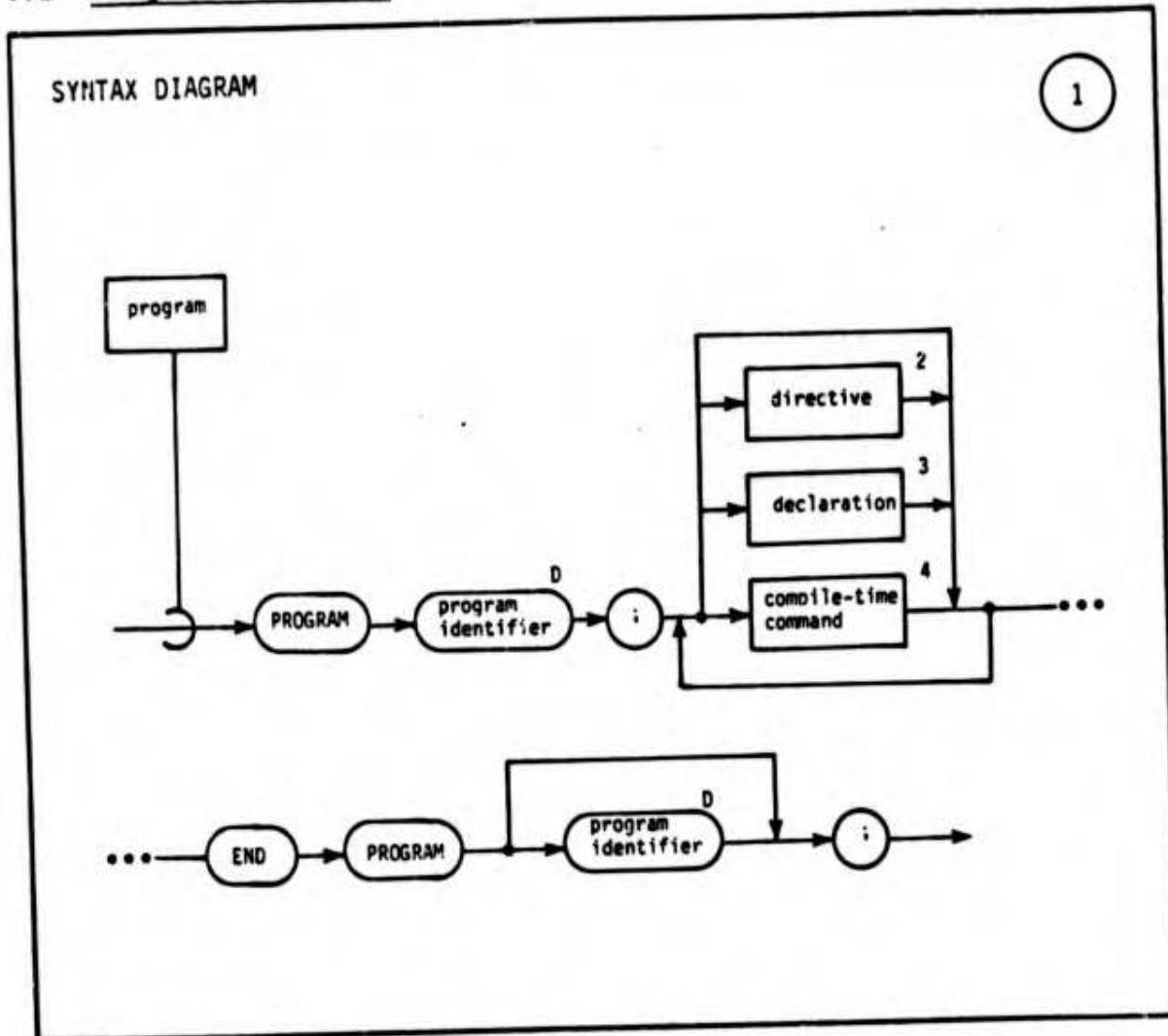


The comment contents indicated in the above diagram may be any sequence of characters except for a line terminator and consecutive *> symbols. Thus each comment begins with <* and is terminated by the first occurrence of either *> or a line terminator character. Sample comments are as follows:

```
<*THIS IS A COMMENT*>
<*THIS IS A COMMENT WITH END-OF-LINE AS TERMINATOR
<*COMMENTS MAY HAVE EMBEDDED *, <, AND *>
```

4.0 PROGRAM STRUCTURE AND SCOPES

4.1 Program Structure.



The unit of compilation in REDL is the <program>. <Program>s may be separately compiled, and program elements declared within one <program> may be accessed in another. Separate compilation facilities are defined in Section 14.1.

The identifier following PROGRAM, if present, must be the same as the identifier following END PROGRAM. This identifier

may be referenced in another <program> which intends to access elements declared in the identified <program>.

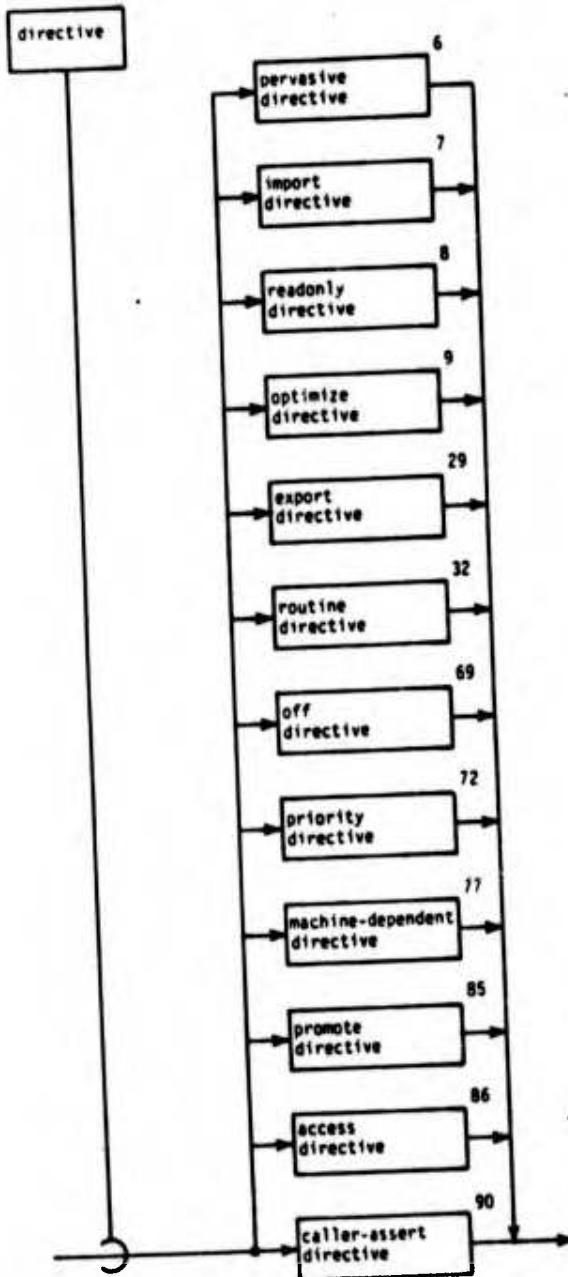
The program structure for REDL differs from Pascal in that <directive>s are allowed, <declaration>s may occur in any order, there is no statement part, and separate compilation is allowed.

A <program> may contain <directive>s, <declaration>s, and <compile-time command>s. Informally, <declaration>s associate names with program elements, and <directive>s supply guidelines to the translator. <Compile-time command>s, when they occur at program level, produce further <directive>s and <declaration>s. The run-time actions of the <program> are performed by <statement>s, which appear as constituents of <routine declaration>s.

4.1.1 Directives.

2

SYNTAX DIAGRAM



A <directive> is a portion of the <program> which provides the translator with information supplementing that contained in the <declaration>s. Syntactically, each <directive> begins with an exclamation mark and ends with a semicolon. Semantically, <directive>s are used to reveal optimization criteria, to elucidate machine dependencies, to indicate main routines, and to control accesses to program elements. The individual <directive>s will be described in the relevant sections later in this document. The <directive>s presented in this chapter may appear in any namescope.

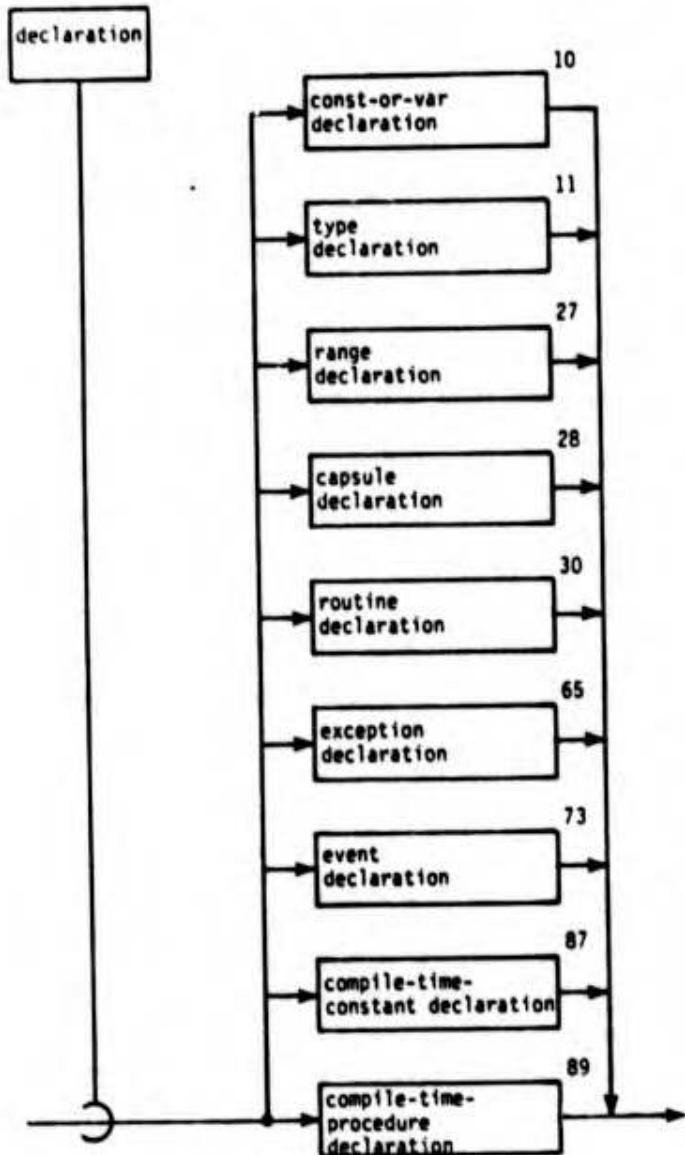
Examples:

```
!PERVASIVE T,X,Y;  <*See Section 4.2.2*>
!IMPORT A,B,C;      <*See Section 4.2.3*>
```

4.1.2 Declarations.

SYNTAX DIAGRAM

3



A <declaration> has the effect of associating a name with a program element. In addition to the program elements specified in Syntax Diagram 3, there are several which are declared in other contexts: labels, formal parameters, function result variables, for-statement variables and path identifiers. Hereafter we shall use the term "declaration" (with no angle brackets) to include name/program-element associations occurring either via a <declaration> or in one of these other contexts.

There are no "default" declarations in REDL. Each constant and variable identifier must be declared explicitly, as in Pascal.

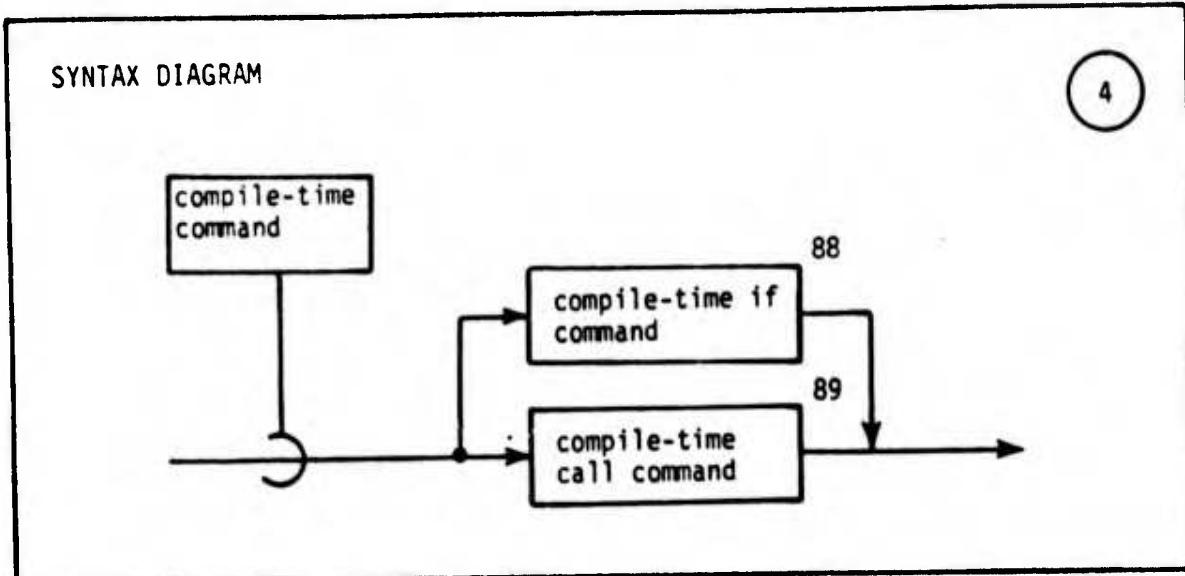
Many declarations contain an end delimiter followed by an optional identifier (capsules, routines, and compile-time procedures have this property). If the user codes an identifier with the end delimiter of such declarations, then it must be the same as the identifier of the program element being declared.

Declarations are considered to be processed sequentially within a scope. Thus, the declaration of a type T may be immediately followed by the declaration of a variable with type T, a variable V may be referenced in initialization <expression>s of declarations following the declaration of V, and the bound of a formal array parameter may be used in the declaration of subsequent formal parameters or local routine variables.

Example:

```
BEGIN
  VAR N: FIXED(1..100) INIT 10;
    <*N is an integer between 1 and 100, inclusive*>
  TYPE T: ARRAY(1..N) OF BOOLEAN;
    <*Each datum of type T will have 10 components*>
  VAR X: T INIT ?;
    <*X is an uninitialized variable of type T*>
    :
END;
```

4.1.3 Compile-Time Commands.

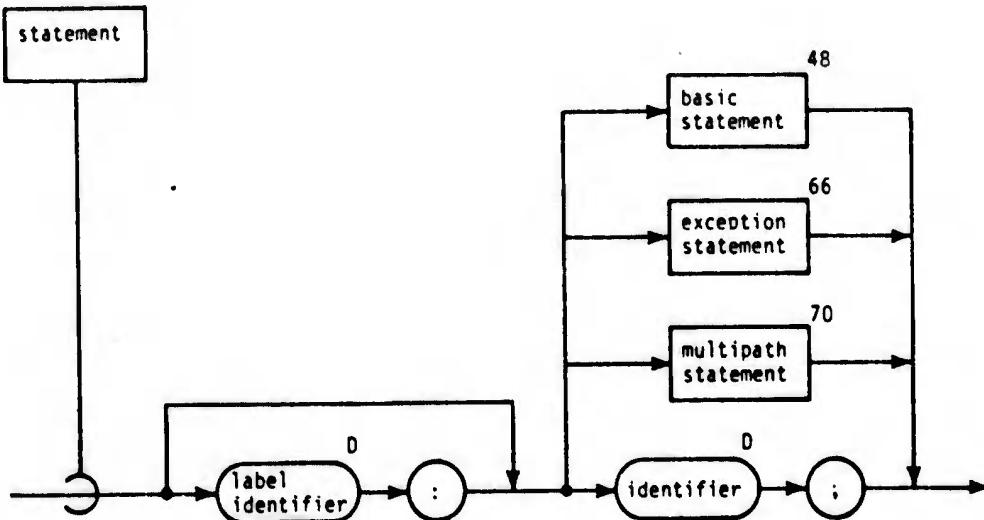


<Compile-time command>s allow for conditional compilation and for the invocation of compile-time procedures (the latter provides a restricted macro facility). These commands are discussed in Sections 14.3 and 14.4. Any <compile-time command> which occurs at program level may only produce <directive>s and <declaration>s (i.e., <statement>s are not permitted).

4.1.4 Statements

SYNTAX DIAGRAM

5



*<Statement>*s denote algorithmic actions, and are said to be executable. They may be prefixed by a label which can be referenced by *<goto statement>*s and *<exit statement>*s. The presence of a label identifier on a *<statement>* constitutes a declaration of the identifier in the scope which immediately contains the *<statement>*. (See Section 4.2.1 for a definition of "immediately contains".)

The <statement> alternative comprising an identifier and a semicolon may only appear in the <body> of a <compile-time procedure declaration> (see Section 14.4). This alternative is used when a <statement> is passed as a compile-time actual parameter.

Many of the <statement>s in REDL have forms with an end delimiter and an optional label identifier. The following rule applies to such <statement>s: if the label identifier appears, then the <statement> must be prefixed by the same label identifier. Thus L: BEGIN ...END L; and L: BEGIN ...END; are both correct, whereas L: BEGIN ...END M; is incorrect.

4.1.5 Terminology.

4.1.5.1 Program Element. A general term designating any entity with which a name has been associated via a declaration; e.g., a declared constant or variable, formal parameter, routine, type, or capsule. Program elements are the units with which certain <directive>s deal, such as the pervasive and import <directive>s. The term "element" is sometimes used as an abbreviation of "program element".

4.1.5.2 Data Element. A program element which is a data object: i.e., a declared constant or variable, a routine parameter, or an enumeration type element. Routines, labels, and exceptions are not data elements.

4.1.5.3 Object (or Data Object). A more general term than "data element", since objects need not have names. For example, <expression>s produce (nameless) objects, and <array component>s, <record component>s, <union alternative>s, and <pointer dereference>s are objects but not data elements.

4.1.5.4 W-Valued and R-Valued. An object is said to be either W-valued or R-valued (but not both). If R-valued, it may only be used in contexts where it is "read". Constants, literals, CONST formal parameters, the results of <expression>s, and data imported readonly are all R-valued. If an object is W-valued, then it may be used in contexts where it is either "written" or read. Variables, VAR and RESULT formal parameters, and components of W-valued objects are all W-valued. The contexts in which only a W-valued object may be used are (1) the target of an <assignment statement>, and (2) an actual parameter bound to a VAR or RESULT formal parameter. Unless otherwise specified by the context in which it is used, the term "value" means "R-valued object".

4.2 Namescopes.

4.2.1 Introduction

The main purpose of this section is to explain how to resolve references to names; i.e., given the occurrence of a name, to determine the declaration which associates the name with the appropriate program element. The rules in REDL use the static (or "lexical") structure of the <program>, in the spirit of Pascal or ALGOL 60. The major differences between REDL and these other languages are:

(1) REDL does not allow names to be known automatically in all inner scopes; for example, a "closed" scope such as a routine can only use names which are "pervasive" or which it explicitly declares or imports.

(2) Through <routine declaration>s, the same name may be "overloaded" with several different interpretations in the same scope.

(3) Aside from point (2), a name declared in a scope cannot be redeclared in any scope in which its original declaration is known.

A namescope (or scope for short) is a language construct which can contain declarations of names. Thus <program>s, routines and capsules are scopes, as are a variety of <statement>s and <statement> constituents. For example, the begin and for <statement>s and the branches of <if statement>s are scopes.

Any program text which is lexically within a scope is said to be contained in that scope. (Otherwise, it is said to be external to that scope.) Any program text contained in a scope S but not contained in any scope contained in S is said to be immediately contained in S. Each program element's declaration

is thus immediately contained in exactly one scope. (Declarations for built-in program elements (such as +, -, BOOLEAN) are not immediately contained in any scope explicitly provided in the <program>; instead, they (and the <program> itself) are immediately contained in a conceptual external scope.)

Declarations immediately contained in the scope delimited by PROGRAM and END PROGRAM are said to be at the program level. Declarations immediately contained in a program level <capsule declaration> are also said to be at program level.

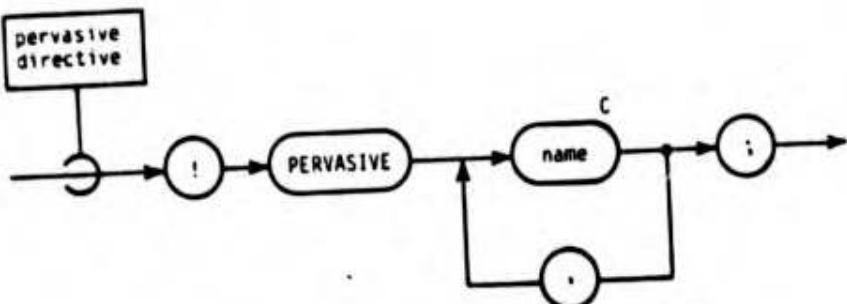
A name N known in a scope S may or may not be known in a scope SS immediately contained in S. When N is known in SS, it is said to have been inherited by SS. The next several sections discuss the rules of inheritance in REDL.

The <pervasive directive> (Section 4.2.2) causes names to be inherited by all contained scopes. The <import directive> (Section 4.2.3) causes a designated name to be inherited only by the importing scope and may prevent other names from being inherited. The <readonly directive> (Section 4.2.5) causes data elements to be inherited in a non-writable way.

4.2.2 Pervasive Directive.

SYNTAX DIAGRAM

6

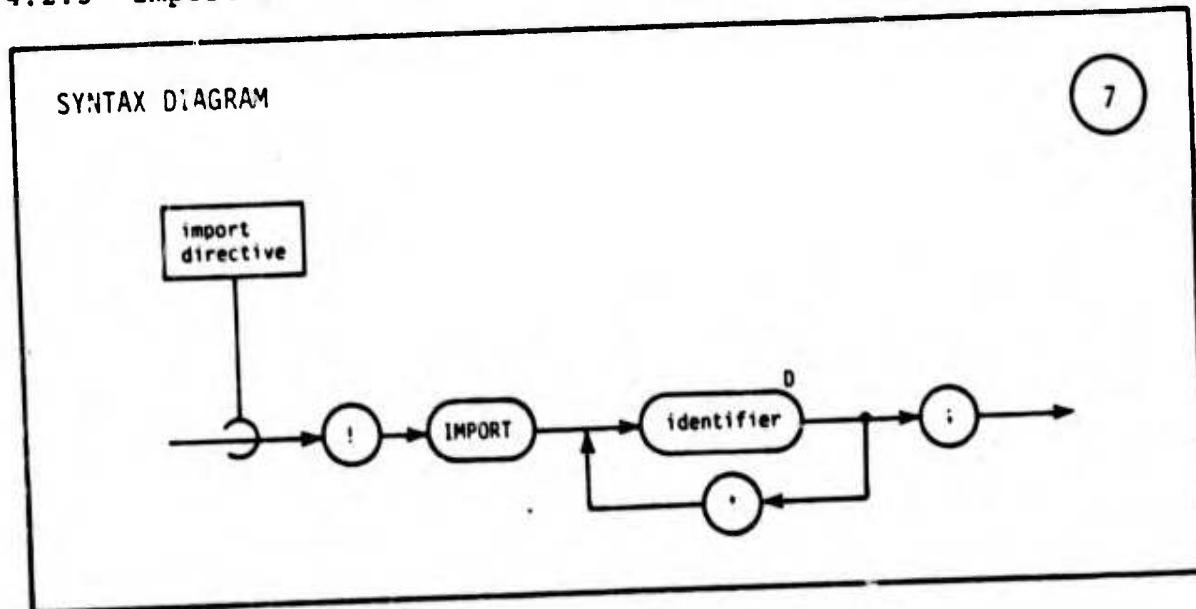


The purpose of the <pervasive directive> is to provide a facility for the frequent cases where the programmer declares a name in one scope and intends that name's declaration to be known in all inner scopes. The names listed in the <pervasive directive> must have declarations which are known in the scope which immediately contains the <directive>. The only restriction is that any data element identifier which appears must be either a constant or a CONST formal parameter. This restriction assists in the prevention of side effects in functions and aliasing in procedures.

A name is said to be pervasive in a scope S if it occurs in a <pervasive directive> either in S or in some scope containing S. If a name N is pervasive in S, then N's declaration (or declarations, in case of overloading) known in S will be automatically known in all scopes contained in S. Thus, the name can be referenced in inner closed scopes without being explicitly imported into such scopes. Probably the most frequent uses of this directive will be for type, routine and constant identifiers.

The names predefined in REDL for types, routines, and exceptions are implicitly established as pervasive in the external scope. Thus they may be used throughout the program without being explicitly imported, and they may not be overridden (although the operators may be overloaded).

4.2.3 Import Directive.



The purpose of the <import directive> is to provide an explicit filter on program element names which are declared outside, but are to be referenced inside, a scope. For ease of description, we refer to the scope immediately containing the <directive> as S.

Each name in the <import directive> must be known in the scope which immediately contains S. The effect of the <import directive> is to establish these names as known in S. Moreover, these are the only names known in S, except for those which are either pervasive or declared in S, or present in another <import directive> immediately contained in S.

4.2.4 Open and Closed Scopes.

Scopes in REDL are divided into two classes, open and closed, corresponding to whether the names of non-pervasive program elements are automatically inherited. The open scopes are those scopes which are <statement>s or <statement> constituents; the closed scopes are

primarily those scopes which are themselves declarations. A precise classification is given in Table 4-1.

We can summarize informally the relationship between open and closed scopes and the pervasive and import <directive>s:

- (1) Pervasive elements are automatically inherited by all inner scopes.
- (2) In the absence of <import directive>s, an open scope automatically inherits all non-pervasive elements, whereas a closed scope inherits none.
- (3) If a scope (open or closed) immediately contains any <import directive>s, then the only non-pervasive elements inherited by the scope are the ones named in these <directive>s.

Section Reference

Open Scopes:

<begin statement>	9.4
THEN, ELSEIF, and ELSE <body>s of <if statement>	9.5.1
<case alternative> and <otherwise alternative> of <select statement>	9.5.2
<body> of <while statement>	9.6.1
<body> of <repeat statement>	9.6.2
<for statement>	9.6.3
<body>s of <on statement>	11.3.2
<fork statement>	12.2.1

Closed Scopes:

<program>	4.1
<capsule declaration>	6.0
<routine declaration>	7.1
<path clause> of <fork statement>	12.2.2
<compile-time procedure declaration>	14.4

Table 4-1. Open and Closed Scopes.

A more formal specification of the semantics of "open" and "closed", together with a definition of the concept of a declaration's being known in a scope, is as follows.

(1) Every declaration is known in the scope in which it is immediately contained.

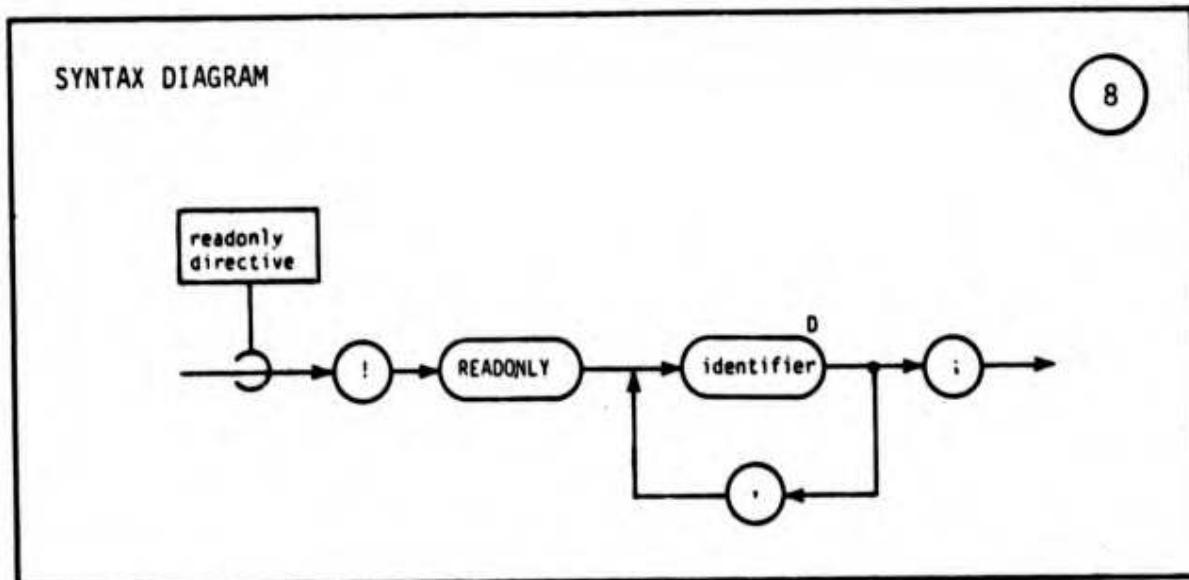
(2) If S is a scope (open or closed), and N is the name of a program element which is pervasive in the scope immediately containing S, then any declaration of N known in this containing scope is also known in S.

(3) If S is an open scope and N is the name of a non-pervasive program element whose declaration is known in the scope immediately containing S, then N's declaration is also known in S unless the programmer explicitly inhibits such inheritance. This inhibition is accomplished via the presence of one or more <import directive>s immediately in S, such that N appears in none.

(4) If S is a closed scope and N is the name of a non-pervasive program element whose declaration is known in the scope immediately containing S, then N's declaration is also unknown in S unless the programmer explicitly enables such inheritance. This enabling is accomplished via the presence of some <import directive> immediately in S, such that this directive mentions N.

We say that a name is known in a scope S provided that there is known in S some declaration of the name. If a name is referenced in a scope, then it must be known in the scope. The only circumstance under which several declarations for a name are permitted to be known in the same scope is the declaration of overloadable routines; the number and types of parameters here determine which declaration is to be used.

4.2.5 Readonly Directive.



The purpose of the <readonly directive> is to allow a scope to restrict the kinds of references it may make to data elements. For ease of description, we refer to the scope immediately containing the <directive> as S.

Each identifier in the <readonly directive> must be known in S as a data element. (The most common cases will be where S is a closed scope and the identifiers are imported, and where S is an open scope and the identifiers are known in the scope immediately containing S.) Within S, and within all scopes contained in S in which the identifier is known, each identifier may only be referenced in R-valued contexts.

4.2.6 Reuse of Names.

One of the basic principles of Pascal and ALGOL 60 namescopes is that any declared name can be redeclared in any contained scope, thus overriding the outer definition. In the interest of program reliability, readability, and maintainability, REDL is not so liberal. In REDL, if a name would be known in a scope S through a declaration external to S--i.e., if the name is pervasive in S or imported into S, or S is an open scope with no <import directive>--then it is illegal to provide a declaration for the name immediately within S (unless both declarations are for overloaded routines). Even in the case of overloading, however, the new declaration must add to the set of overload lines without overriding any which is known in S. Thus declarations are not overridable, but they are hidable via closed scopes and/or <import directive>s.

4.2.7 Example.

The following program fragment illustrates REDL's namescoping conventions.

```
! PervasivE PC;
CONST PC: ...;
VAR V1,V2: ...;

PROCEDURE Q;
    <*Only PC is known here*>
    :
END PROCEDURE Q;

A: BEGIN
    !IMPORT V1,A;
    !READONLY V1;
    <*PC, V1, and A are known; V2 and Q are not
     *If the import directive were absent, V2 and Q
     <*would also be known*>
    :
END A;

B: BEGIN
    VAR X: ...;
    BEGIN
        VAR X: ...; <*Illegal, since outer X is known here*>
        :
    END;

    BEGIN
        !IMPORT V1;
        VAR X: ...; <*Legal, since outer X is not known here*>
        :
    END;

    :
END B;

C: BEGIN
    VAR X: ...; <*Legal, since B and C are disjoint scopes*>
    B: ... <*Illegal, since label B from outside is known here*>
    :
END C;
```

4.2.8 Forward References.

As noted in Section 4.2.4, a name must be known in a scope in order to be referenced within the scope. In general, a name will be declared before (i.e., lexically preceding) any reference to it. However, a forward reference (a reference to a name before or within its declaration) is permitted in the following contexts:

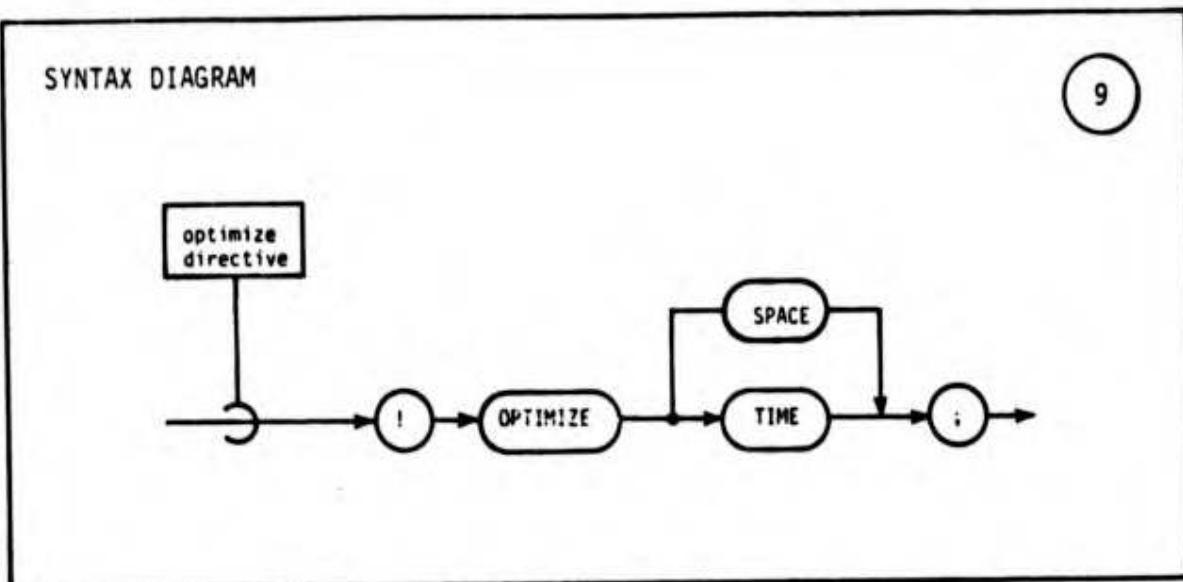
(1) Labels. A label may be referenced in a <goto statement> before being declared.

(2) Routines. A routine R₁ may be referenced in a routine R₂ even when R₂'s declaration precedes R₁'s. Recursion and mutual recursion are thus allowed.

(3) Types. A type identifier T₁ may be referenced in the declaration of a pointer type identifier T₂ even when T₂'s declaration precedes T₁'s. Recursive types are thus allowed.

(4) Directives. The pervasive, promote, export, align, and store <directive>s may reference a name before its declaration has appeared.

4.2.9 Optimize Directive.



The <optimize directive> informs the compiler as to the tradeoff to make (execution time vs. data space) in generating code for a namescope. If SPACE appears, the compiler will opt for producing compact object code. If TIME appears, the compiler will opt for generating object code which executes quickly.

An <optimize directive> appearing in a scope overrides the effect of <optimize directive>s of enclosing scopes. For example, space optimization can be specified for a routine, while time optimization can be specified for a frequently executed inner loop of the routine. In this way, space is sacrificed only in that portion of the routine where large gains in overall system execution speed are expected.

5.0 DATA TYPES

REDL is strongly typed: each data object has a unique type, determinable at compile-time, which defines the possible values for the object and its set of behavioral properties. Figure 5-1 displays a categorization of the types and type generators provided in the language.

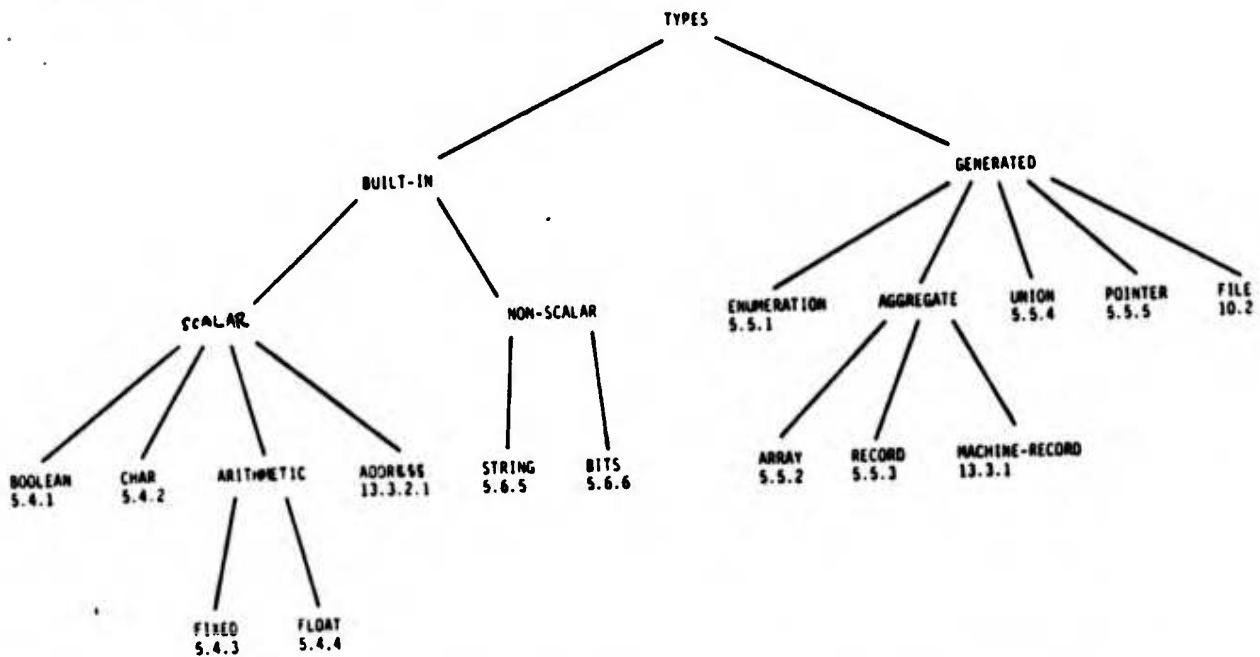


FIGURE 5-1: Types and Type Generators, with Section References

5.1 Basic Concepts.

5.1.1 Attributes and Representation.

It is not necessarily the case that different objects of the same type possess identical representations. For example, two objects of type STRING may have different numbers of components, and FLOAT objects of different precision may be represented differently. Size and precision are said to be attributes of the

STRING and FLOAT types, respectively. Individual data objects are said to possess values for the attributes of their types. For example, saying that the STRING literal 'LANGUAGE' has size 8 is short for saying that 8 is the value of the size attribute for the literal.

An attribute of a type T is thus a property of T's objects, and different objects of type T may have different values for the attribute. The set of attributes for any type T is always determined by T's declaration. The values of these attributes for any data object D having type T are always determined at the time of D's creation, and these values will not change during the lifetime of D. We may regard the representation of D as simply a list of values, one for each attribute of T. Thus each data object in a <program> has a type, established at compile-time; a representation, established at the time the object is created and unchangeable thereafter; and a value, established when the object is initialized or assigned to and changeable (if the object is a variable) by subsequent assignments.

The values of some attributes are always compile-time determinable; e.g., the scale of any FIXED object and the precision of any FLOAT object. In some cases, such as range bounds for FIXED or FLOAT objects, determination of the values of attributes may be deferred until run-time. In other cases (viz., for parameterized types) the value of an attribute may be "unresolved": if a formal parameter to a routine is specified with an unresolved value for some attribute, then different actual parameters may have different values for this attribute.

Every type has an attribute known as grouping (for a description of the semantics of this attribute, see Section 5.8). In addition to grouping, each scalar type (including user-defined enumeration types) has a specific set of attributes dependent on the type, and each parameterized type has as attributes the formal parameters to the type.

To interrogate the value of an attribute for an object of type T, the programmer may either use a function invocation form (for grouping and scalar attributes) or a qualification form (for type formal parameters).. For example, SCALE(X) returns the value of the scale attribute for any FIXED object X; with the parameterized declarations

```
TYPE VECTOR(N): ARRAY(1..N) OF FLOAT(6, 0.0 .. 1.0);
VAR X: VECTOR(5) INIT VECTOR(5 OF 0.0);
```

the form X.N gives the value of the N attribute of X (in the example, this value is 5).

5.1.2 Assignment and Parameter Passing.

In the case of types which are not parameterized, the following rules relate types and representations to assignment and parameter passing. (The analogous rules for parameterized types are given in Section 5.6.4.)

(1) Assignment, initialization, and parameter passing by CONST or RESULT are permitted (with respect to compile-time checking) if and only if the source and target have the same type.

(2) VAR parameter passing is permitted if and only if the formal and actual parameters have both the same type and the same representation (i.e., the same values for each attribute).

A minor exception to rule (1) in the case of the FIXED type is described in Section 5.4.3.4. We also point out that assignment (and hence initialization, CONST, and RESULT parameter passing) will raise the X_RANGE exception in the case of FIXED, FLOAT, or enumeration types, when the source value is not within the range of the target.

5.1.3 Type Naming.

A fundamental principle of REDL's data facility is that the type of any data element is always named explicitly; i.e., it is always given by a single identifier, and distinct identifiers denote distinct types. For example, the following attempted declarations are syntactically illegal:

```
VAR V: ARRAY(1..16) OF BOOLEAN INIT ?;  
TYPE T: RECORD  
    A: BOOLEAN;  
    B: POINTER(VAR FIXED(-100..100));  
END RECORD;
```

Correct versions are the following:

```
TYPE V_TYPE: ARRAY(1..16) OF BOOLEAN;  
VAR V: V_TYPE INIT ?;  
TYPE B_TYPE: POINTER(VAR FIXED(-100..100));  
TYPE T: RECORD  
    A: BOOLEAN;  
    B: B_TYPE;  
END RECORD;
```

One major benefit of this principle is that the rules for type identity are extremely simple; see Section 5.3.3.

5.1.4 Storage Classes.

REDL provides two storage classes, static and dynamic, exactly as in Pascal. (To prevent possible confusion, we point out that the term "static" as used in REDL and Pascal corresponds not to "static" in the sense of PL/I, but rather to PL/I's "automatic".) An object is said to be static if it has the static storage class; analogously for dynamic.

Components of aggregate objects inherit the storage class of the parent object. For example, if X is a one-dimensional array with static storage class, then each X(i) is likewise static.

Static data objects are created by explicit declaration and are referenced through identifiers. Dynamic data objects are created by calling the procedure NEW and are referenced through pointer dereferences. The difference between the two storage classes lies in the relationship between the lifetime of an object and the namescope structure of the <program>. Static data objects come into existence and are initialized on (each) entry to the scope in which they are declared, and cease to exist when control leaves this scope. This relationship does not hold for the dynamic storage class: dynamic objects can be deallocated either before or after control leaves the scope in which they are created.

The declarations which create static data objects are <const-or-var declaration>s, routine formal parameter and result variable declarations, and for-variable declarations.

5.1.5 Comparison with Pascal.

The syntax and terminology of REDL's type facility is based on Pascal, but there are a number of significant differences between the two languages.

5.1.5.1 Type Naming. REDL requires every type to be named, as mentioned in Section 5.1.3.

5.1.5.2 Arithmetic Types. REDL's FIXED and FLOAT types are substantial generalizations of Pascal's INTEGER and REAL, allowing user specification of scale and precision. Range specifications are mandatory in REDL (helping achieve machine-independence and program readability) whereas they are optional in Pascal.

5.1.5.3 Arrays. Array bounds may be run-time determined in REDL, there are forms for catenating and selecting sections of one-dimensional arrays, and there is a form for constructing arrays out of components.

5.1.5.4 Records and Unions. In Pascal the variant record is a concept which combines two independent notions: that of a structure containing all of the named fields, and that of a structure containing exactly one of the named fields. REDL separates these notions and supplies the RECORD and UNION type generators for the two purposes. There are forms for constructing record and union objects.

5.1.5.5 Bits. REDL provides the BITS type as opposed to Pascal's set generator.

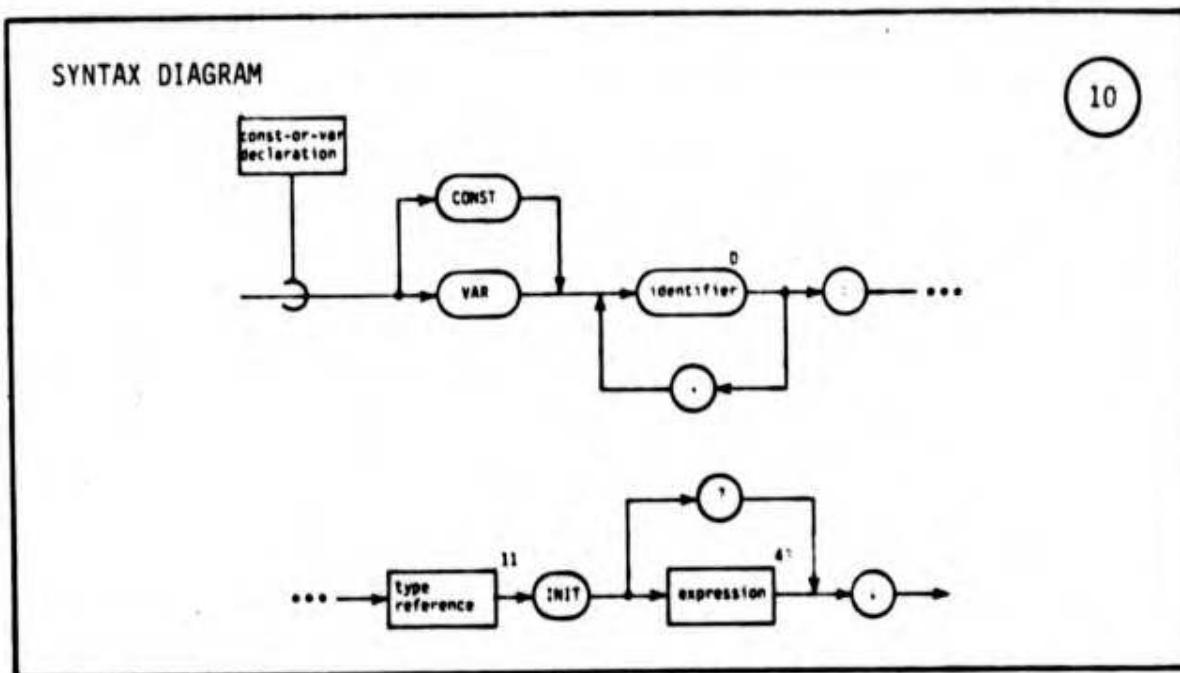
5.1.5.6 Parameterized Types. Parameterized types in REDL allow the user to write routines which can be invoked at different times with aggregate objects of different sizes. STRING and BITS are built-in parameterized types.

5.1.5.7 Assignment. REDL automatically provides assignment with each data type.

5.1.5.8 Formal Parameters to Routines. There are no restrictions in REDL, as there are in Pascal, on the types of formal parameters and function results.

5.1.5.9 Initialization. REDL requires initialization phrases on the declarations of variables, constants, and result parameters. A special form ("INIT ?") is provided if initialization is undesired.

5.2 The Declaration of Constants and Variables.



5.2.1 The Concepts of "Constant" and "Variable".

The terms "constant" and "variable" are used with specific meanings for objects of either static or dynamic storage class. An identifier declared in a <const-or-var declaration> prefixed with CONST is said to denote a static constant; an identifier declared with VAR is said to denote a static variable. The corresponding definitions for the dynamic storage class depend on the material in Section 5.5.5, but are included here for completeness: if p denotes a pointer object whose pointed-to type is specified as CONST, then the pointer dereference p@ denotes a dynamic constant; if p's pointed-to type is specified as VAR, the p@ denotes a dynamic variable.

The difference between constants and variables (whether static or dynamic) is that variables are W-valued objects whereas constants, after their initialization, are R-valued. Thus variables (but not constants) may appear in W-valued contexts (i.e., as the target of an *<assignment statement>* or as a VAR or RESULT parameter to a procedure).

5.2.2 Initialization.

The *<var-or-const declaration>* allows two forms for the initialization phrase, one involving an *<expression>* and the other specifying "?".

If it is desired to suppress initialization for a variable (this is sometimes useful, for example, if the variable is to be passed as a RESULT parameter to a procedure), then the special symbol "?" is coded following INIT. Such a variable is said to be uninitialized. An attempt to reference an uninitialized variable in an R-valued context will raise the X_INIT exception. We note that, in the absence of hardware support, the checking code for this condition can be very expensive at run-time unless the X_INIT exception is suppressed (see Section 11.4).

If an *<expression>* follows INIT, the semantics are then exactly that of assignment. For example, the declaration

```
VAR V: <type reference> INIT <expression>;
```

is equivalent to

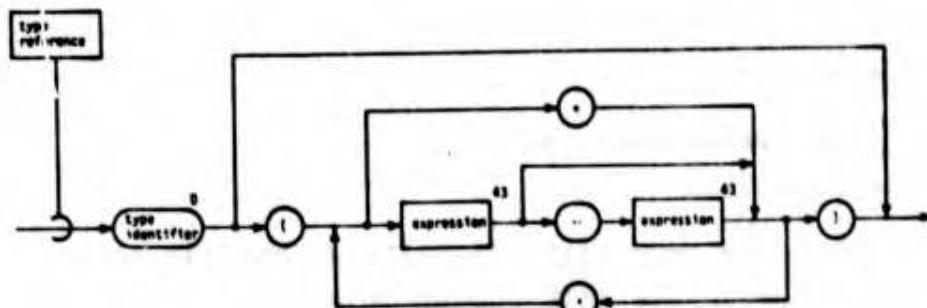
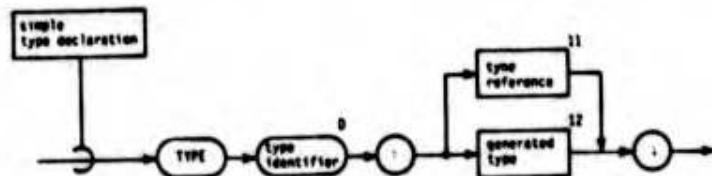
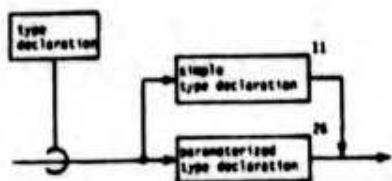
```
VAR V: <type reference> INIT ?;  
V := <expression>;
```

We note that the term initialization as used in this document includes both the INIT phrase on a declaration and the effect of the NEW procedure on the dynamic object which it creates (see Section 5.5.5.4).

5.3 The Declaration of Types.

SYNTAX DIAGRAM

11



5.3.1 Examples:

```
TYPE T1: ARRAY(1..100) OF FIXED(-1000..1000);
```

Each object of type T1 will be an array of 100 components, each component being an integer in the range -1000 through 1000.

```
TYPE T2: FLOAT(5,0.0..1.0);
```

Objects of type T2 are represented as though they were floating point values, with (at least) 5 decimal digits of precision, in the range 0.0 through 1.0. T2 and FLOAT are different types.

```
TYPE T3(M,N): ARRAY(1..M,1..N) OF BOOLEAN;
```

T3 is a parameterized type whose objects are two-dimensional boolean matrices of arbitrary sizes.

```
TYPE T4: POINTER(CONST T3(*,*));
```

Objects of type T4 point to dynamic constants of type T3; an object of type T4 may point to T3 objects of different sizes at different times.

```
TYPE T5: ARRAY(-K..K) OF STRING(5);
```

Each object of type T5 will have the same number of components (viz., the value of $2*K+1$ at the time this <type declaration> is processed in the scope in which it appears), and each component will be a STRING of 5 CHARs.

5.3.2 Simple and Parameterized Type Declarations.

Each data object has a single type, denoted by an identifier, which is either built into the language or declared by the user in accordance with normal namescoping conventions.

<Type declaration>s (and thus types themselves) are said to be either simple or parameterized. The semantics of type parameterization is discussed in Section 5.6.

A <type reference> consists of a type identifier together with a list of specifications for the values of the attributes of the type. If any of these specifications is an asterisk, the <type reference> is said to be unresolved; otherwise, it is said to be resolved. For example, the <type reference> T3(*,*) is unresolved, and all the other <type reference>s in the examples above are resolved. The only places where unresolved <type reference>s may appear are in declarations of formal parameters to routines, and in the declaration of a pointer type.

5.3.3 Type Identity.

Two data objects X and Y are said to have the same type provided that the identifiers denoting the types used in the declarations of X and Y are identical and refer to the same <type declaration>. In the following example, no two of the three data elements have the same type:

```
TYPE T1: ARRAY(1..3) OF FIXED(-1000..1000);
TYPE T2: ARRAY(1..3) OF FIXED(-1000..1000);
VAR V1: T1 INIT T1(10,20,30);
VAR V2: T2 INIT T2(5,10,15);

V2 := V1; /*Illegal, since V1 and V2 have different types*/
BEGIN
!IMPORT V1; /*Outer T1 is not known in this scope*/
TYPE T1: ARRAY(1..20) OF BOOLEAN;
VAR V3: T1 INIT ?;
V3 := V1; /*Illegal, since V1 and V3 have different types*/
:
END;
```

5.4 Scalar Types.

5.4.1 BOOLEAN.

5.4.1.1 General Properties. The BOOLEAN type is used to represent logical truth values. The only attribute of the type is the grouping attribute. The <type reference> for the type is simply the word BOOLEAN itself.

5.4.1.2 Literals. TRUE and FALSE are the only BOOLEAN literals.

5.4.1.3 Routines. The BOOLEAN operators consist of a prefix operator and several infix operators. In all cases, each operand must have BOOLEAN type, and the value returned has BOOLEAN type.

The prefix operator, NOT, is defined by the following table:

P	NOT P
TRUE	FALSE
FALSE	TRUE

The infix operators AND, OR, and XOR are defined by the following table:

p	q	p AND q	p OR q	p XOR q
TRUE	TRUE	TRUE	TRUE	FALSE
TRUE	FALSE	FALSE	TRUE	TRUE
FALSE	TRUE	FALSE	TRUE	TRUE
FALSE	FALSE	FALSE	FALSE	FALSE

The infix operators & and ! are short-circuited forms for AND and OR, respectively. (Mnemonically, & and ! are "shorter" than the spelled-out operators AND and OR.) By "short-circuited" we mean that the left operand is evaluated first, and, if this value determines the result of the operation, then the right operand is not evaluated. For example, in the <expression> $(I \leq N) \& (X(I) \neq 0)$, if I is greater than N then the result of the <expression> is FALSE, and $(X(I) \neq 0)$ is not evaluated. There is no short-circuited form for the XOR operator, since both operands must be evaluated to determine the result.

The relational operators = and # are defined for BOOLEANS and have their conventional meanings of equality and inequality, respectively.

5.4.2 CHAR.

5.4.2.1 General Properties. CHAR is an ordered enumeration type whose elements are identified by special enumeration symbols. Independent of the implementation, the character set denoted by CHAR is assumed to contain at least the following:

- (1) The alphabetically ordered set of capital Roman letters A through Z (denoted '^A' through '^Z'),
- (2) The numerically ordered and contiguous set of the decimal digits 0 through 9 (denoted '^0' through '^9'), and
- (3) The blank character (denoted '^ ').

The attributes of the CHAR type are grouping, low-bound, and high-bound.

The <type reference> for the type is either CHAR or CHAR(range). The range may be written as either "<expression>..<expression>" where each <expression> has type CHAR, or as a range identifier (see Section 5.7).

5.4.2.2 Literals. The form for a CHAR literal is the special enumeration symbol. A standard set of literals for 128-character ASCII is given in Appendix B. In the event that the user defines an enumeration type which contains elements also appearing in CHAR, references to such elements must be made unambiguous by a qualification form (see Section 5.5.1.1).

5.4.2.3 Routines. As an ordered enumeration type, CHAR provides the following:

- (1) Relational operators =, #, <, <=, >, >=
- (2) Predecessor and successor functions (PRED and SUCC)
- (3) Conversion functions to and from integers (the type identifiers FIXED and CHAR serve as such functions)
- (4) The attribute-inquiry functions LOW and HIGH.

It may be noted that conversions between CHAR and user-defined character sets can be realized either by functions or by arrays indexed by and containing character set values.

The following program fragment builds the integer value corresponding to a sequence of decimal digits read in. (For simplicity, the status value returned by the READ procedure is ignored.)

```
VAR CH: CHAR INIT ?;
VAR STATUS: FIXED(MIN_INTEGER..MAX_INTEGER) INIT ?;
VAR N: FIXED(0..MAX_INTEGER) INIT 0;
CALL READ(CH,STATUS);
WHILE ^0^ <= CH & CH <= ^9^ DO
    N := 10*N + FIXED(CH) - FIXED(^0^);
    CALL READ(CH,STATUS);
END WHILE;
```

5.4.3 FIXED.

5.4.3.1 General Properties. REDL's FIXED type provides for scaled integer arithmetic as required by Ironman 3.1, and offers a system in which either exact or approximate fixed-point computation can be performed. A scaled integer is the product of an integer mantissa, M, and a compiler-maintained scale (or step size), S. For example, the abstract value three-eighths could be represented by a mantissa 3 and a scale one-eighth. Alternatively, other representations are possible: mantissa 1 and scale three-eighths, mantissa 375 and scale one-thousandth, for example. Actual mantissa and scale may be either specified by the user or determined by the compiler.

The attributes of the FIXED type are grouping, scale, low-bound, and high-bound.

The <type reference> for FIXED has the general form FIXED(scale,range). The scale may be any compile-time determinable FIXED value. The range may be written as either "<expression>..<expression>", where each <expression> has type FIXED, or as a range identifier (see Section 5.7).

If the scale specification is omitted from a declaration, the default is scale 1 (i.e., integers). The term integer is used in this document to mean a FIXED object with scale 1. The range specification may be omitted only in the declaration of compile-time constants; see Section 14.2.1.

The following are examples of declarations of FIXED objects:

```
VAR A: FIXED(1B-3, 0..7B-3) INIT 0;
```

The scale for A is 1B-3 (i.e., one-eighth), and the range is from 0 to seven-eighths. The possible values for A's mantissa are 0,1,...,7.

```
VAR B: FIXED(0.01, -100..100) INIT 1.0;
```

The value-space for B is -100,-99.99,-99.98,...,99.99,100.

```
VAR C: FIXED(-M..N) INIT (N-M) DIV 2;
```

C is an integer in the run-time determined range -M through N.

```
%CONST D: FIXED INIT 4;
```

D is the integer compile-time constant 4.

The machine-dependent integer constants MIN_INTEGER and MAX_INTEGER reflect the smallest and largest mantissa values supported by the target machine.

It was mentioned at the beginning of this section that REDL's FIXED type can support approximate numeric computation if exact computation is not required. To achieve this, the user selects a scale of suitable granularity. The following declarations illustrate how to obtain the behavior of approximate fractional arithmetic (i.e., with a value space between -1 and +1 exclusive).

```
%CONST FRAC_SCALE: FIXED(1B-31) INIT 1B-31;  
/*This establishes FRAC_SCALE as the compile-time constant 2^-31*/  
%CONST FRAC_LOW: FIXED(FRAC_SCALE) INIT -1+FRAC_SCALE;  
%CONST FRAC_HIGH: FIXED(FRAC_SCALE) INIT 1-FRAC_SCALE;  
RANGE FRAC_RANGE: FRAC_LOW..FRAC_HIGH;  
VAR X: FIXED(FRAC_SCALE,FRAC_RANGE) INIT 0;
```

The reason we say that X is "approximate" is that assignments to X from decimal scaled data (i.e., data whose scale is a power of ten) will have to specify ROUND or TRUNCATE (see Section 5.4.3.3), since such values are not, in general, exactly representable with scale 2^{-31} .

5.4.3.2 Literals. The general form for numeric literals is given in Lexical Diagram G. (It may be noted that this same form is used for both FIXED and FLOAT literals; the compiler will assume FIXED unless the literal appears in a context where a FLOAT value is required.) The "E" may be read as "times ten to the power", whereas "B" may be read as "times two to the power". For example, 1.23E12 represents $1.23 \cdot 10^{12}$, and 3B-5 represents $3 \cdot 2^{-5}$.

Any FIXED literal is representable in a variety of scales. The choice made by the compiler will depend on the scale required by the context in which the literal is used. If the programmer wishes to specify the scale explicitly, one of the rescaling functions described in Section 5.4.3.3 may be used.

When a FIXED literal appears as a scale specification in a FIXED <type reference>, the choice of binary or decimal scaling depends on the presence of "B" or "E" in the literal. Thus the programmer can use the form of the literal to achieve an efficient implementation. For example, although 5B3 = 4E1 = 40, these three forms may have different effects on the representation of FIXED values when used as scale specifications.

5.4.3.3 Routines.

(1) Arithmetic Routines. The prefix arithmetic operators defined for the FIXED type are + and -. The + operator simply returns the value of its operand and the - operator signifies negation. The scale of the result is the same as the scale of the operand.

The infix arithmetic operators defined on FIXED operands are + (addition), - (subtraction), * (multiplication), / (division with FLOAT result), DIV (division with integer FIXED result), MOD (remainder). Each operation except / yields a FIXED result. Table 5-1 displays the scale of the result of FIXED operations:

Operation	Scale of Result (assume that S1 and S2 are the scales of the operands)
+,-,MOD	$\text{GCD}(S1, S2)$
*	$S1 \times S2$
DIV	1

Table 5-1. Scale Determination for FIXED Operations

In Table 5-1, $\text{GCD}(S1, S2)$ denotes the Greatest Common Divisor of S1 and S2 and is applicable for both integers and non-integers. For example, $\text{GCD}(12, 18) = 6$; $\text{GCD}(0.1, 1B-1) = 0.1$.

The effect of DIV and MOD is as follows:

(a) if $Y \neq 0$ then

$$(i) \quad X = Y * (X \text{ DIV } Y) + (X \text{ MOD } Y)$$

(ii) $(X \text{ MOD } Y)$ has the same sign as Y (thus the DIV operator truncates toward minus infinity)

$$(iii) \quad 0 \leq |X \text{ MOD } Y| < |Y|$$

(b) $X \text{ DIV } Y$ and $X \text{ MOD } Y$ both raise the `X_ZERO_DIVIDE` exception if $Y = 0$.

The `X_OVERFLOW` exception will be raised if a value is produced outside the range implemented for the result of the operation. This exception may be raised either during the rescaling of the operands to the scale of the result, or during the actual operation. The implemented range will always be at least that required for the eventual target of the result (e.g., in an `<assignment statement>` the implemented range for the operations on the right-hand side will always be at least as large as the range of the left-hand side).

To allow FIXED division to return a result of user-specified scale, REDL provides the function `FDIV(S,X,Y)` where S is the scale of the result (S must be a compile-time determinable positive FIXED value), and X and Y have type FIXED. It follows that $X \text{ DIV } Y$ is equivalent to `FDIV(1,X,Y)`.

(2) Relationals. The relational operators `=`, `#`, `<`, `<=`, `>`, `>=` are defined for FIXED operands. It may be noted that the operands (if of different scale) may have to be rescaled to a common scale (their GCD) with the resulting possibility of an `X_OVERFLOW` exception being raised.

(3) Scale and Range Inquiry Functions. `SCALE(X)` is a compile-time evaluable function returning the scale of `FIXED <expression> X`. `LOW(X)` and `HIGH(X)` return the low and high bounds, respectively, of the range of `X`. Machine-dependent functions `LOW_ACTUAL(X)` and `HIGH_ACTUAL(X)` return the actual (i.e., implemented) low and high bounds of the range of `X`. `LOW_ACTUAL` and `HIGH_ACTUAL` are compile-time evaluable; `LOW` and `HIGH` might not be. As an example, if `X` is declared:

```
VAR X: FIXED(0..10) INIT 0;
```

then `SCALE(X) = 1`, `LOW(X) = 0`, `HIGH(X) = 10`, and possibly `LOW_ACTUAL(X) = -215` and `HIGH_ACTUAL(X) = 215-1`.

(4) Explicit Scale Conversions. `TRUNCATE(S,X)` and `ROUND(S,X)` are functions which take a compile-time-evaluable `FIXED <expression> S` (the scale) and an `<expression> X` of type `FIXED`. Let `SX = SCALE(X)`. The result of each function is a `FIXED` value with scale `S`. If `SX` is not an integer multiple of `S`, then information may be lost in the rescaling; `TRUNCATE` indicates a truncation of the value (toward minus infinity), and `ROUND` indicates a truncation of the sum of the value and half the scale `S`. If `SX` is an integer multiple of `S`, then no information is lost in the rescaling, and `TRUNCATE` and `ROUND` have the same effect.

Examples:

```
VAR DEC: FIXED(0.1, 0..100) INIT 0.1;  
VAR BIN: FIXED(1B-3, 0..100) INIT ?;  
BIN := TRUNCATE(1B-3, DEC); /*Now BIN = 0*/  
BIN := ROUND(1B-3, DEC); /*Now BIN = 1B-3*/
```

(5) Explicit Type Conversions. The scale conversion functions TRUNCATE and ROUND are overloaded so that they work on FLOAT as well as FIXED objects. That is, TRUNCATE(S,X) and ROUND(S,X) are defined when S is a FIXED scale and X is a FLOAT object. The effect is to convert X from FLOAT to a FIXED value of scale S, with either truncation or rounding.

The type identifier FIXED may itself be used as an explicit conversion function from any ordered enumeration type. A description of the semantics is in Section 5.5.1.3.

5.4.3.4 Assignment and Parameter Passing. FIXED is different from other types in that assignment is restricted (i.e., it is not always legal to assign an arbitrary FIXED value to a FIXED object). Assignment is legal if the scale of the target permits the exact representation of any value having the source scale, and is illegal otherwise. For example, if X,Y,Z are objects having scales 2,3, and 6, respectively, then the assignments $X := Z$ and $Y := Z$ are permitted, and the other four ($Y := X$, $Z := X$, $X := Y$, $Z := Y$) are illegal. More formally, let ST and SS be the target and source scales, respectively, for an assignment. The assignment is legal if and only if SS is an integer multiple of ST. If this is not the case, the user must provide an explicit TRUNCATE or ROUND to control the loss of significant digits.

The above restriction on assignment applies also to initialization and CONST and RESULT parameter passing.

It should be noted that the above rule pertains to the compile-time legality or illegality of assignment; even though an assignment is legal, an X_RANGE exception may still be raised at run-time if the source is outside the range of the target.

5.4.4 FLOAT.

5.4.4.1 General Properties. The values of type FLOAT are a finite approximation to numbers in scientific notation. It is useful to view a FLOAT object as being represented by a mantissa M, an exponent E, and a radix R; the value denoted is $M \cdot R^E$.

The attributes of the FLOAT type are grouping, precision, low-bound, and high-bound.

The <type reference> for FLOAT has the general form FLOAT (precision, range). The precision must be a compile-time determinable positive integer value. The range may be written as either "<expression>..<expression>", where each <expression> has type FLOAT, or as a range identifier (see Section 5.7). Both precision and range specifications are mandatory, except in the declaration of compile-time constants (see Section 14.2.1); in such contexts each is optional.

The precision is interpreted as the minimum number of (decimal) digits contained in the mantissa.

Examples of FLOAT declarations are:

```
%CONST PI: FLOAT INIT 3.14159;  
VAR THETA: FLOAT(6, 0 .. 2.0*PI) INIT PI/2.0;  
VAR AREA: FLOAT(6, 0.0 .. 1000.0) INIT 0.0;
```

The integer configuration constants MIN_EXPONENT, MAX_EXPONENT, RADIX, and MAX_PRECISION may be referenced in machine-dependent <program>s. These constants are the smallest exponent, the largest exponent, the radix, and the largest precision which are implemented for the particular target machine. The X_UNDERFLOW (or X_OVERFLOW) exception is raised by any routine which produces a FLOAT value with exponent less than MIN_EXPONENT (or greater than MAX_EXPONENT).

5.4.4.2 Literals. FLOAT literals have the same form as FIXED literals; see Lexical Diagram G and Section 5.4.3.2. The precision of a FLOAT literal is the number of decimal digits (not counting leading zeroes) present in the "fraction" part of the literal. For example, 1.000E7 has precision 4, 123.45 has precision 5, 0.001 has precision 1, 123 has precision 3.

5.4.4.3 Routines.

(1) Arithmetic Operations. The arithmetic operators defined on FLOAT operands are prefix + and -, and infix +, -, *, and /. The ** operator (exponentiation) takes a FLOAT and a compile-time evaluable integer. Each operator yields a FLOAT value.

The precision of the result of each operator, called the defined precision, is the smaller of two values: MAX_PRECISION and the precision given in Table 5-2 below.

Expression to Be Evaluated	Result Precision (Assume that p1, p2 are precisions of X1, X2)
X1 + X2	MAX(p1,p2) + 1
X1 - X2	MAX(p1,p2) + 1
X1 * X2	p1 + p2
X1/X2	MAX(p1,p2)
X1**I	MAX(p1*I,p1)

Table 5-2. Result Precision Obtainable from Given Operand Precisions.

In practice, the actual precision supplied by an implementation may differ from the defined precision specified above, depending on the context in which the operation appears:

(a) Greater precision may be supplied if efficiency is not thereby sacrificed. For example, although the defined precision for the quotient $1.0/3.0$ is 2, the actual precision may be larger.

(b) Less precision may be supplied if the target of the operation does not require as many digits as given by the defined precision. For example, in the assignment $A := B*C$, if A, B, and C all have the same precision p then the product may be performed with precision p rather than with the defined precision $2p$.

(2) Relational. The relational operators $=$, $\#$, $<$, \leq , $>$, \geq are defined for FLOAT operands. If the actual precisions of the operands differ, the operand of smaller precision is treated as though its mantissa were filled out with trailing zeroes. It should be noted that the $=$ operator denotes identity of operands, and that caution should thus be exercised in the use of $=$ and $\#$ (in light of the approximation inherent in FLOAT values).

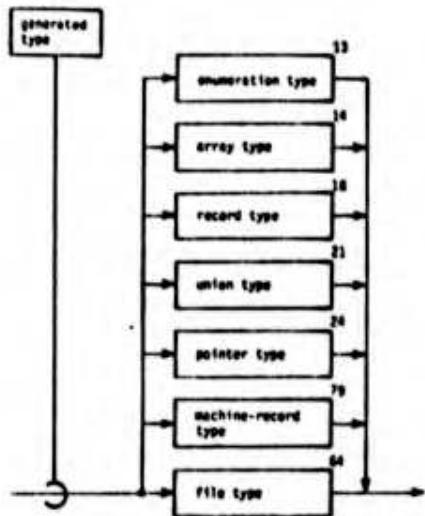
(3) Precision and Range Functions. PRECISION(X) is a compile-time evaluable function returning the defined precision of FLOAT expression X. PRECISION_ACTUAL(X), also compile-time evaluable, is a machine-dependent function which returns the actual (i.e., implemented) precision of X. LOW, HIGH, LOW_ACTUAL, and HIGH_ACTUAL are analogous to the corresponding functions for FIXED.

(4) Explicit Type Conversion. The type identifier FLOAT may be used as an explicit conversion function from FIXED. The form for this conversion is FLOAT(precision, fixed-value) where precision must be a positive compile-time evaluable integer <expression> and fixed-value is an <expression> of type FIXED.

5.5 Type Generators.

SYNTAX DIAGRAM

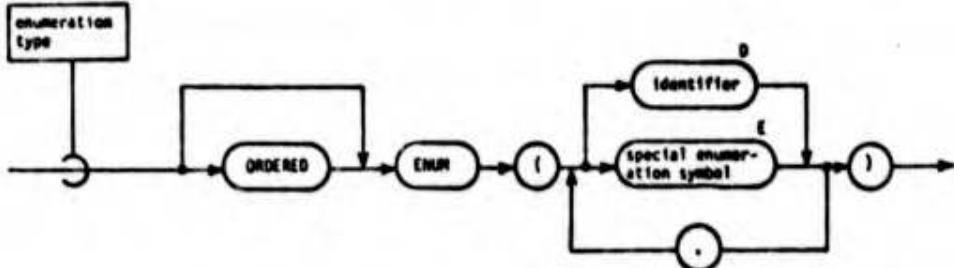
12



5.5.1 Enumeration.

SYNTAX DIAGRAM

13



5.5.1.1 General Properties. Enumeration types in REDL are basically like their Pascal analogue, with the following differences:

(1) Enumeration types in REDL may be either ordered or unordered; in Pascal they are always ordered.

(2) Special enumeration symbols are permitted as enumeration elements in REDL, facilitating user definition of character sets.

(3) Enumeration types may overlap in REDL (i.e., the same enumeration element may occur in different enumeration types). When this happens, the programmer must use a qualification form to disambiguate the literal. For example, if enumeration types COLOR and FLAVOR are declared as follows:

```
TYPE COLOR: ORDERED ENUM(RED,ORANGE,YELLOW,GREEN);  
TYPE FLAVOR: ENUM(ORANGE,GRAPE,LIME);
```

then the forms COLOR.ORANGE or FLAVOR.ORANGE must be used to refer to the element ORANGE.

The <type reference> for an enumeration type T is either T itself or, only if T is ordered, T(range). For example, COLOR(RED..YELLOW) is a legal <type reference> for COLOR.

An enumeration type is said to be either ordered or unordered, depending on the presence of the word ORDERED in its declaration. An ordered enumeration type defines an implicit total ordering among its elements, based on the sequencing of the elements in the declaration of the type.

The attributes of an enumeration type are grouping and, if ordered, low-bound and high-bound.

The identifiers and special enumeration symbols comprising an enumeration type are considered to be declared constants in the scope in which the type is declared.

5.5.1.2 Literals. The only special form for enumeration type literals is the special enumeration symbol.

5.5.1.3 Routines. The relational operators = and # are defined for each enumeration type. For ordered enumeration types (but not for unordered ones) a set of additional functions and operators is provided. Assume that T is an arbitrary ordered enumeration type, and that X is an <expression> of type T (perhaps declared with the <type reference> T(range)).

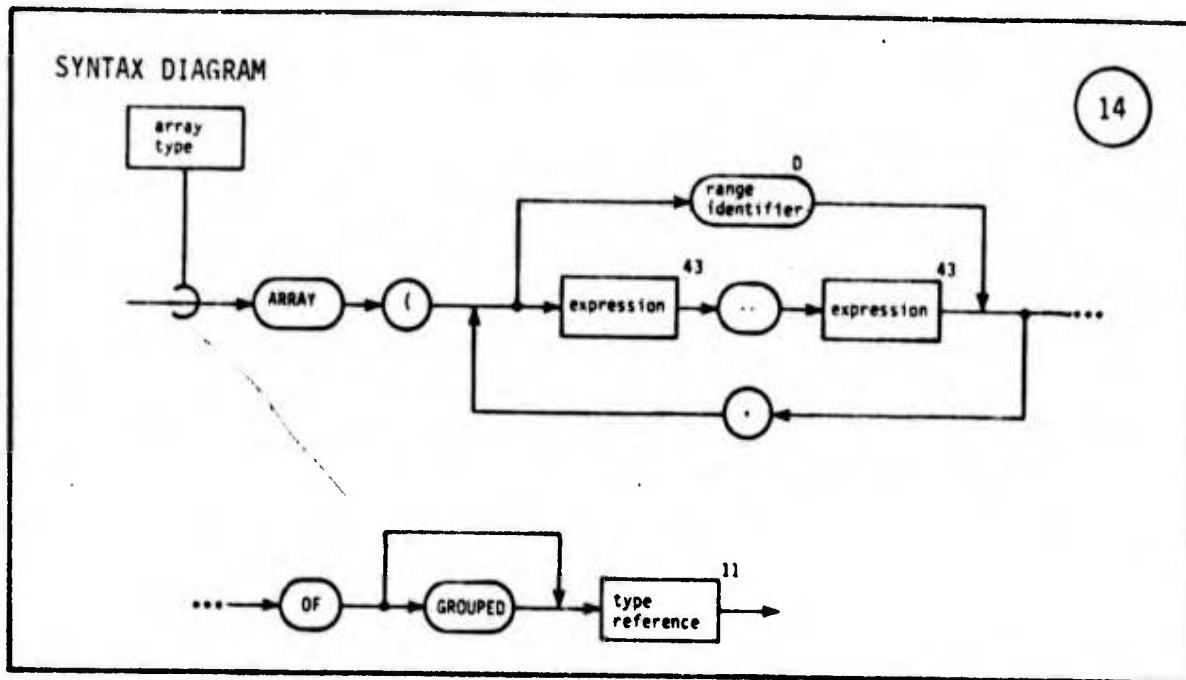
(1) LOW(X) and HIGH(X) are attribute-inquiry functions returning the first and last elements, respectively, in T (or in the range). For example, if X is declared to be a COLOR(RED..YELLOW), then HIGH(X) = YELLOW.

(2) PRED(X) is the element preceding X in T. If X is the first element in T, the X_RANGE exception is raised. SUCC(X) is analogously defined as a successor function. For example, if X above has the value YELLOW, then SUCC(X) = GREEN.

(3) FIXED(X) is the integer corresponding to the position of X within T. If X is the first element in T, then FIXED(X) = 0. T(N), where N is an integer, is the element of T such that FIXED(T(N)) = N. If there is no such element, the X_RANGE exception is raised.

(4) The relational operators <, <=, >, and >= are defined on operands of type T.

5.5.2 Array.



Examples:

```

TYPE T1: ARRAY(1..10) OF GROUPED BOOLEAN;
TYPE T2: ARRAY(1..M, K..L) OF T1;
  
```

T1 and T2 are array types.

5.5.2.1 General Properties. The array type generator is used for declaring aggregate types whose objects are "homogeneous" (i.e., each component having the same type and representation). Unlike Pascal, the number of components may be computed at run-time. If an array type T is declared via a <simple type declaration> (as opposed to a <parameterized type declaration>), then each object of type T will have the same number of components. In either event the number of components of an array cannot change after the array has been created.

The following terminology will be useful below. The type identified in the <type reference> is called the component type. The elements of the parenthesized list are called the bounds ranges. A data object whose type is an array type is called an array.

Each bounds range must either be the identifier of a declared RANGE (see Section 5.7) or comprise two <expression>s each of which is either of integer type or of an ordered enumeration type. In the latter case, the X_RANGE exception is raised if the value of the second <expression> is less than the value of the first.

Each object of an array type consists of an n-dimensional array of objects which are of the component type, where n is the number of bounds ranges. The number of components of the array is the product of the sizes of the bounds ranges.

Grouping is the only attribute of non-parameterized array types.

If GROUPED is specified in the declaration of an array type, then each component is grouped (see Section 5.8).

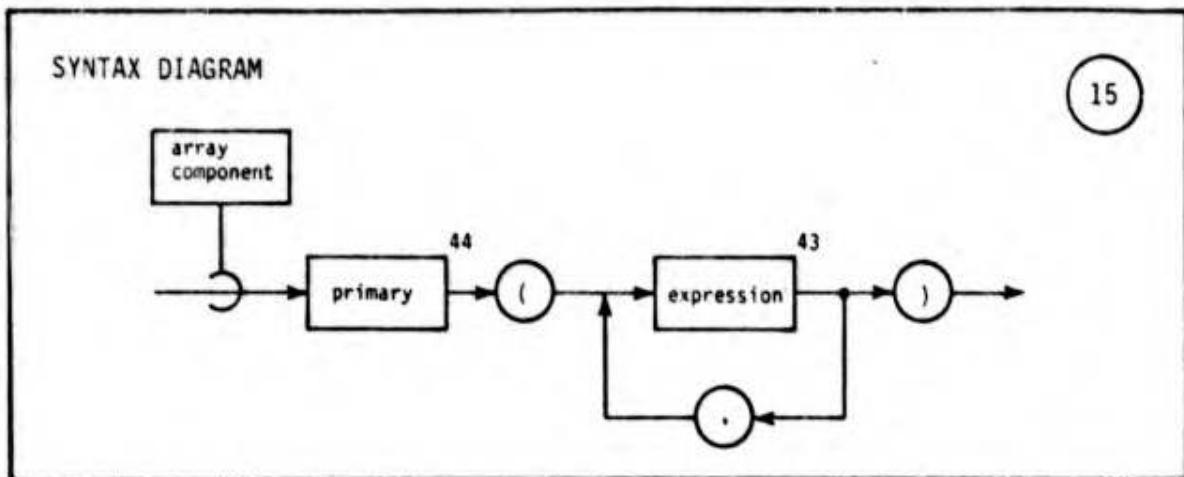
5.5.2.2 Routines. If X is an object having an array type with n dimensions, then LOW(X,i) and HIGH(X,i) return the low-bound and high-bound of the ith dimension of X. SIZE(X,i) returns the number of elements in the ith dimension; thus SIZE(X,i) = HIGH(X,i)-LOW(X,i)+1 when the ith dimension is an integer range. The parameter i must be a compile-time-evaluatable integer <expression> in the range 1..n.

If X's type is one-dimensional, then LOW(X), HIGH(X), and SIZE(X) may be written instead of LOW(X,1), HIGH(X,1), and SIZE(X,1).

SIZE always yields an integer value. LOW and HIGH return either integer or enumeration values, depending on the declaration of the array type.

A catenation operator, CAT, is defined for one-dimensional arrays. This is described below in conjunction with array slicing.

5.5.2.3 Component Selection.

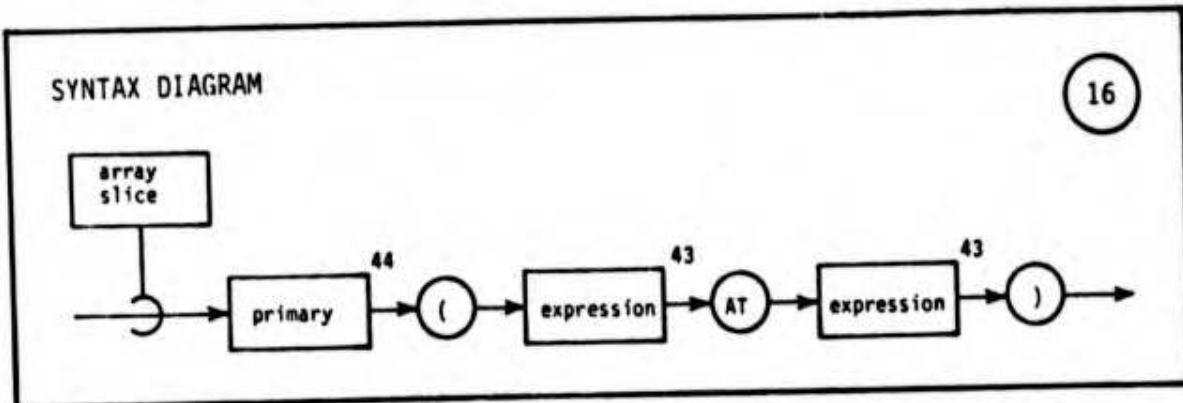


The primary must be of some array type T, and the number of <expression>s must be the same as the number of dimensions in the declaration of T.

Let X be the primary; the value of the i^{th} <expression> must be in the range $\text{LOW}(X,i).. \text{HIGH}(X,i)$, or the $X_{\text{SUBSCRIPT}}$ exception is raised.

An array component is a W-valued object if and only if the primary is W-valued. The type of the component is T's component type.

5.5.2.4 Array Slicing and Catenation.



The primary must be of some one-dimensional array type T.

Let X be the primary. The value of the first <expression> must be a non-negative integer, say N, and the value of the second <expression> must be an integer or enumeration element I such that $\text{LOW}(X) \leq I$ and $I+N \leq \text{HIGH}(X)$ (or $\text{FIXED}(I)+N \leq \text{FIXED}(\text{HIGH}(X))$, if I is an enumeration element); otherwise, the X_SUBSCRIPT exception is raised.

Conceptually, the result of an array slice $X(N \text{ AT } I)$ is an array comprising the components $X(I), X(I+1), \dots, X(I+N-1)$; if $N=0$, the result is an "empty" array. The slice is W-valued if and only if the primary array is W-valued.

The CAT operator has its conventional interpretation of catenation and is defined for all one-dimensional array types. In the interests of efficiency (see also Ironman 3-3E) REDL restricts the usage of array slicing and catenation such that

- (1) array slices may only be used as the source or target of an <assignment statement> or as an operand to CAT, and
- (2) the CAT operation may only be used as the source of an <assignment statement> or as an operand to another CAT.

Conceptually, the effect of $X := A_1 \text{ CAT } A_2 \dots \text{ CAT } A_n$ is to copy A_1 into the first $\text{SIZE}(A_1)$ components of X , then to copy A_2 into the next $\text{SIZE}(A_2)$ components of X , and so on. If the sum of the sizes of the A_i differs from the size of X , the x_SIZE exception is raised. If any A_i is a slice of X , the effect of the assignment is as though this slice is saved (in a "temporary") before any change is made to X .

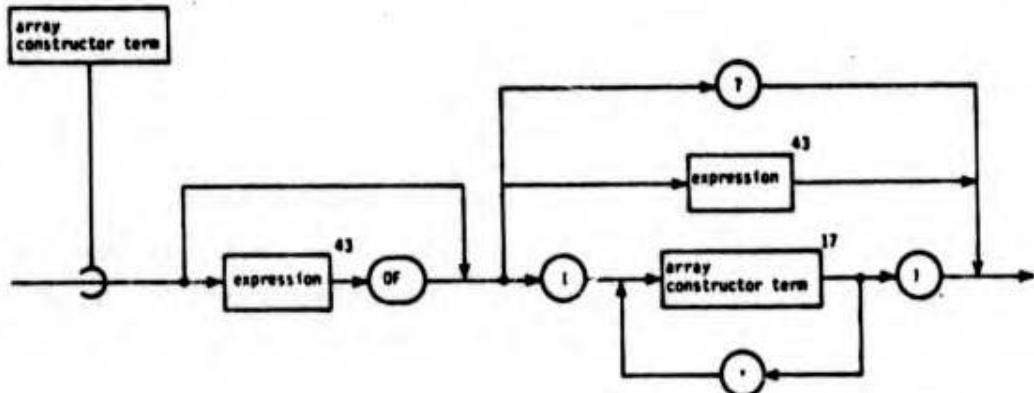
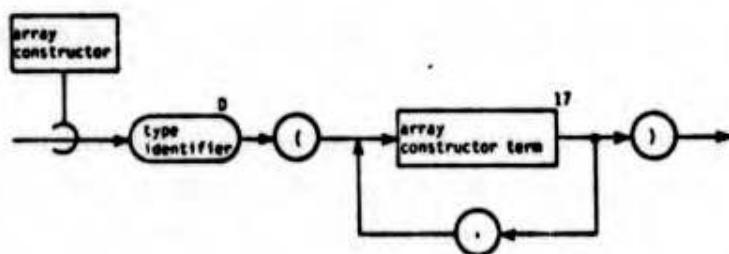
Examples:

```
TYPE T: ARRAY(1..10) OF FIXED(-100..100);
VAR X,Y: T INIT T(1,2,3,4,5,6,7,8,9,10);
Y(6 AT 2) := X(2 AT 1) CAT X(3 AT 8) CAT X(1 AT 6);
<*Now Y has the value T(1,1,2,8,9,10,6,8,9,10)*>
X := Y(5 AT 1) CAT X(5 AT 1);
<*Now X has the value T(1,1,2,8,9,1,2,3,4,5)*>
Y(3 AT 1) := X(2 AT 2) CAT X(2 AT 5);
<*Illegal, since the sizes of source and target differ*>
X(3 AT 6) := X(3 AT 5);
<*Now X has the value T(1,1,2,8,9,9,1,2,4,5)*>
```

5.5.2.5 Array Construction.

SYNTAX DIAGRAM

17



An *array constructor* builds an array having the type specified by the type identifier, out of components given by the *array constructor term*s.

Example:

```
TYPE T1: ARRAY(1..2,1..3) OF FIXED(0..10);
VAR X1; T1 INIT T1([10,2 OF I+J],[3 OF ?]);
```

If "expression OF" is present in an *array constructor term*, then the *expression* must be an integer (perhaps run-time determinable) and its value indicates how many times to replicate the item following "OF".

The component *expression*s must each be assignable to an object which is of the array component type. The use of "?" is restricted to initializations (i.e., object declarations and calls to NEW).

Examples:

```
TYPE T2: ARRAY(1..10) OF BOOLEAN;
VAR X2: T2 INIT T2(TRUE, 5 OF FALSE, 4 OF ?);
TYPE T3: ARRAY(1..2,1..3,1..4) OF CHAR;
VAR X3: T3 INIT T3([[4 OF ^A^],[^B^,^C^,^D^,^E^], ?],
[[^I^,3 OF ^J^],2 OF [^K^,^L^,^M^,^N^]]);
```

The conventions regarding the use of square brackets in *array constructor term*s are as follows. (We have assumed that all "OF" replication clauses have been expanded into the appropriate number of items. That is, we regard "<expression of item>" as simply "item,...item" (n times, where n is the value of the *expression*).) Assume that T is an N-dimensional array type,

with dimension sizes S_1, S_2, \dots, S_N . The form for any constructor for T is

$T(item_1, \dots, item_M)$

where $M = S_1$. The form for each of these items is "?" or

$[item_1, \dots, item_M]$

where $M = S_2$. The form for each of these items is "?" or

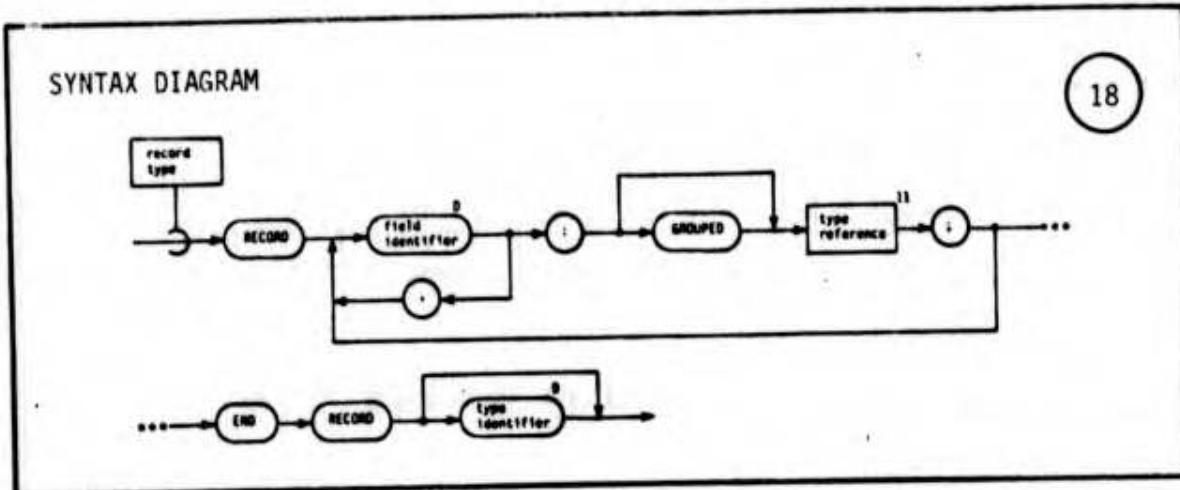
$[item_1, \dots, item_M]$

where $M = S_3$, and so on. The last level is simply "?" or

$[item_1, \dots, item_M]$

where $M = S_N$ and each item is either "?" or an <expression> having T 's component type. Thus the number of levels of square brackets is $N-1$ and each square-bracketed list contains a number of items equal to its dimension size.

5.5.3 Record.



Example:

```
TYPE T: RECORD
  A,B,C: GROUPED BOOLEAN;
  D: FIXED(0..1000);
  E: STRING(20);
END RECORD T;
```

T is a record type.

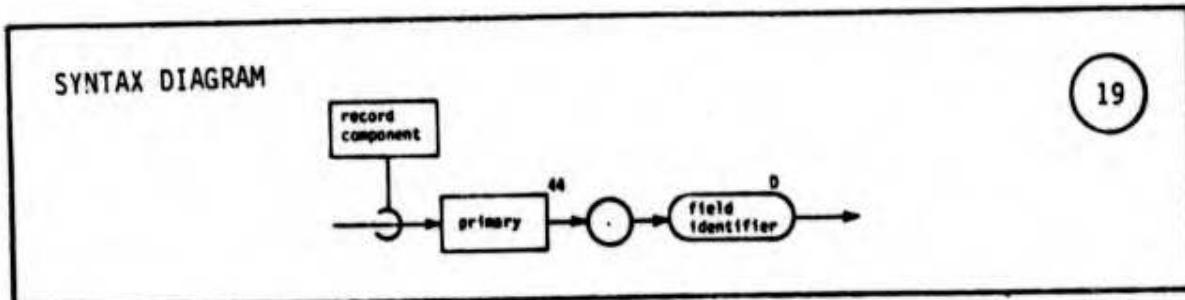
5.5.3.1 General Properties. The record type generator in REDL, as in Pascal, is used for declaring aggregate types whose objects are "heterogeneous" (i.e., different components possibly differing in type and/or representation).

The types identified in the <type reference>s are called the component types. A data object whose type is a record type is called a record. Each record consists of a collection of components, called fields. A field may be referenced by qualifying the record reference with a field identifier. No two field identifiers from the same record type may be the same.

The only attribute of a non-parameterized record type is the grouping attribute.

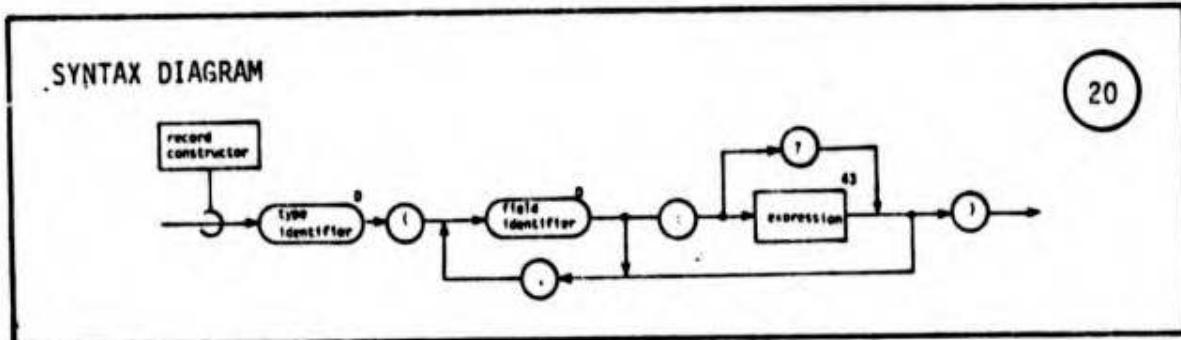
Each component for which GROUPED is specified is defined to be grouped.

5.5.3.2 Record Component Selection.



The <primary> must have some record type T such that the field identifier used for the record component is one of the field identifiers in the declaration of T. A record component is a W-valued object if and only if the <primary> is W-valued.

5.5.3.3 Record Construction.



The type identifier must be that of a record type; it has the effect of a constructor function which builds record objects out of components.

The field identifiers in the designated record type must be exactly the identifiers which occur preceding the colons in the comma-separated list. The effect of the construction is to build an object of the record type each of whose components is assigned the value of the corresponding <expression> in the comma-separated

list; the correspondence is established by matching the record field names with the identifiers preceding the colons. (In cases where the constructor is the source of an assignment or initialization, which will be their most frequent uses, the translator might choose not to actually construct a new object but instead to perform a component-wise assignment to the target. This is true for the array, union and machine record constructors in addition to the one for records.)

It is illegal to use "?" except in initializations.

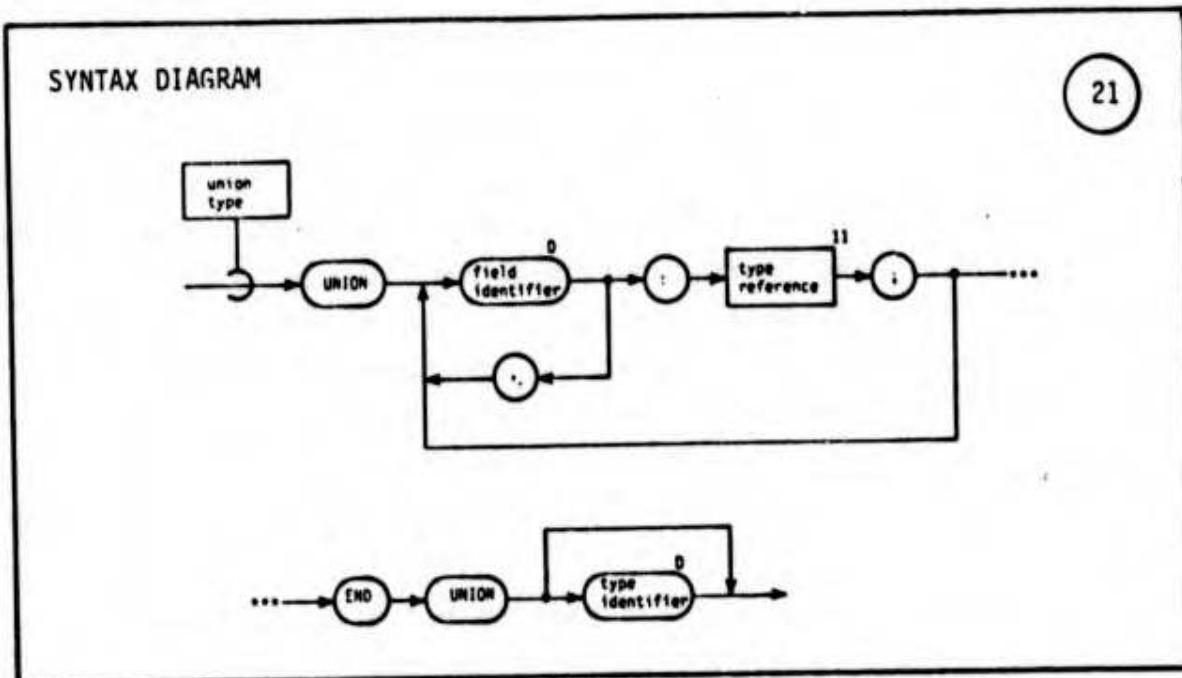
5.5.3.4 Examples.

```
TYPE T1: RECORD
  A: GROUPED BOOLEAN;
  B: GROUPED CHAR;
END RECORD;

TYPE T2: RECORD
  A: GROUPED T1;
  B: GROUPED CHAR;
  C: FIXED(0..1000);
END RECORD;

VAR V: T2 INIT T2(A: T1(A:TRUE,B:^Z^), B: ^CR^,C:?);
V.A := T1(A: FALSE, B: ^Y^);
```

5.5.4 Union.



Example:

```
TYPE T: UNION
  A: FIXED(0..1000);
  B: FLOAT(5,-1E4..1E4);
END UNION;
```

T is a union type.

5.5.4.1 General Properties. A union type corresponds to a Pascal record type consisting solely of a variant part. Basically, union types are used for representing data which can take on values from different types at different times. At any instant, a conceptual field called the tag indicates which one of the alternatives is present.

The types identified in the <type reference>s are called the component or alternative types. A data object whose type is a union type is called a union. The behavior of unions can be explained by contrast with records. If X is a record, then X consists of as many components as there are field identifiers in the declaration of X's type; any component of X can be selected by qualifying X with the appropriate field identifier. In contrast, if X is a union, then at any instant X comprises exactly two components. One component, the tag, has the effect of a constant component (modifiable only through an assignment to X) and specifies which of the union field identifiers is applicable for selecting from X. The second component (also called an alternative of X) is an object which can be obtained by qualifying X with the field identifier given by the tag.

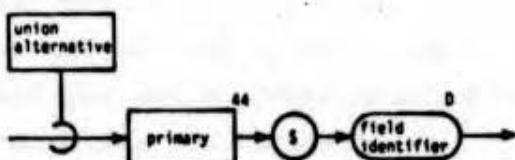
The <select statement> (see Section 9.5.2) may be used to interrogate (at run-time) the value of the tag.

Grouping is the only attribute of non-parameterized union types.

"Variant records" as in Pascal may be obtained by declaring a record type, one of whose components has a union type.

5.5.4.2 Union Component Selection.

SYNTAX DIAGRAM

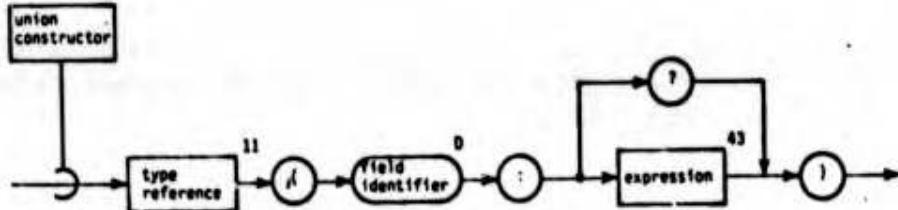


22

The <primary> must have some union type T such that the field identifier used for the union component is one of the field identifiers in the declaration of T, and such that the tag of the <primary> indicates this identifier. If any other field is indicated by the tag, the X_TAG exception is raised.

5.5.4.3 Union Construction.

SYNTAX DIAGRAM



23

The <type reference> must be that of a union type; it has the effect of a constructor function which builds an object having a particular alternative. For any non-parameterized union

type T, the <type reference> is simply the type identifier T itself. See Section 5.6.2 for further information on parameterized union types.

The field identifier must be one of the field identifiers of the designated union type. The effect of the construction is to build an object of the union type, whose tag indicates the field identifier present in the constructor, and whose alternative is assigned the value of the <expression> (or has an unpredictable value, if "?" appears).

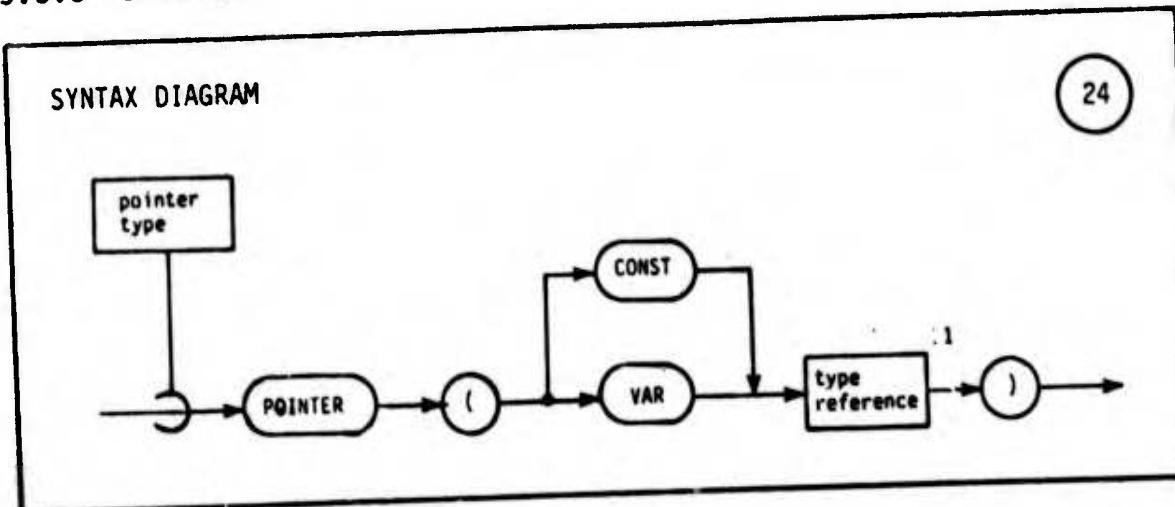
It is illegal to use "?" except in initializations.

5.5.4.4 Example.

```
TYPE U: UNION
    A: BOOLEAN;
    B: FIXED(0..100);
END UNION;

VAR V1: U INIT ?;
VAR V2: U INIT U(A: TRUE);
VAR V3: U INIT U(B: ?);
V1 := V2; /*Now V1 has an A-alternative with value TRUE*/
V3$B := 5;
V1 := V3; /*Now V1 has a B-alternative with value 5*/
V2$B := 0; /*The X_TAG exception will be raised, since V2
/*has an A-alternative*/
```

5.5.5 Pointer.



Examples:

```
TYPE PSTRING: POINTER(CONST STRING(*));
TYPE PLIST: POINTER(VAR LIST);
TYPE LIST: RECORD
    HEAD: FIXED(0..1000);
    TAIL: PLIST;
END RECORD;
```

PSTRING and PLIST are pointer types.

5.5.5.1 General Properties. Pointer types provide a facility for creating objects of dynamic storage class, i.e., objects whose lifetimes are not dependent on the lexical scoping of the <program>. Recursive, shared, and interconnected data structures may be defined via pointer types.

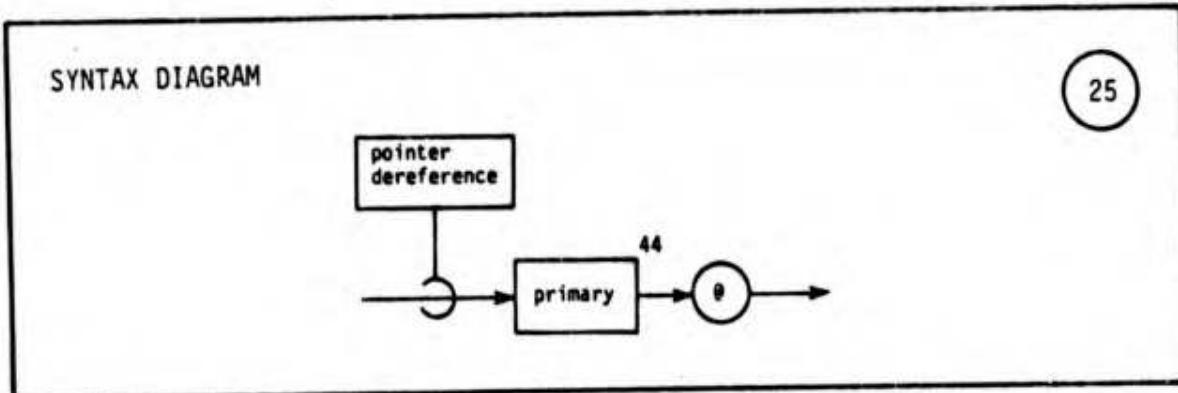
The main difference between REDL and Pascal with respect to pointers is that REDL requires the user to indicate explicitly, with the declaration of the type, whether the dynamic objects will be constants or variables. Thus in REDL one may have either constants or variables pointing at either (dynamic) constants or (dynamic) variables.

The type identified in the <type reference> is called the pointed-to type. A data object whose type is a pointer type is called a pointer.

Grouping is the only attribute of pointer types.

5.5.5.2 Literals. The identifier NIL denotes an R-valued object which belongs to every pointer type. NIL has the property that it points to no object.

5.5.5.3 Pointer Dereference.



The <primary> must be of some pointer type T. If X is the <primary>, then the type of X@ is T's pointed-to type. X@ is W-valued if T is declared with VAR preceding the pointed-to type, and X@ is R-valued if CONST appears there. In the first case, X is said to point to a dynamic variable: in the second case, a dynamic constant. Alternatively, we say that the pointer dereference X@ itself is a dynamic variable (or a dynamic constant).

If X has the value NIL, the pointer dereference X@ will raise the X NIL POINTER exception.

A dynamic object comes into existence through a call on the procedure NEW, and it ceases to exist as soon as no pointer points to it. If X is a pointer, and no other pointer points to the dynamic object X@, then any of the following situations will cause X@ to cease existing:

- (1) X is the target of an assignment.
- (2) Control is transferred out of the scope in which X is declared (if X is a static object).
- (3) A <call statement> invokes FREE(X).

We note that, as a space optimization, an implementation may reuse the storage occupied by a dynamic object which has ceased existing.

5.5.5.4 Routines.

(1) Relationals. The only relational operators defined for pointer types are = and #. If X and Y are objects of the same pointer type, then X=Y is true if and only if X and Y have the same pointer values (i.e., point to the same object). It may be the case that X@=Y@ but that X#Y.

(2) NEW. The built-in procedure NEW takes two parameters: a W-valued object p having any pointer type T, and a value v of the same type as T's pointed-to type. The effect of NEW(p,v) is to create a new dynamic object such that, after the call, p@=v. Thus p will point to a dynamic object (either a constant or a variable, depending on T's declaration) whose initial value is v. To obtain an uninitialized dynamic variable, the user may specify "?" for v (if the <type reference> for p's pointed-to type is not unresolved) or in a constructor which builds v.

Example:

```
TYPE PVSTRING: POINTER(VAR STRING(*));
VAR p: PVSTRING INIT NIL;
CALL NEW(p, 'ABCDE'); <*p@ = 'ABCDE'*>
CALL NEW(p, 'DEF'); <*p@ = 'DEF'*>
CALL NEW(p, STRING(10 OF ?)); <*p@ is an uninitialized STRING
                                <*of size 10*>
```

(3) FREE. The built-in procedure FREE takes a single parameter, a W-valued object p having any pointer type T. The effect of FREE(p) is as follows:

(a) If there is no other pointer q such that p=q, then p is set to NIL and the storage formerly occupied by p@ is made available for reuse.

(b) If there is some other pointer q such that p=q, then the X_NON_FREEABLE exception is raised.

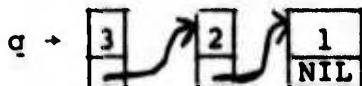
It may be noted that the checking for these conditions necessitates a scheme such as "reference counts" for keeping track of dynamic storage, unless the checking code for the exception is suppressed (see Section 11.4).

5.5.5.5 Examples.

```
TYPE PSTRING: POINTER(CONST STRING(*));
TYPE PLIST: POINTER(VAR LIST);
TYPE LIST: RECORD
    HEAD: FIXED(0..1000);
    TAIL: PLIST;
    END RECORD;

VAR p1,p2: PSTRING INIT ?;
CALL NEW(p1, 'ABCDE');
CALL NEW(p2, 'FGH');
p1 := p2; <*Now p1@=p2@='FGH'*>
p1@ := 'ABC'; <*Illegal, since p1@ is a constant*>
VAR q: PLIST INIT NIL;
FOR I FROM 1 UPTO 3 DO CALL NEW(q, LIST(HEAD:I, TAIL:q)); END FOR;
```

The effect of the <for statement> is to establish the following linked list:



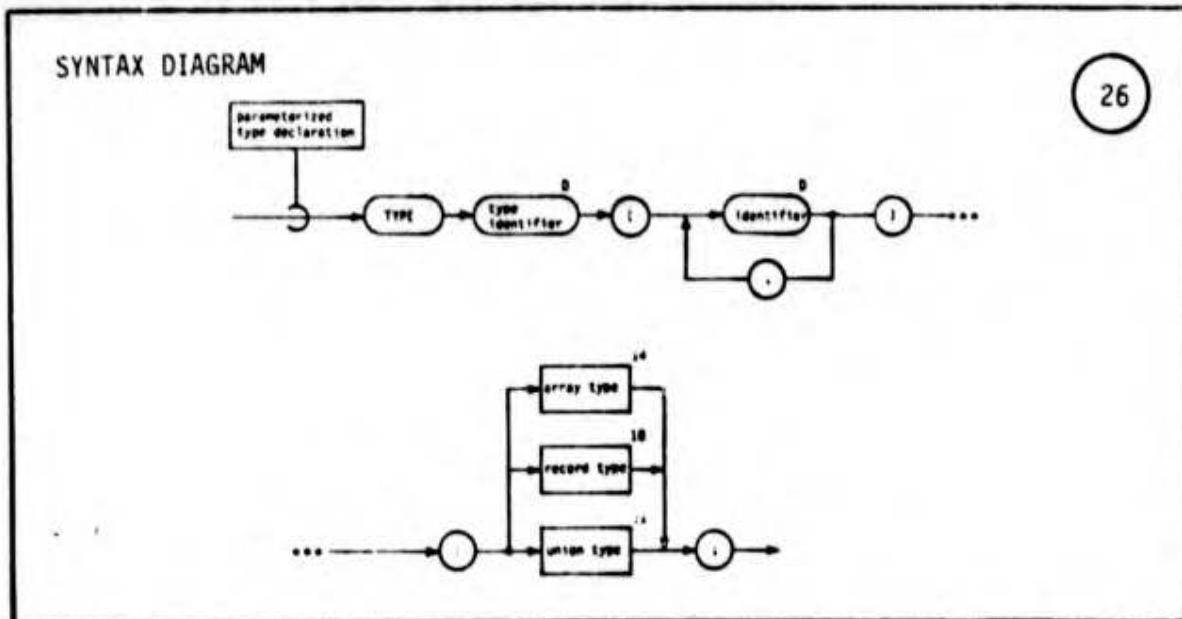
The list can be made circular by the following:

```
BEGIN
VAR qtemp: PLIST INIT q;
WHILE qtemp@.TAIL#NIL DO qtemp := qtemp@.TAIL; END WHILE;
qtemp@.TAIL := q;
END;
```

5.6 Parameterized Types.

In order to allow the user to write routines which take array parameters whose sizes may vary from one call to another (or which take records or unions containing such arrays), REDL provides for the declaration and use of parameterized types. In this section we describe the semantics of parameterized types, both built-in (STRING and BITS) and user-defined.

5.6.1 The Declaration of Parameterized Types.



Each identifier in the parenthesized list is said to be a formal parameter to the type. Each formal parameter may appear within the <array type>, <record type>, or <union type> in the following contexts (and only in these contexts):

- (1) as a low-bound or high-bound of one or more bounds ranges, if the type being parameterized is an <array type>, or
- (2) as an actual parameter in a <type reference> appearing in the <array type>, <record type>, or <union type>.

The following are examples of <parameterized type declarations>:

```
TYPE SQUARE_MATRIX(N): ARRAY(1..N,1..N) OF FLOAT(5,-1.0..1.0);
TYPE VARYING_STRING(N): RECORD
    CUR_SIZE: FIXED(0..MAX);
    DATA: STRING(N);
END RECORD;
TYPE T(M,N): ARRAY(M..N) OF GROUPED CHAR;
```

5.6.2 The Creation of Objects Having Parameterized Types.

When an object of a parameterized type is to be created, actual parameters must be supplied for each formal parameter to the type. These actual parameters must be integer-valued <expression>s (not necessarily compile-time computable), and their values will determine values for attributes of the type in a manner to be described below.

As an example using the type declarations from the preceding section, consider the following:

```
VAR X: SQUARE_MATRIX(2) INIT SQUARE_MATRIX([0.1,0.2],[0.3,0.4]);
VAR Y: VARYING_STRING(10) INIT VARYING_STRING(CUR_SIZE:5,
    DATA:'ABCDE      ');
VAR Z: T(3,6) INIT T(^CR^,^LF^,^A^,^B^);
```

The attributes of a parameterized type are the type's formal parameters and the grouping attribute. The value of an attribute which is a formal parameter to the type can be obtained by using a qualification form with the formal parameter as a field identifier. (Conceptually, one may think of an object of a parameterized type as a record with separate fields for each formal parameter; in practice such a separate field may be unnecessary at run-time.) Thus in the above example X.N=2, Y.N=Y.DATA.SIZE=10, Z.M=3, and Z.N=6. If the parameterized type is a record type or a union type, the type formal parameter identifiers may not be the same as any of the field identifiers.

The object constructor forms for arrays, records, and unions are applicable when their types are parameterized. The only qualification is that when a union type is parameterized, the constructor must explicitly contain a <type reference> for the

union type such that actual parameters are supplied for the type formal parameters. For example, if there is a declaration:

```
TYPE U(M,N): UNION
    A: STRING(M);
    B: SQUARE MATRIX(N);
END UNION;
```

then the constructor $U(5,3)(A:'ABCDE')$ is legal, whereas the abbreviated form $U(A:'ABCDE')$ is not.

In the case of a parameterized array type, the bracketing of <array constructor term>s is used to determine the "shaping" of the constructed array into rows and columns. For example, with the declaration:

```
TYPE BMAT(M,N): ARRAY(1..M,1..N) OF BOOLEAN;
```

the <array constructor> $BMAT([TRUE,FALSE,TRUE],[FALSE,TRUE,FALSE])$ yields a 2-row by 3-column array, whereas $BMAT([TRUE,FALSE],[TRUE,FALSE],[TRUE,FALSE])$ is 3-by-2. The form $BMAT(TRUE,FALSE,TRUE,FALSE,TRUE,FALSE)$ is illegal, since a two-dimensional array is required for BMAT.

5.6.3 Unresolved Type References.

As described in Section 5.3.2, a <type reference> for a parameterized type may be unresolved (i.e., "*" may be used as actual parameters to the type) in a declaration of either a routine's formal parameter or a pointer type. An example of the latter is the type PSTRING from Section 5.5.5. As an example of the former, a procedure P may have the header:

```
PROCEDURE P(CONST C:SQUARE MATRIX(*), VAR V: VARYING_STRING(*),
            RESULT R: T(1,*));
```

5.6.4 Assignment and Parameter Passing.

The purpose of this section is to extend the rule given in Section 5.1.2 concerning assignment and routine parameter passing.

5.6.4.1 Resolved Type References.

(1) Assignment and routine parameter passing by CONST and RESULT are permitted if and only if the source and target have the same type and the same values for their type formal parameters. As an example, it is not permissible to assign a STRING(5) to a STRING(10): the sizes must be the same.

(2) VAR parameter passing is permitted if and only if the routine's formal and actual parameters have the same type and the same representation (i.e., the same values for each type formal parameter, and the same grouping). Thus the only difference between (1) and (2) is that for assignment the grouping of the two objects may differ.

5.6.4.2 Unresolved Type References. The rules are the same as for a resolved <type reference>, with the following additions.

(1) If a routine formal parameter has "*" for any type actual parameter, then this is said to be resolved by the corresponding attribute of the routine actual parameter. Thus, if a routine takes a STRING(*), either by CONST, VAR, or RESULT, then the actual parameter may be any object of type STRING (with the proviso that groupings must be the same if by VAR).

(2) If a routine formal parameter P whose <type reference> is unresolved is used in an assignment or as an actual parameter to another routine, then the legality of such use depends on the current values of P's type parameter attributes as established by the call of the original routine. For example, if P is a formal parameter declared as a STRING(*), and X is declared inside the routine as a STRING(10), then the assignment P := X is legitimate only if P.SIZE=10.

5.6.5 STRING.

5.6.5.1 General Properties. Conceptually, the built-in type STRING is declared as follows:

```
TYPE STRING(SIZE): ARRAY(1..SIZE) OF GROUPED CHAR;
```

The only properties which distinguish STRING from a user-defined type declared as above are (1) the presence of a special literal form, and (2) the presence of overloaded = and # operators.

5.6.5.2 Literals. The general form for STRING literals is given in Lexical Diagram H. For example, 'ABCDE' is a STRING literal of size 5; 'ABC^CR^D^^E' is a STRING literal of size 7, equivalent to the constructor STRING(^A^, ^B^, ^C^, ^CR^, ^D^, ^'^, ^E^). Thus, each CHAR in the literal is given by the corresponding string literal component itself, if this is a special enumeration symbol; otherwise it is the CHAR formed by surrounding the component with ^ symbols.

5.6.5.3 Equality and Inequality. The = and # operators are the only relational operators defined for STRING. In order for S1=S2 to be TRUE, it is necessary for S1.SIZE=S2.SIZE and for the objects to be componentwise the same.

5.6.5.4 Notes. Since SIZE is the name of the formal parameter to the STRING type, the form X.SIZE is used to obtain the number of CHARS contained in the STRING X. Also, since STRING is an array type, all the properties described in Section 5.5.2 are applicable; e.g., one may perform catenation or slicing on STRING objects. It may be noted that STRING assignment is permissible only when the sizes of source and target are the same, in consequence of the rules on parameterized types (Section 5.6.4).

5.6.6 BITS.

5.6.6.1 General Properties. Conceptually, the built-in type BITS is declared as follows:

```
TYPE BITS(SIZE): ARRAY(1..SIZE) OF GROUPED BOOLEAN;
```

The only properties which distinguish BITS from a user-defined type declared as above are (1) the presence of special literal forms, and (2) the presence of overloaded operators =, #, NOT, AND, OR, and XOR.

5.6.6.2 Literals. The forms for BITS literals are given in Lexical Diagram I and may be used for binary, octal, and hexadecimal literals. For example, `O'23'` denotes the same value as `B'010011'`, and `X'3E'` denotes the same value as `B'00111110'`. Thus, if the number of components of a BITS literal is n, then the value denoted will have size n, 3n, or 4n, depending on whether the literal was binary, octal, or hexadecimal. The value will have TRUE for those components which were binary 1 in the literal, and FALSE elsewhere.

5.6.6.3 Routines.

(1) Equality and Inequality. The = and # operators are the only relational operators defined for BITS. As with STRING, the SIZEs must be the same in order for equality to be TRUE.

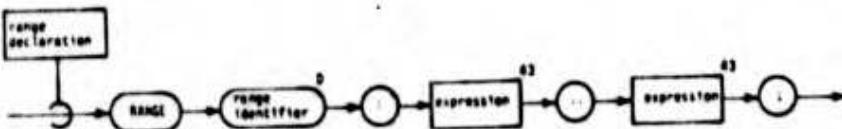
(2) Logical Operators. The prefix operator NOT and the infix operators AND, OR, and XOR are defined so that they work component-wise on their operand(s). Thus NOT `B'101'` is `B'010'`, and `B'0110' XOR B'1101'` is `B'1011'`. For the infix operators, the operands must have the same SIZE (otherwise the X_SIZE exception is raised).

5.6.6.4 Notes. Aside from the presence of the logical operators, BITS is analogous in its properties to the STRING type, and the comments in Section 5.6.5.4 are applicable. It may be noted that if the user wishes to perform a logical operation on BITS objects of different SIZES, this may be done by an explicit catenation of the shorter operand with a BITS value of appropriate SIZE and value. (The decision of "left justification" vs. "right justification" is thus made explicit.)

5.7 The Declaration of Ranges.

SYNTAX DIAGRAM

27



The declaration of a range allows the range identifier to be used in type references for FIXED, FLOAT, ordered enumeration types, and in array type declarations. The <expression>s must both have the same type, and this type must be either FIXED, FLOAT, or an ordered enumeration type. The <expression>s may be run-time evaluable; the X_RANGE exception is raised if the value of the second <expression> is less than the value of the first.

It should be noted that a range is not a type; i.e., it is illegal to use a range identifier where a type identifier is required. Thus ranges provide a shorthand notation without creating new types.

Examples:

```
RANGE R: 1..10;  
TYPE T: ARRAY(R) OF BOOLEAN;  
VAR V: FIXED(R) INIT 1;  
RANGE LETTER: ^A^..Z^;  
VAR C: CHAR(LETTER) INIT ^Q^;
```

5.8 The Grouping Attribute.

Each type has a grouping attribute; this allows the programmer to specify the tradeoff between data storage compactness and efficiency of access. The compile-time evaluable function GROUPED(X) is defined for each data object X and returns TRUE if X is grouped and FALSE if X is not grouped (i.e., ungrouped). "Grouped" implies economization of the data storage space at the possible expense of data access code and time, while "ungrouped" implies the inverse.

In declaring an aggregate type (i.e., an array, record, or machine-record type) the programmer may optionally specify GROUPED before any of the component types. Moreover, the components of aggregate type declarations are the only places where grouping may be specified. Thus all data objects are ungrouped except for those components of aggregates marked GROUPED in the declaration of the aggregate.

The grouping attribute applies to the component and not to the aggregate in which the component appears. However, since aggregate types can themselves be components of other aggregate types, grouping is applicable to any type.

The reason for the choice of the term "grouped" is that, if T1 is an aggregate type with a component declared as a GROUPED T2, then such a component may be grouped with other components in the same addressable storage unit. For example, if T is an array of GROUPED BOOLEANS, then an implementation may reduce data storage by grouping the components of T's objects one per bit in each word.

The only context in which differences in a type's grouping are relevant is in VAR parameter passage; in particular, an object which is GROUPED may not be passed as a VAR parameter.

As an example of the use of grouping, assume that the user declares an enumeration type COLOR:

```
TYPE COLOR: ENUM(RED,ORANGE,YELLOW,GREEN,BLUE,VIOLET);
```

and then declares two array types with COLOR as their component type.

```
TYPE GROUPED_COLOR_ARRAY: ARRAY(1..100) OF GROUPED COLOR;
```

```
TYPE UNGROUPED_COLOR_ARRAY: ARRAY(1..100) OF COLOR;
```

PACK and UNPACK procedures may be defined by the user to copy an UNGROUPED_COLOR_ARRAY to a GROUPED_COLOR_ARRAY and vice versa. We display the PACK procedure below; the UNPACK procedure is analogous.

```
PROCEDURE PACK(CONST UCA: UNGROUPED COLOR ARRAY,  
               VAR GCA: GROUPED COLOR ARRAY);  
   FOR I FROM 1 TO SIZE(UCA) DO GCA(I) := UCA(I); END FOR;  
END PROCEDURE PACK;
```

5.9 Type Opacity, "Lifting" and "Lowering".

In both simple and parameterized *<type declaration>*s (see Syntax Diagrams 11 and 26) the type being declared (say T) may be defined in terms of a *<generated type>*. In such a case T will automatically come equipped with the properties intrinsic to the individual type generator, such as array subscripting or record component selection. (With type encapsulation, described in Chapter 6, these properties can be "hidden" so that they may only be used within the encapsulation.)

Alternatively, in the case of a *<simple type declaration>* for T, T may be defined in terms of a *<type reference>* for a type T₁. For example,

```
TYPE SMALL_INT: FIXED(0..127);
```

In such a case T and T₁ are distinct types (this follows from Section 5.3.3), and none of the properties of T₁ is automatically inherited by T. We say that types are thus opaque. If the user wishes to define for T some of the routines which are defined for T₁, then the names of such routines must be overloaded (see Section 7.1.3). The type identifiers may be used as explicit type-transfer functions to convert between T and T₁: T(t₁), where t₁ has type T₁, is a value of type T (a "lift"), and T₁(t), where t has type T, is a value of type T₁ (a "lower").

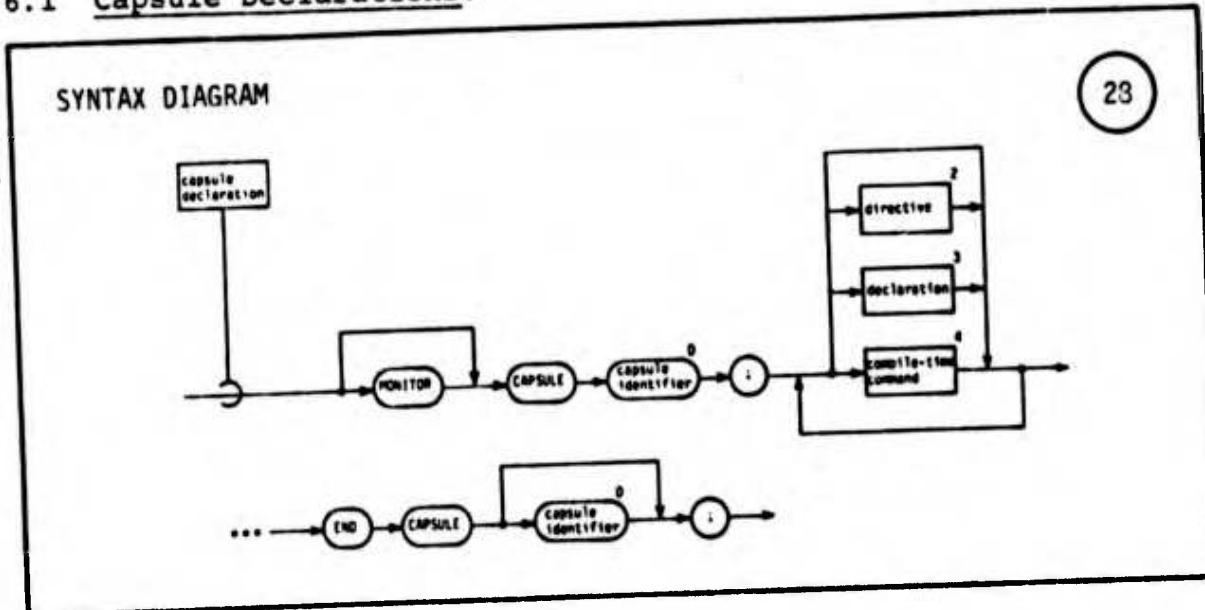
Example:

```
OVERLOAD OPERATOR + (I,J:SMALL_INT) RESULT K:SMALL_INT INIT ?;
  K := SMALL_INT((FIXED(I)+FIXED(J)) MOD 128);
END OPERATOR +;
```

6.0 CAPSULES

REDL's data abstraction facility is realized in a namescoping mechanism called the capsule, which provides for encapsulated definitions as required by Ironman 3-5A through 3-5D.

6.1 Capsule Declarations.



A capsule is a closed scope consisting of <directive>s and <declaration>s. (Any <compile-time command> immediately contained in a <capsule declaration> must produce <declaration>s and <directive>s, analogous to the convention for <program>s.) The elements declared within a capsule are called capsule elements; they come into existence when control enters the scope in which the capsule is declared. The capsule may be regarded as a "fence" around its elements. The essential property of the capsule is that it allows a precise specification of this fence's transparency. The <import directive> (Section 4.2.3) mentions all non-pervasive names declared outside the capsule that may be referenced inside. The <export directive> mentions all types, routines and exceptions declared within the capsule that may be referenced outside the capsule. This facility provides an effective means to make available selectively those capsule elements that represent an intended abstraction, and to hide those elements that are considered as details of implementation.

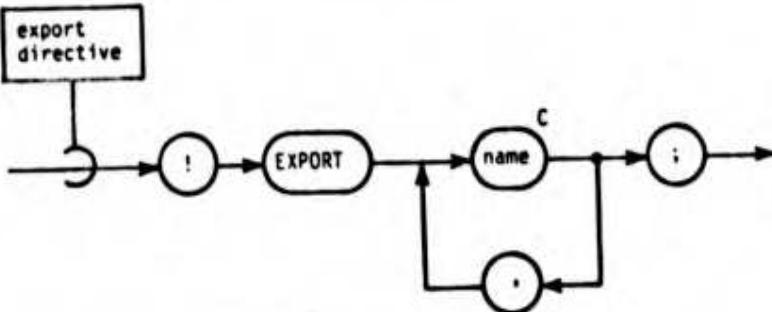
A capsule encapsulates those parts that are non-essential to the remainder of a <program>, or that are even to be protected from inadvertant access. Capsules may be nested.

A monitor capsule is a capsule declared with the initial token MONITOR. Monitor capsules are used to protect accesses to shared data by separate parallel paths. They are described in Section 12.3.

6.2 Export Directive.

SYNTAX DIAGRAM

29



The <export directive> names those types, routines, and exceptions which are declared within but usable outside a capsule. A name occurring in the <export directive> is said to be exported. If a type is exported, then only the type's identity, but none of its structural details, becomes known outside--this is the essence of the capsule's usefulness as a data abstraction facility. For example, if an exported type T is a record type, then the field names and constructor form remain unknown; moreover, outside the capsule it cannot be determined whether T is a record type, an array type, or an enumeration type.

By exporting a name, a capsule makes that name available for use in the scope which immediately contains the capsule.

It is not permissible to export data elements from a capsule. Thus, the names of such elements may only be referenced from inside the capsule.

6.3 The Lifetime of Capsule Data.

A capsule may contain declarations of constants and variables. It must be emphasized that the capsule is the scope of accessibility, but not the scope of allocation, for such data. That is, although these data elements can only be referenced from inside the capsule, their storage is allocated with the local data of the scope in which the capsule is immediately contained. The capsule's local data elements thus come into existence (and are initialized) when control is transferred into that containing scope, retain their values between invocations of the capsule's routines, and vanish when control is transferred out of the containing scope. (In the event of nested capsule declarations, the scope containing the outermost capsule declaration is the one in which the lifetimes of the capsules' local data are established.)

6.4 Capsule Example.

The <capsule declaration> below illustrates how one might define the behavior of a push-down stack of integers represented in an array. (In Section 14.4 we show how the compile-time facilities may be used to obtain a "stack generator" which produces stacks of elements having any specified type.)

```
CAPSULE STACK;
!EXPORT STACK_INIT, PUSH, POP, STACK_OVERFLOW, STACK_UNDERFLOW;
!CONST STACK_SIZE: FIXED INIT 100;
TYPE STACK REP: ARRAY(1..STACK_SIZE) OF FIXED(MIN_INT..MAX_INT);
VAR S: STACK REP INIT ?;
VAR TOP: FIXED(0..100) INIT ?;
EXCEPTION STACK_OVERFLOW, STACK_UNDERFLOW;
PROCEDURE STACK_INIT;
    !IMPORT TOP;
    TOP:=0;
END PROCEDURE STACK_INIT;
PROCEDURE PUSH(CONST C: FIXED(MIN_INT..MAX_INT));
    !IMPORT S, TOP, STACK_SIZE, STACK_OVERFLOW;
    IF TOP = STACK_SIZE
    THEN RAISE STACK_OVERFLOW;
    ELSE TOP:=TOP+1;
        S(TOP):=C;
    END IF;
END PROCEDURE PUSH;
PROCEDURE POP(RESULT R: FIXED(MIN_INT..MAX_INT) INIT ?);
    !IMPORT S, TOP, STACK_UNDERFLOW;
    IF TOP = 0
    THEN RAISE STACK_UNDERFLOW;
    ELSE R:=S(TOP);
        TOP:=TOP -1;
    END IF;
END PROCEDURE POP;
END CAPSULE STACK;
```

The following might then appear in the scope in which the above <capsule declaration> appears:

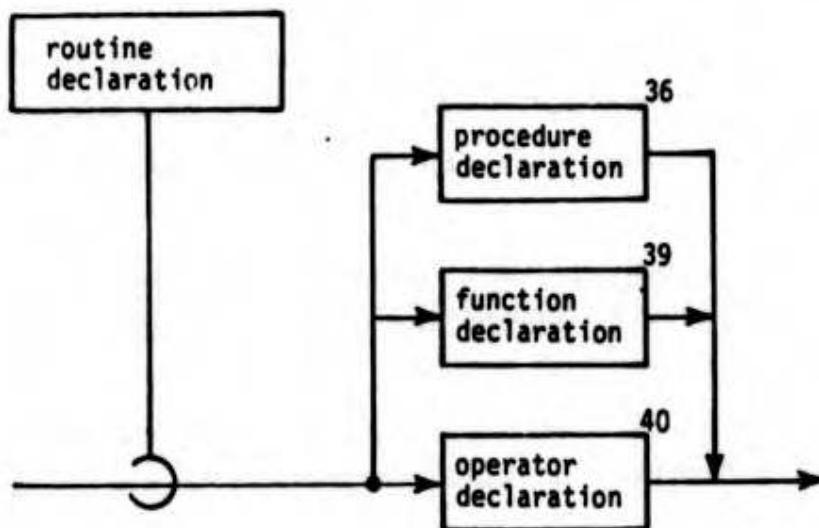
```
VAR J: FIXED(MIN_INT..MAX_INT) INIT ?;
CALL STACK_INIT;
CALL PUSH(3);
CALL PUSH(7);
CALL POP(J); <*J NOW HAS THE VALUE 7*>
```

7.0 ROUTINES

7.1 Introduction

SYNTAX DIAGRAM

30



A <routine declaration> serves to define a part of a <program> and to associate a name with it so that it can be activated by a <call statement>, a <function invocation>, or an operator invocation, depending on whether the routine is a procedure, a function, or an operator, respectively. A <routine declaration> declares the routine's name (the name following the token PROCEDURE, FUNCTION, or OPERATOR) in the scope which immediately contains the declaration.

7.1.1 Comparison with Pascal.

The major points of difference between REDL and Pascal are the following:

- (1) REDL allows the names of routines to be overloaded; all built-in operators are overloadable and thus can be extended to user-defined types.
- (2) REDL contains three parameter binding classes: CONST (input), VAR (by reference) and RESULT (copy out).
- (3) There are language-enforced restrictions in REDL on side effects in functions and operators, and on aliasing.
- (4) REDL contains a variety of <directive>s for optimization purposes.
- (5) In REDL, routines are closed scopes (see Section 4.2.4) and thus must explicitly import any non-pervasive program elements which are needed from outer scopes.
- (6) Unlike Pascal, REDL does not restrict to scalar types the result of a function or input parameters to routines.

7.1.2 Overloading Routines.

A routine may be declared as overloaded; this is accomplished by coding the token OVERLOAD before PROCEDURE, FUNCTION, or OPERATOR. If R is the routine name for an overloaded <routine declaration>, then it is permissible to supply additional declarations for R in scopes in which the original declaration is known. Each such additional declaration is said to be an overload line (or simply a line) for R. The only constraints on the set of declarations are:

(1) Each one must be an overloaded <routine declaration> of the same kind (i.e., all procedures, all functions, or all operators).

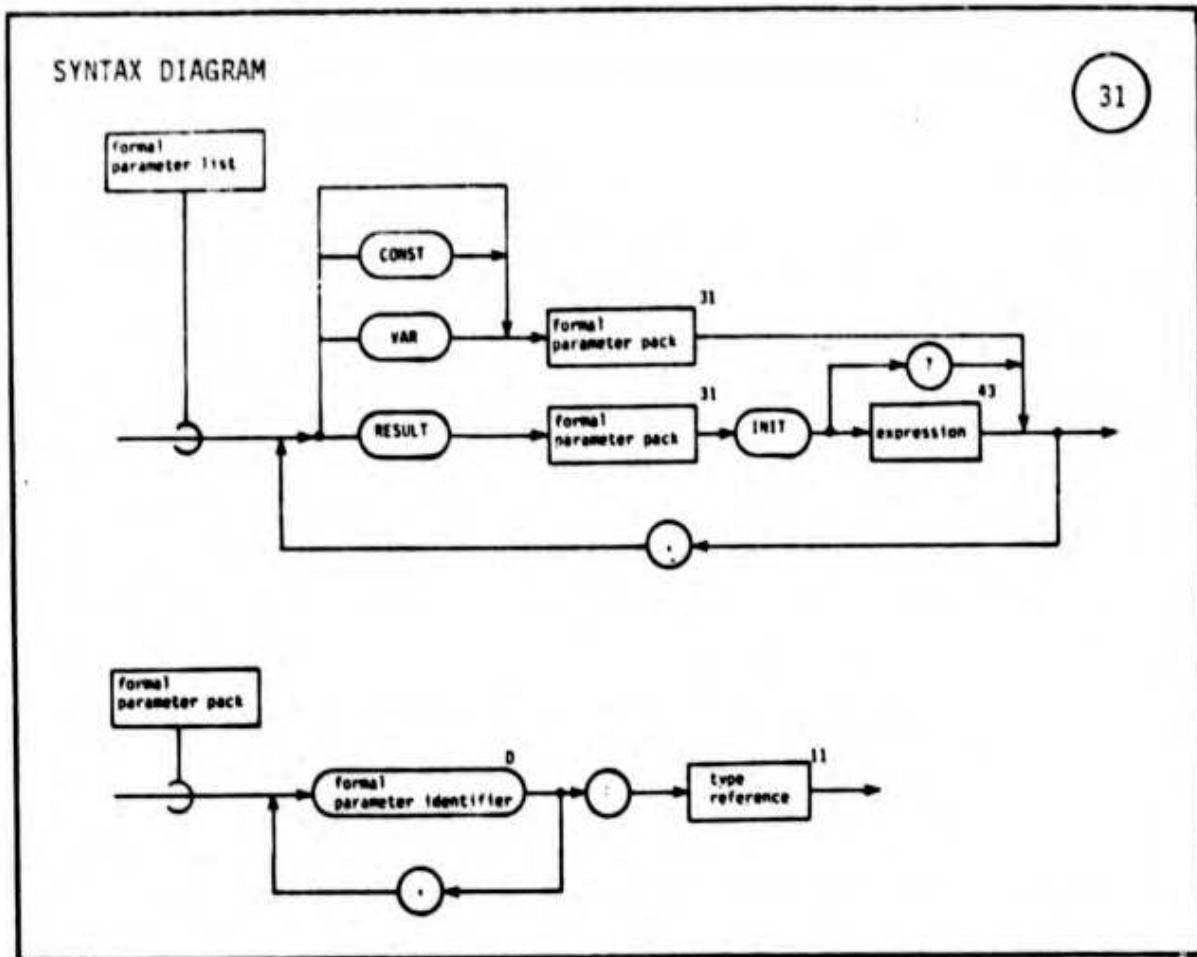
(2) For any two declarations in the set, if they have the same number of formal parameters, then, pairing the formal parameters in sequence in the two formal parameter lists, there must be at least one pair with distinct types.

The first constraint aids readability by revealing the programmer's intent to declare the routine as overloaded. The second constraint enables the translator to determine which routine is intended for any given invocation.

7.1.3 Invocation and Return.

Routines are brought into execution (i.e., invoked) in three ways. Procedures are invoked via <call statement>s, functions via <function invocation>s, and operators through operator invocations (operator/operand combinations in <expression>s). Control may leave a routine normally (through execution of an <exit statement> naming the routine, or by "dropping out the bottom") or abnormally (if an exception is raised in the routine but not handled there). The normal return from a procedure causes the <statement> following the <call statement> to be executed, and the normal return from a function or operator produces a value to be used in the <expression> from which the function or operator was invoked.

7.1.4 Formal Parameters and Actual Parameters.



The <formal parameter list> (and the result variable, for functions and operators) specify the interface between the point of invocation and the <body> of the routine. Each formal parameter is declared in the scope of the routine. The binding class of the parameter is either CONST, VAR, or RESULT, depending on which of these precedes the <formal parameter pack> containing the parameter. If no binding class is given, the default is CONST. The type of the parameter is specified in the <type reference> in the <formal parameter pack>.

When a routine is invoked, evaluation of a set of actual parameters (also known as arguments) supplied at the point of invocation yields R-valued or W-valued objects to be used in place of the formal parameters during execution of the routine. For each routine other than the short-circuited operations, the actual parameters are evaluated in undefined order, prior to routine invocation. The resulting R- and W-valued objects are bound to the corresponding formal parameters; there must be an equal number of formals and actuals. The rules for parameter passing appear in Sections 5.1.2 and 5.6.4. The semantics for the three binding classes is as follows.

(1) The CONST binding class specifies "by value" (i.e., "copy in") binding with the restriction that the formal parameter may not appear in any context in which its value could be changed (i.e., it behaves like a local CONST datum). Thus, a CONST formal parameter may not appear as the target of an <assignment statement>, and may not be passed as a VAR or RESULT parameter to a procedure. We note that a compiler can choose (for efficiency reasons) to implement CONST binding as a call "by reference" as opposed to "copy" provided that it can guarantee that the effect of the routine is as though the parameter were passed by copy---viz., that the value of the actual parameter remains unchanged during execution of the routine.

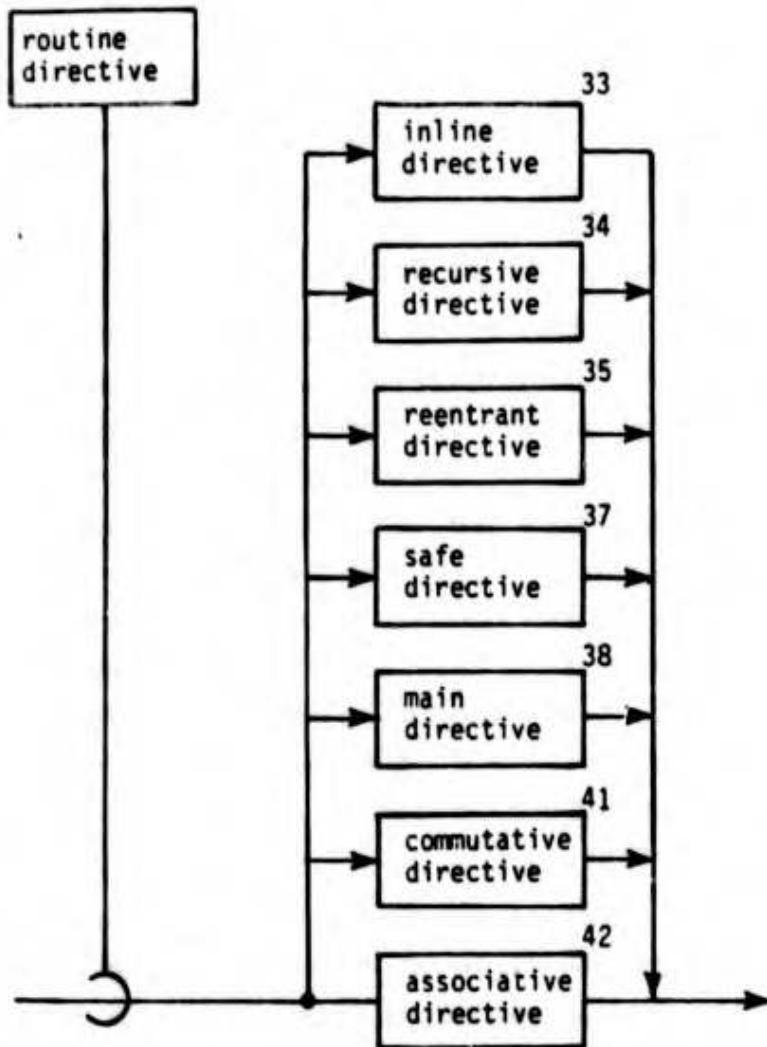
(2) The VAR binding class specifies "by reference" binding, as illustrated by the VAR class of Pascal. During routine execution, a reference to the formal parameter is a reference to the actual parameter. The actual parameter must be a W-valued object.

(3) The RESULT binding class specifies "copy out" binding. When a routine is invoked, a local object is created for the formal parameter. Any use of the formal parameter is a use of the local object. Upon normal termination of the routine, the value of the formal parameter is assigned to the actual parameter, which must be a W-valued object. Upon abnormal termination (i.e., the raising of an exception which is not handled in the routine), no such copy occurs.

7.1.5 Routine Directives.

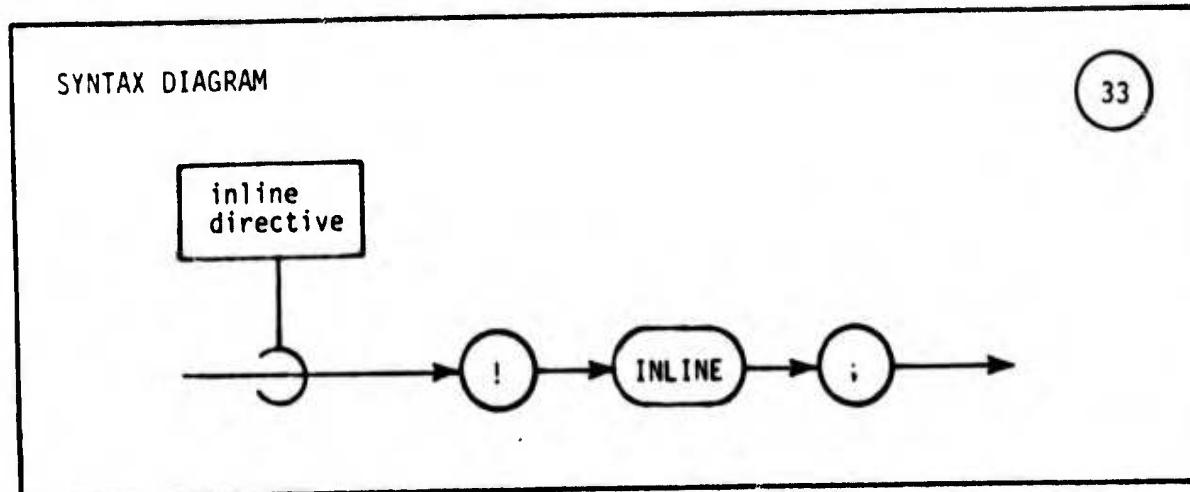
SYNTAX DIAGRAM

32



<Routine declaration>s may include a variety of <directive>s, most of them relating to optimizations of different kinds. The inline, recursive, and reentrant <directive>s may be provided with the declaration of any routine; they are described in the next several sections. The safe and main <directive>s are specific to procedures. The commutative and associative <directive>s apply only to operators. Machine-dependent <directive>s applicable to routines are discussed in Chapter 13.

7.1.6 Inline Directive.



When a routine is invoked, the compiler will choose whether to expand the invocation inline (replacing the invocation with the routine body while preserving the semantics of parameter binding and the environment of the routine's declaration) or to generate a call of out-of-line code for the routine. This choice is governed by efficiency considerations. For example, the arithmetic operators such as + and - will likely be compiled inline, whereas a lengthy routine will likely be compiled out-of-line. Also, in general a recursive routine will be compiled out-of-line.

The <inline directive> may be supplied to guide the compiler in choosing between the two implementations. At run-time the effect of an invocation of a routine declared with the <inline directive> is the same as if this <directive> were absent, except that the execution times and/or code space requirements for the invocations may differ.

7.1.7 Reachability

To facilitate the presentation in subsequent sections, we here define the concepts of "directly reachable" routines and "reachable" routines.

(1) A routine R2 is said to be directly reachable from routine R1 provided that R1 contains an invocation of R2.

(2) A routine R2 is said to be reachable from routine R1 provided that

- (a) R2 is directly reachable from R1, or
- (b) R2 is reachable from some routine that is reachable from R1

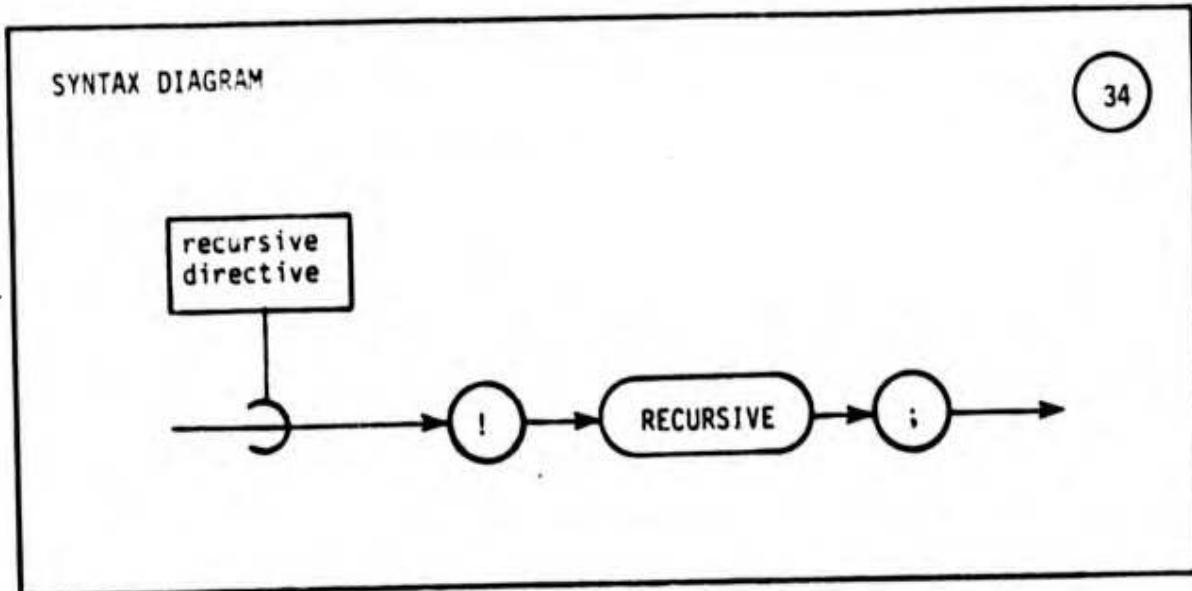
Reachability is compile-time (or link-time) detectable.

Example:

```
PROCEDURE P; ... END PROCEDURE P;  
PROCEDURE Q;  
...  
PROCEDURE R; ... CALL P; ... CALL Q; ... END PROCEDURE R;  
...  
CALL R;  
...  
END PROCEDURE Q;
```

R is directly reachable from Q. P and Q are directly reachable from R. R, P and Q are reachable from both Q and R.

7.1.8 Recursion



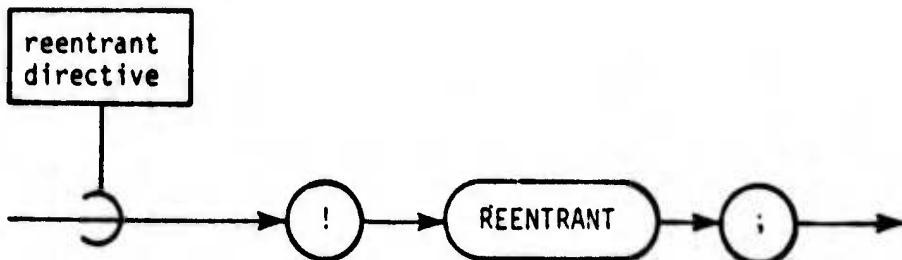
A routine is said to be recursive if it is reachable from itself. A recursive routine must be so marked by the presence of a <recursive directive>; i.e., the omission of this <directive> in the declaration of a recursive routine is a compile-time or link-time error.

REDL places the following weak restriction on routines so that recursion does not necessitate the use of a "display" (see Ironman 7B): a recursive routine may not contain the declaration of another recursive routine.

7.1.9 Reentrancy

SYNTAX DIAGRAM

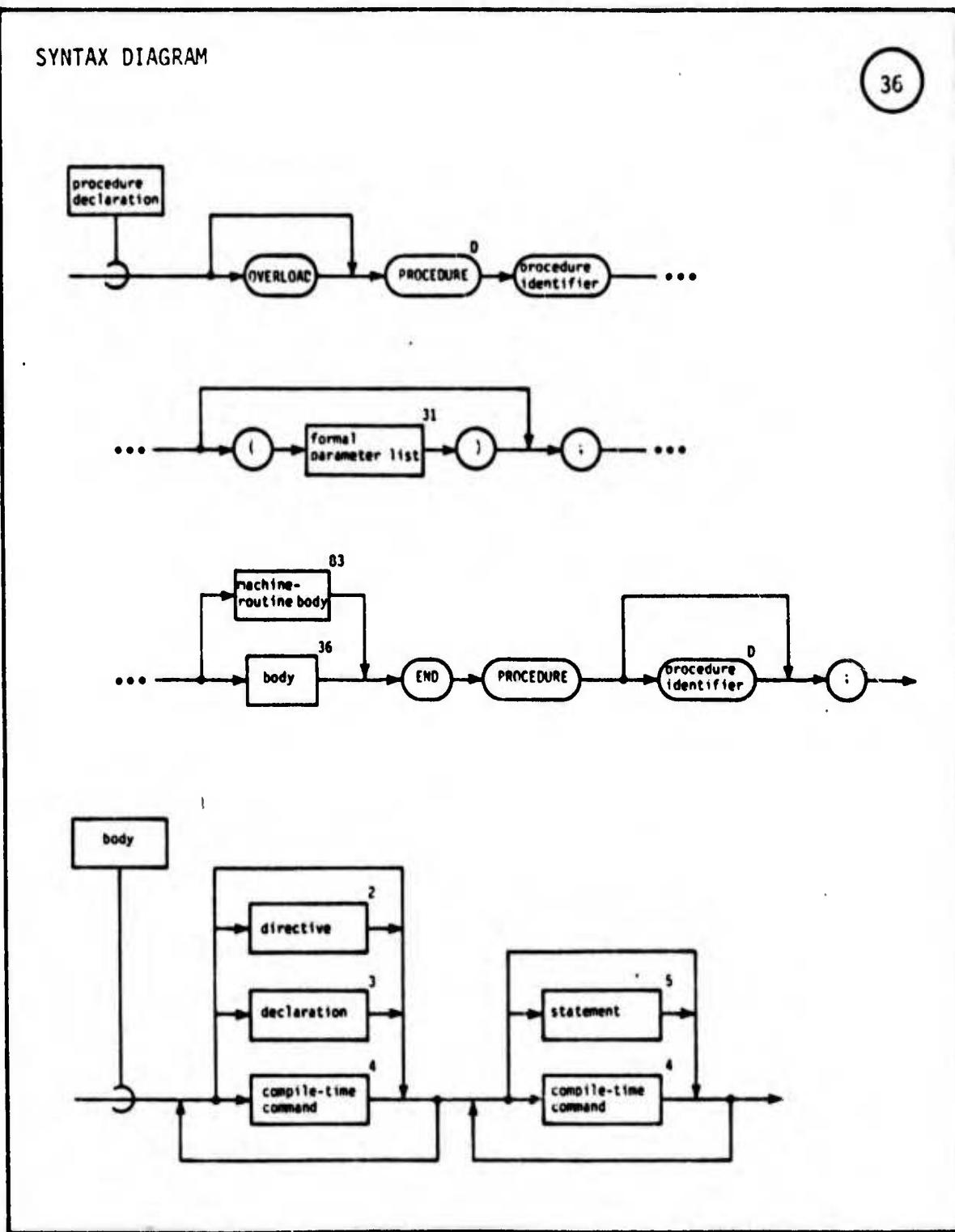
35



A routine is said to be reentrant if it is reachable from two different paths of some <fork statement>. Analogously to recursion, a reentrant routine must be so marked by the presence of a <reentrant directive>.

The recursive and reentrant <directive's are useful for program readability (they announce the programmer's intended use of the routine), reliability (they are checked by the compiler or linker), and efficiency (if a routine is nonrecursive and nonreentrant, the compiler may choose to allocate storage for the routine's local data statically --- i.e., at load time --- as a time optimization).

7.2 Procedures.



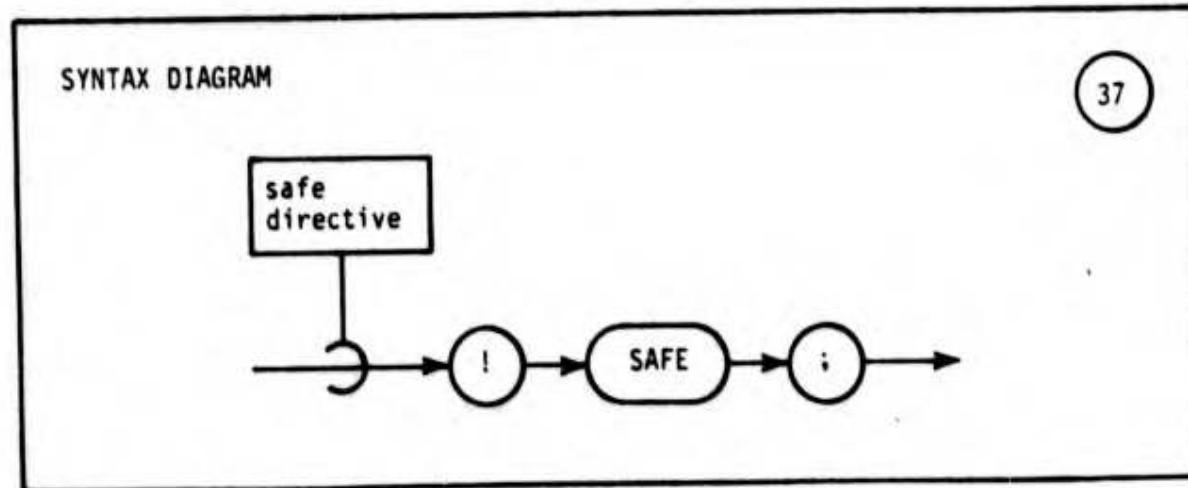
A procedure is a routine which is declared via a <procedure declaration> and invoked via a <call statement>. Procedures are executed for their effect on data which are either passed as actual parameters or are imported.

<Compile-time command>s in a procedure <body> may produce <directive>s, <declaration>s, and/or <statement>s, subject to the constraint that all <directive>s and <declaration>s must precede all <statement>s after expansion of the <compile-time command>.

The <machine-routine body> is used only for procedures written in assembly or machine language. This machine-dependent facility is described in Section 13.4.1.

In the interest of reliability, REDL places restrictions on the kinds of procedures which may be called from functions and operators, and restricts "aliasing" of data. These restrictions will be described in the next two sections.

7.2.1 Safe Directive.



It is desirable to allow both functions and operators to call procedures which are declared external to themselves. Since functions and operators are not permitted to cause side effects, the only side effects allowed in such procedures are changes to

the VAR and RESULT parameters (i.e., data declared external to the procedure but not passed as VAR or RESULT parameters will not be modified). The <safe directive> is used to specify that a procedure has this property. A procedure declared with the <safe directive> is said to be a safe procedure.

Any procedure, P, declared to be safe is checked at compile (or link) time to guarantee that

(1) All imported W-valued data elements are restricted to be readonly (by appearing in <readonly directive>s),

(2) Pointer dereferences contained in the procedure are used in R-valued contexts only, and

(3) All procedures external to P which are directly reachable from P or from routines internal to P must be safe procedures.

The external procedures reachable from functions, operators, or safe procedures must be safe procedures.

Example:

```
PROCEDURE P(VAR B: BOOLEAN); !SAFE; B := NOT B; END PROCEDURE P;
FUNCTION F RESULT R: BOOLEAN INIT TRUE;
  !IMPORT P;
  PROCEDURE Q; !IMPORT R; R := NOT R; END PROCEDURE Q;
  :
  CALL P(R);
  CALL Q;
  :
END FUNCTION F;
```

It may be noted that F can legitimately call P (an external procedure directly reachable from F and declared with the <safe directive>) and Q (an unsafe procedure defined inside F).

7.2.2 Aliasing.

Aliasing is the occurrence of different identifiers referring either to the same object or to an object and a component of the object. If aliasing occurs in a context where the object can be modified, it will be called dangerous aliasing. REDL prohibits all instances of dangerous aliasing. The following examples illustrate aliasing:

```
!PERVERSIVE T, REC_T, PTR_T;
TYPE T:...;
TYPE REC_T: RECORD...X:T;...END RECORD;
TYPE PTR_T: POINTER(VAR T);

VAR VT: T INIT...;
VAR VR: REC_T INIT...;
VAR VP1,VP2: PTR_T INIT...;

PROCEDURE P1(VAR t:T);
    !IMPORT VT;
    !READONLY VT;
    :
END PROCEDURE P1;

PROCEDURE P2(VAR r:REC_T, VAR t:T);
    :
END PROCEDURE P2;

PROCEDURE P3(VAR p1,p2: PTR_T, VAR t1,t2:T);
    :
END PROCEDURE P3;

CALL P1(VT); /*t and VT constitute an alias in P1*/
CALL P2(VR, VR.X); /*r and t constitute an alias in P2*/
CALL P2(VR, VT); /*no alias*/

VP1 := VP2;
CALL P3(VP1, VP2, VP1@, VP2@);
/*The only alias is that of t1 and t2*/
```

The only times during program execution when dangerous aliasing may arise are at procedure calls. (It is not possible for an alias to occur at a function or operator invocation, because these routines can only take parameters with CONST binding class and can only import data READONLY.) Formally, a dangerous alias of identifiers I1 and I2 is said to arise at a call of procedure P whenever:

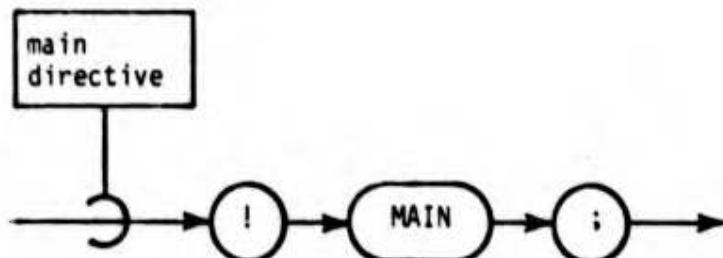
- (1) I1 and I2 refer either to the same object or to an object and a component of the object, and
- (2) I1 and I2 are:
 - (a) VAR formal parameters to P,
 - (b) RESULT formal parameters to P,
 - (c) a VAR and a RESULT formal parameter to P, or
 - (d) a VAR or RESULT formal parameter to P, and a datum imported by P.

Detection of such aliasing may involve run-time checking (e.g., when X(I) and X(J) are passed as VAR parameters). The occurrence of a dangerous alias raises the X_ALIAS exception (or causes a compile-time error if detected at compile-time).

7.2.3 Main Directive.

SYNTAX DIAGRAM

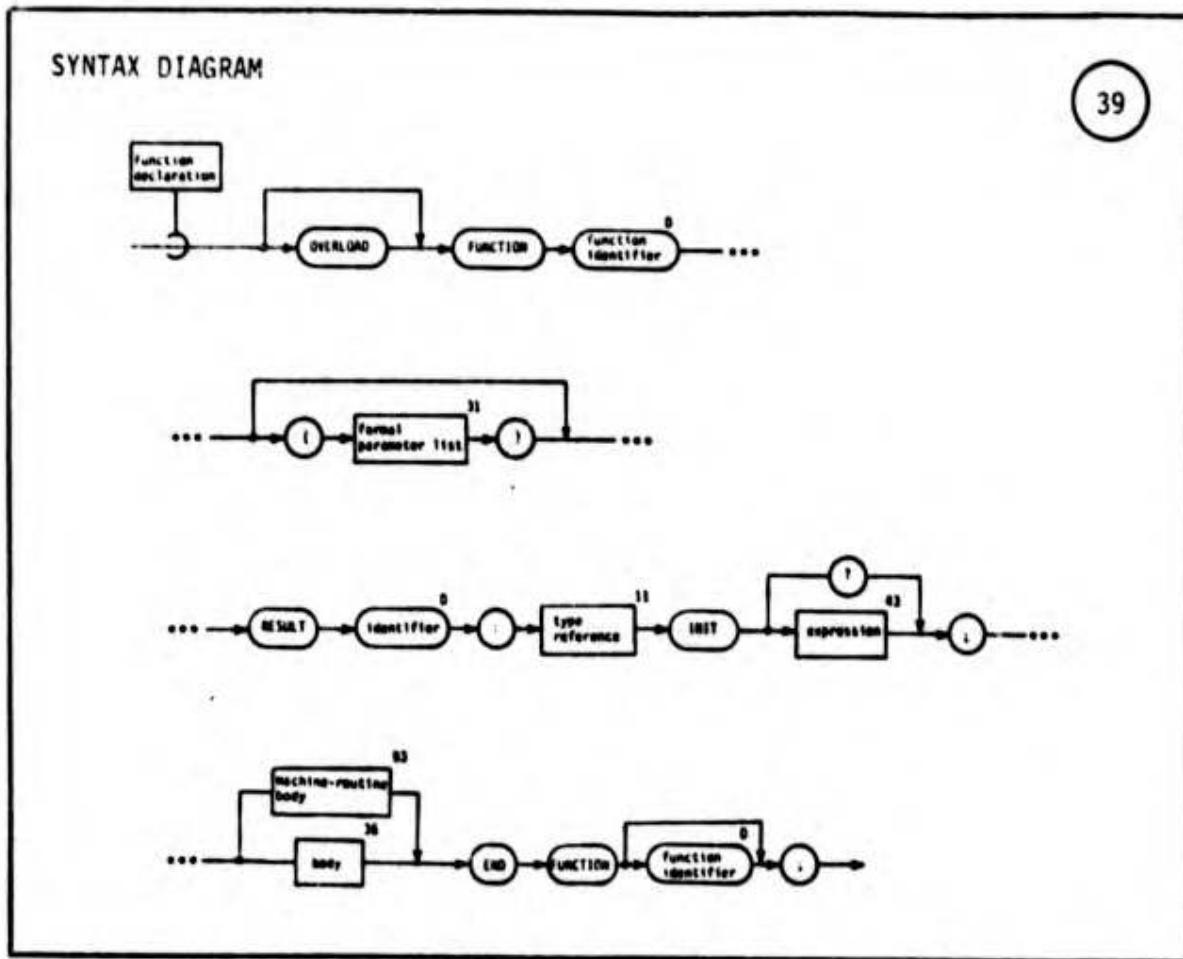
38



The user must supply the <main directive> for each top-level procedure which is capable of being the starting point for program execution. Each such procedure must take either no parameters or a single CONST parameter of type STRING. A load-time command will be used to select one procedure as the starting point, from the <program>(s) being loaded. The following example illustrates a possible application of the STRING parameter; when P is passed a STRING S, it will dispatch to either a debugging or production version, based on the value of S.

```
PROGRAM MAIN_EXAMPLE;
PROCEDURE P_DEBUGGING;...END PROCEDURE P_DEBUGGING;
PROCEDURE P_PRODUCTION;...END PROCEDURE P_PRODUCTION;
PROCEDURE P(CONST S:STRING(*));
  !MAIN;
  !IMPORT P_DEBUGGING, P_PRODUCTION;
  IF S = 'DEBUGGING' THEN CALL P_DEBUGGING;
  ELSEIF S = 'PRODUCTION' THEN CALL P_PRODUCTION;
  ELSE VAR STATUS: FIXED(-100..100);
    CALL WRITE('IMPROPER PARAMETER', STATUS);
    /*For simplicity, we ignore the status value
     *returned by WRITE*/
  END IF;
END PROCEDURE P;
END PROGRAM MAIN_EXAMPLE;
```

7.3 Functions.



There are two important differences between functions and procedures:

- (1) Functions return a result (an R-valued object) to be used in an <expression>, and
- (2) Functions have no side effects.

These points will be considered in more detail in the following two sections.

7.3.1 Function Result Variable.

The heading of each <function declaration> must include the declaration of a result variable. When the function is invoked, the initial value specified in the INIT phrase (if any) is assigned to the result variable. When a normal exit from the function occurs, the current value of the result variable is the value returned by the function to the point where it was invoked.

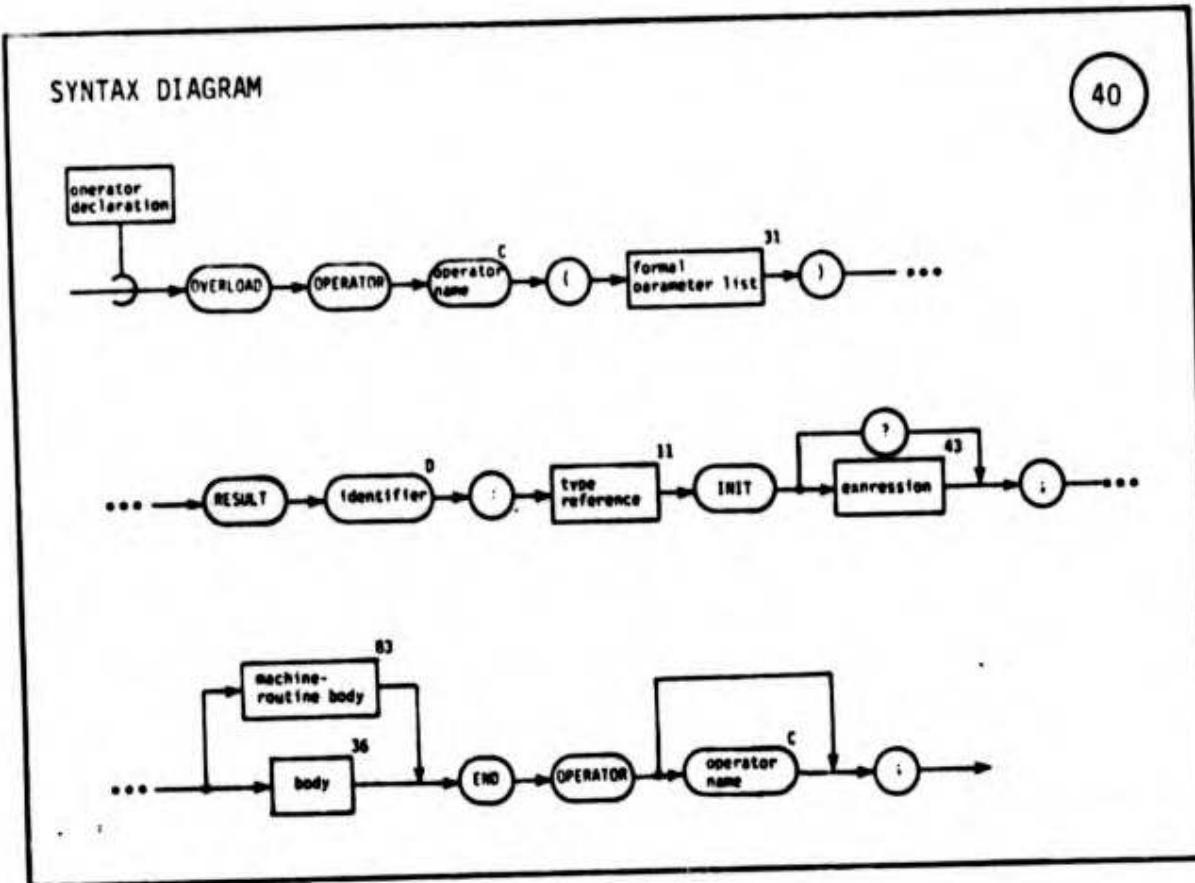
In compliance with Ironman 7D, the value for every low bound and high bound must be compile-time determinable for each array which is either a function result variable or a component of such a variable. Thus a result variable may be declared as a STRING(10) but not as a STRING(N) if N is a variable or run-time constant.

7.3.2 Prevention of Side Effects.

A routine is said to have a side effect if a normal execution of the routine (i.e., with a normal return) causes some data object declared outside of the routine to have a different value than it did before the routine was invoked. To prevent side effects from occurring in functions, REDL places the following restrictions:

- (1) The only binding class permitted for formal parameters to functions is CONST (i.e., VAR and RESULT binding classes are prohibited).
- (2) The function is "safe" in that the conditions of Section 7.2.1 are met (there is no need to supply a <safe directive> with the function, however).

7.4 Operators.

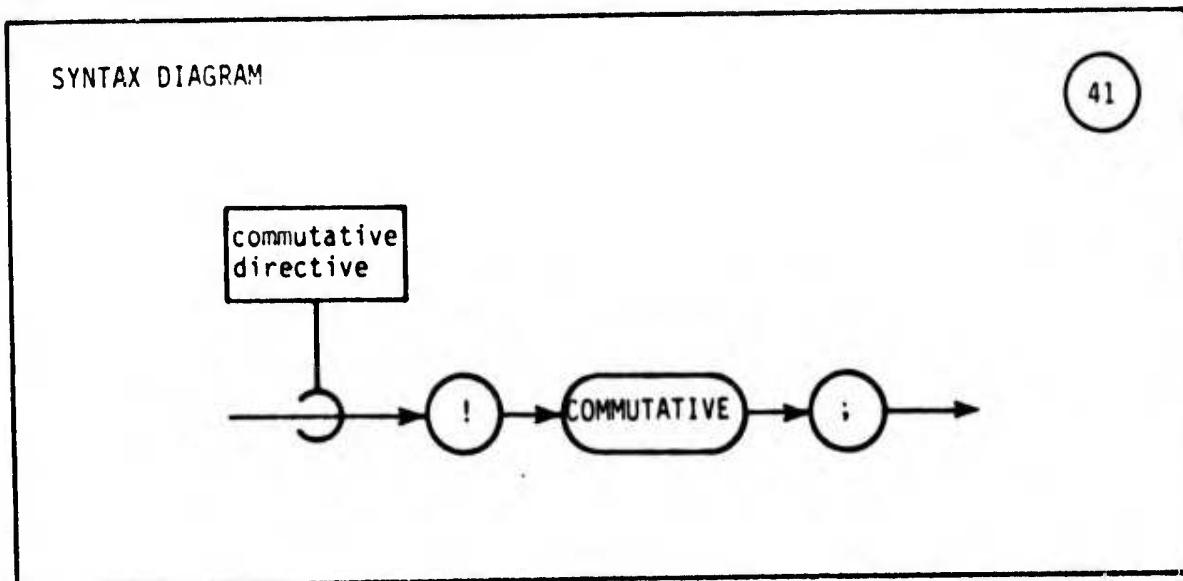


REDL's operators are all defined as overloaded, thus allowing the user to extend them to new data types. The syntactic properties of operator invocations (i.e., <expression>s) are described in Chapter 8. The semantic properties of the operators on the built-in types are described in Chapter 5. In this section we describe the differences between operators and functions, and discuss two <directive>s which are relevant to operators.

7.4.1 Contrast between Functions and Operators.

- (1) The syntax for <function invocation>s is distinct from the <expression> syntax used to invoke operators.
- (2) The order of evaluation of the actual parameters to a <function invocation> is always undefined; the built-in short-circuited conjunction and disjunction operators require the left operand to be evaluated before the right operand.
- (3) The names of functions are programmer-supplied identifiers, whereas operators have language-provided names.
- (4) Functions may or may not be overloadable; all operators are overloadable. However, each user-supplied overload line for an operator must have the same number of formal parameters (either one or two) as some system-provided line for the same operator. Thus, e.g., the user may not overload * to be a prefix operator.

7.4.2 Commutative Directive.

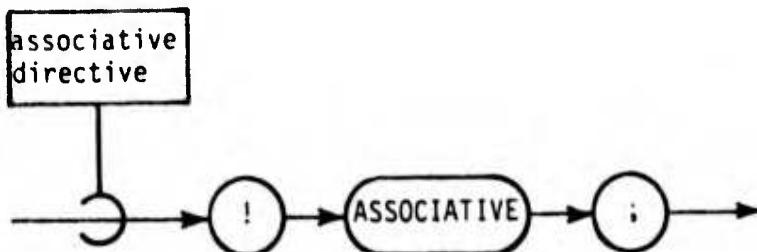


By supplying the <commutative directive> with an overload line for one of the infix operators listed below, the user is asserting that the compiler may interchange the two operands (i.e., treat the right operand as left and vice versa). The only operators for which this is permissible are *, +, =, #, AND, OR, and XOR. Both formal parameters of a line declared with the <commutative directive> must have the same type.

7.4.3 Associative Directive.

SYNTAX DIAGRAM

42



By supplying the <associative directive> with an overload line for one of the infix operators listed below, the user is asserting that the compiler may group either left-to-right or right-to-left those operands separated by consecutive occurrences of the operator. That is, if op is declared with the <associative directive>, then A op B op C may be grouped as (A op B) op C or as A op (B op C). The only operators for which this is permissible are *, CAT, +, AND, OR, &, !, and XOR. Moreover, for each line declared with the <associative directive>, the result variable and both formal parameters must have the same type. (Note that & and ! lose their short-circuit semantics when overloaded by the user.)

8.0 EXPRESSIONS

8.1 Introduction.

An *<expression>* is a notation derived from usage in mathematics which allows an operator invocation to be written as a combination of operands and operator symbols instead of as a *<function invocation>*. An operand is either an atomic constituent known as a *<primary>*, or a subexpression which is in turn composed of further operators and operands. An operator is either an infix operator, positioned between its left and right operand; or a prefix operator, positioned to the left of its operand. The invocation of an operator produces an R-valued object, never a W-valued one.

This chapter describes the syntactic, but not the semantic properties of operators. Semantic properties are discussed in Chapter 5.

8.2 Comparison with Pascal.

The major differences in expression syntax between Pascal and REDL are the following:

(1) REDL contains several operators absent from Pascal: exponentiation, catenation, exclusive disjunction, and short-circuited conjunction and disjunction. The Pascal operator for set membership is absent from REDL.

(2) In REDL the boolean operators are at lower precedence levels than the relational operators, whereas in Pascal the reverse is true. In this respect REDL adopts the conventions of ALGOL 60. Thus the *<expression>* $X < Y \text{ OR } R = S$ is parsed as $((X < Y) \text{ OR } (R = S))$ in REDL, whereas it would be illegal in Pascal.

8.3 The Parse of an Expression.

The parse of an $\langle \text{expression} \rangle$ is an explicit decomposition of the $\langle \text{expression} \rangle$ so that the operand(s) for each operator are apparent. (As will be seen below in connection with commutative and associative operators, the parse is not necessarily the decomposition which will be used in the evaluation of the $\langle \text{expression} \rangle$.) It is convenient to represent a parse as a fully parenthesized $\langle \text{expression} \rangle$, so that between each paired left and right parenthesis there is exactly one operator which does not appear within any inner pair of parentheses. Thus we can represent the parse of $A+B*C$ as $(A+(B*C))$, indicating that the product of B and C is to be added to A .

There are three factors which determine the parse for any $\langle \text{expression} \rangle$: the precedence (or "binding strength") of the operators, the presence of explicit parentheses, and the associativity of the operators. For example, the fact that * (multiplication) has higher precedence than + (addition) implies that $A+B*C$ is parsed as $(A+(B*C))$ instead of as $((A+B)*C)$. In order to produce the latter, the user would have to supply explicit parentheses surrounding $A+B$.

REDL has 7 precedence levels, reflecting the operators' common usage in mathematics. These levels are displayed in Table 8-1 and are also implied by the syntax diagrams for $\langle \text{expression} \rangle$ (these appear in Section 8.5). As an example, $A+B=C$ AND $D < E$ has the parse $((A+B)=C)$ AND $(D < E))$.

The precedence levels by themselves are not sufficient to determine the parse of an $\langle \text{expression} \rangle$; in particular, when two consecutive operators in an $\langle \text{expression} \rangle$ have the same precedence, "associativity" information is required. Consider $A-B+C$. The "-" and "+" operators have the same precedence, and the above

LEVEL	OPERATOR	PURPOSE
HIGHEST ↓ LOWEST	**	Exponentiation
	*	Multiplication
	/	Division with FLOAT result
	DIV	Division with FIXED result
	MOD	FIXED remainder
	CAT	Catenation
	+	Addition (infix) or Identity (prefix)
	-	Subtraction (infix) or Negation (prefix)
	=	Equality
	#	Inequality
	<	Less Than
	<=	Less Than or Equal
	>	Greater Than
	>=	Greater Than or Equal
	NOT	Complement (prefix)
	AND	Conjunction
	&	Short-Circuited Conjunction
	OR	Disjunction
	!	Short-Circuited Disjunction
	XOR	Exclusive Disjunction

TABLE 8-1: Precedence Levels

<expression> appears as though it might be parsed either as $((A-B)+C)$, or as $(A-(B+C))$. REDL adopts the following simple convention: whenever two consecutive operators in an *<expression>* have the same precedence, associativity is defined to be left-to-right. Thus $A-B+C$ is parsed as $((A-B)+C)$; $-A-B*C-(D+E)$ is parsed as $(((-A)-(B*C))-(D+E))$. (We point out that "consecutive" in the above rule must be interpreted in the proper sense: in $A-B*C-D$, the two minuses are considered to be consecutive--in effect, $B*C$ is regarded as a single operand--whereas in $A-(B-C)$ the minuses are not consecutive, because of the intervening parenthesis. A more precise statement of the convention is: whenever two operators of the same precedence appear in an *<expression>* where the only non-parenthesized operators appearing between them have higher precedence, then associativity is left-to-right.)

Some of the infix operators model algebraic operations which are commutative or associative in the mathematical sense: *, +, =, #, AND, OR, and XOR model commutative operations; *, CAT, +, AND, OR, &, !, and XOR model associative operations. (An operator op on a set S is said to be commutative if $x \text{ op } y = y \text{ op } x$ for all x, y in S. An operator op on a set S is said to be associative if $(x \text{ op } y) \text{ op } z = x \text{ op } (y \text{ op } z)$ for all x, y, z in S.) By taking advantage of such properties a compiler may be able to realize significant run-time efficiencies. In order to allow this, the language permits the rearrangement of operands within an *<expression>* provided that the associativity caused by explicit programmer-supplied parentheses is used, and that the mathematical properties of the modeled operations are preserved. For example, the compiler may choose to evaluate an arithmetic *<expression>* $A+B+C$ in any of twelve different ways, taking advantage of the commutativity and associativity of addition: $((A+B)+C)$, $(A+(B+C))$, ... $((C+B)+A)$, $(C+(B+A))$. If the user overloads any of the operators named above, then the commutative or associative *<directive>* (see Section 7.4) can be supplied to allow the compiler to perform rearrangement of operands.

8.4 Order of Evaluation

For the short-circuited operators, "&" and "!", the left operand will be evaluated first and, if this is sufficient to determine the value of the result, the right operand will not be evaluated.

For each operator except the short-circuited ones, the order of evaluation of operands is undefined. Implementations are free to evaluate operands in either order and also to leave an operand unevaluated if the result can be determined without evaluating the operand.

Examples:

(1) In the expression $(A-B)/2^{**}K$, the denominator may be evaluated first (some languages, requiring parenthesized operands to be evaluated before non-parenthesized ones, insist that $(A-B)$ be evaluated before $2^{**}K$).

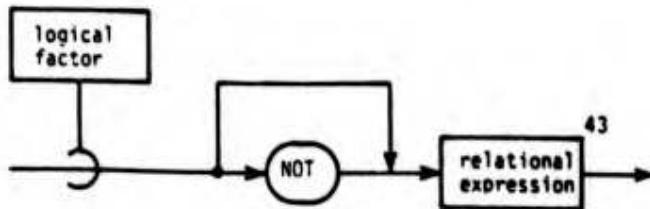
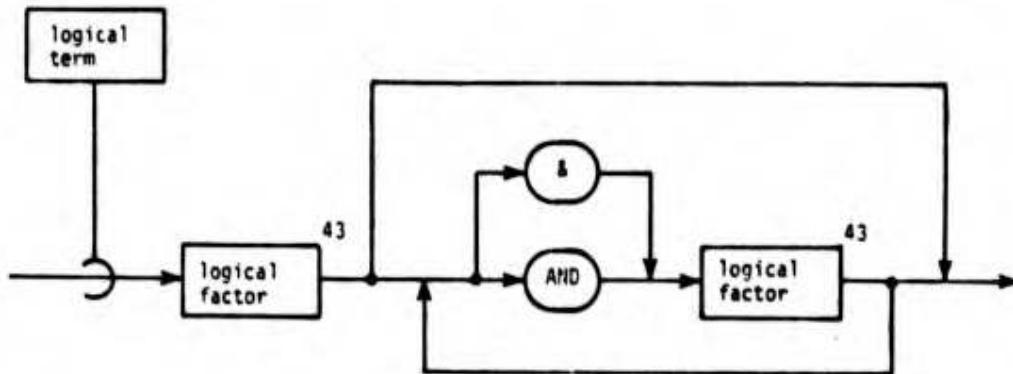
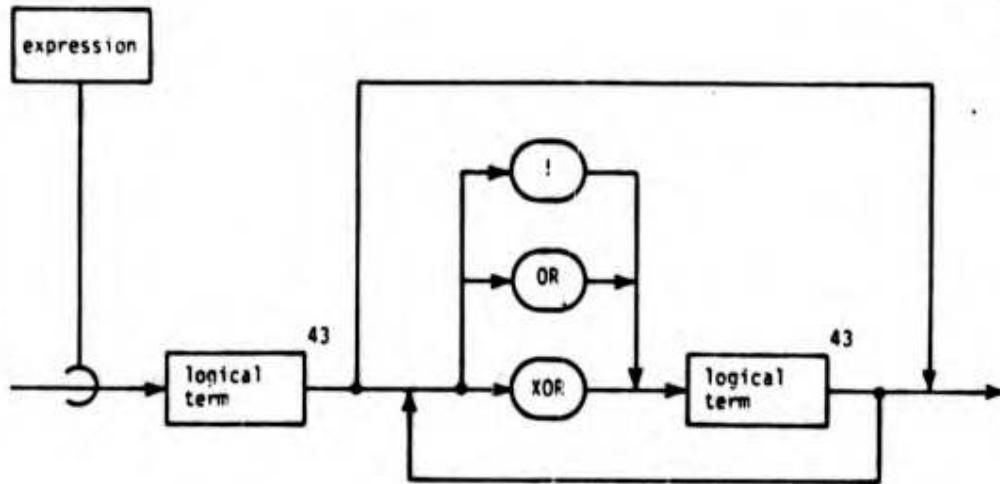
(2) In the expression $Z*(A+B)$, if the translator can guarantee that Z is 0 (Z may be a compile-time constant), then $(A+B)$ need not be evaluated.

(3) Implementations may choose to evaluate AND and OR in short-circuit form.

8.5 Diagrams for Expression Syntax.

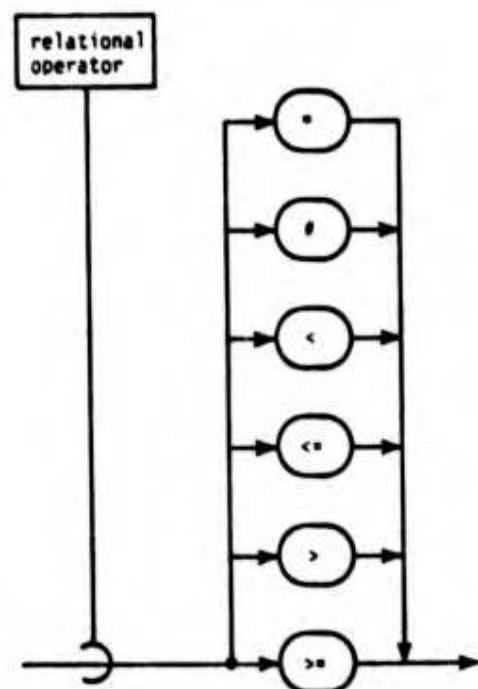
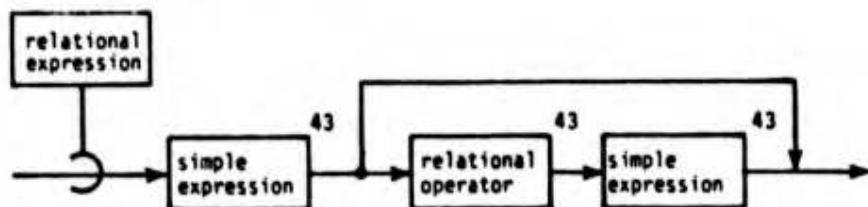
SYNTAX DIAGRAM

43



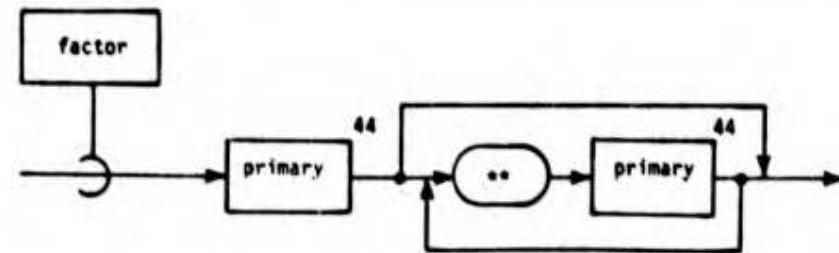
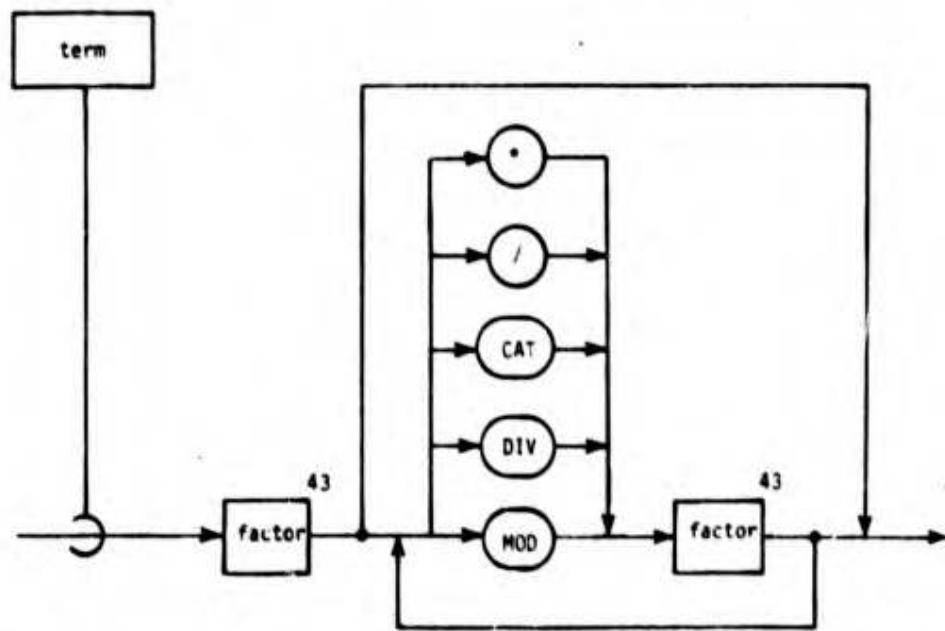
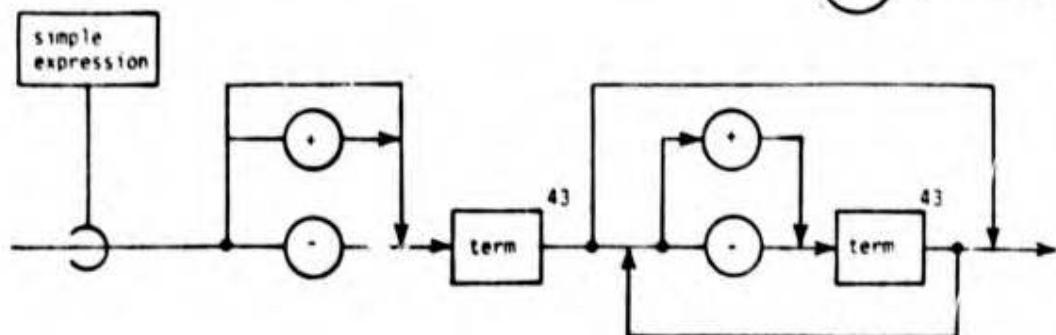
SYNTAX DIAGRAM

43 (CONTINUED)



SYNTAX DIAGRAM

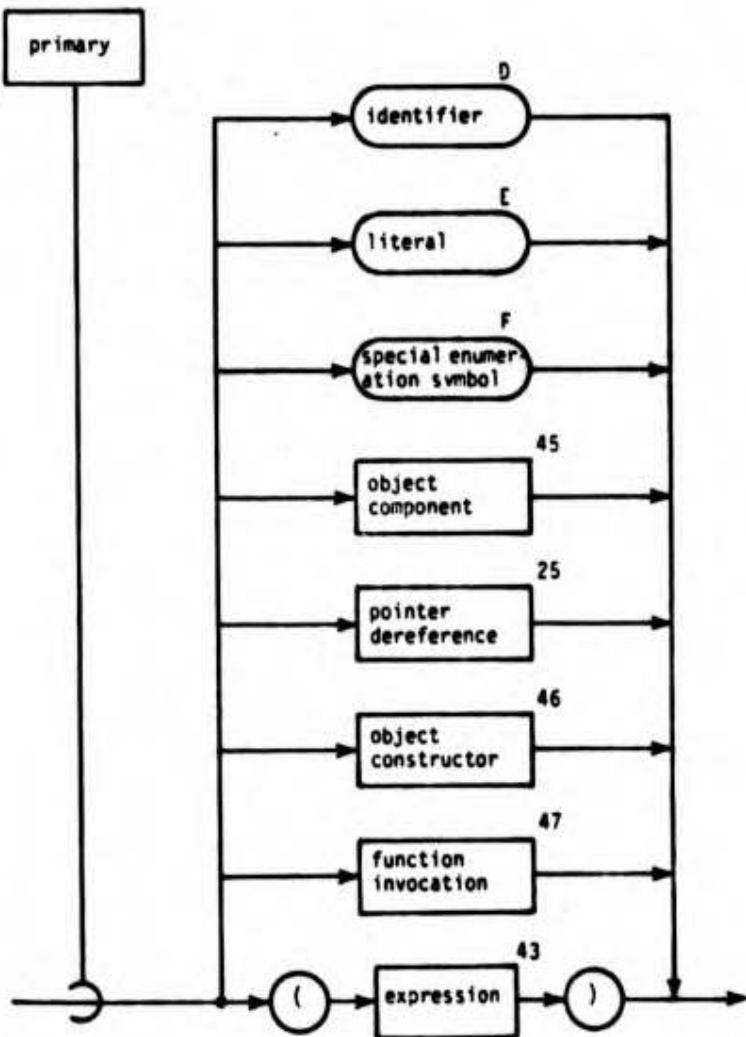
(43) (CONTINUED)



8.6 Primary.

SYNTAX DIAGRAM

44



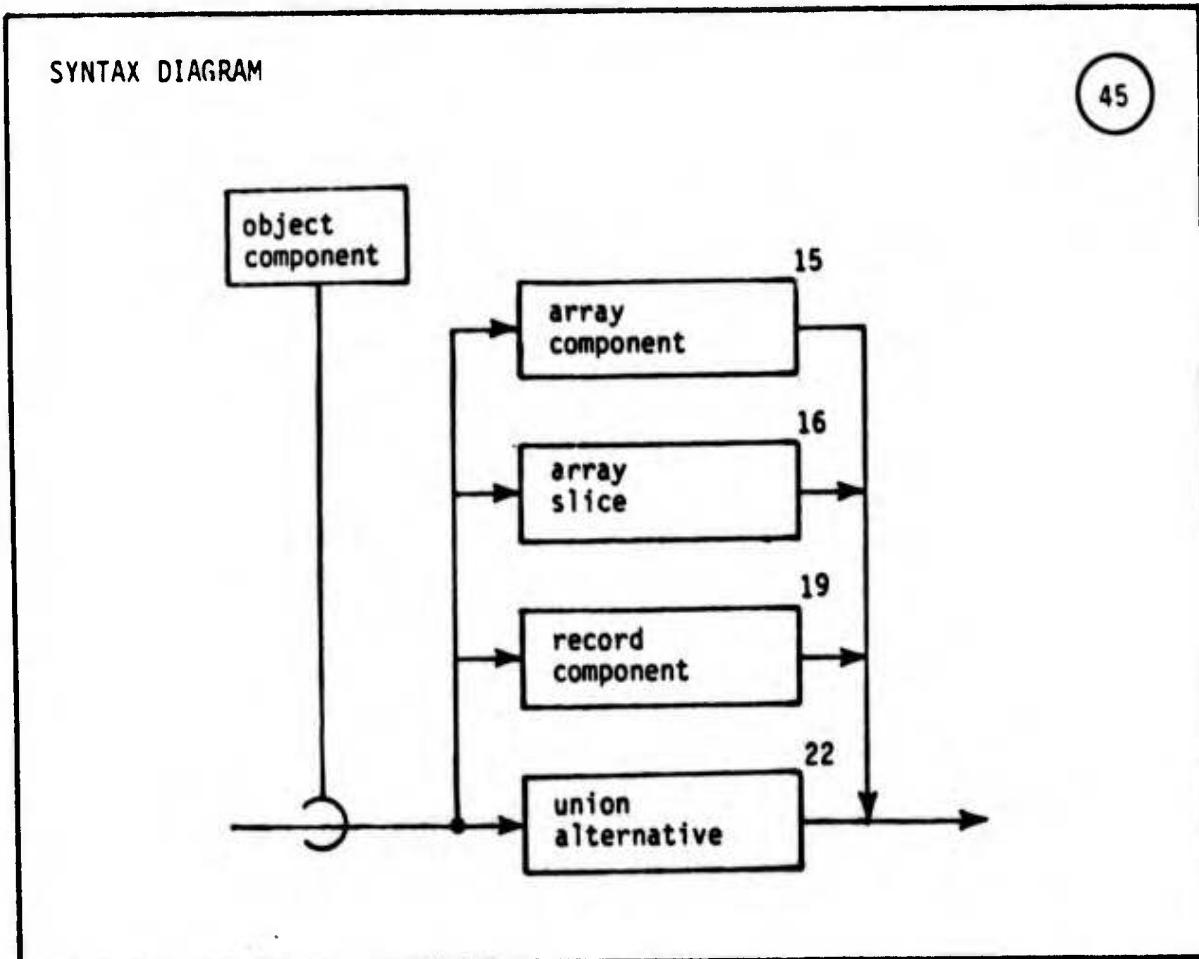
<Primary>s serve two purposes in the syntactic scheme of REDL: they are the atomic constituents of <expression>s, and they provide a means for generating W-valued objects.

A <primary> is evaluated to produce an object. The resulting object is W-valued in the following cases:

- (1) The <primary> is an identifier which denotes a W-valued object such as a variable or a VAR or RESULT formal parameter.
- (2) The <primary> is an <object component> for an object which is itself W-valued.
- (3) The <primary> is a <pointer dereference>, and the pointed-to type is specified as VAR.

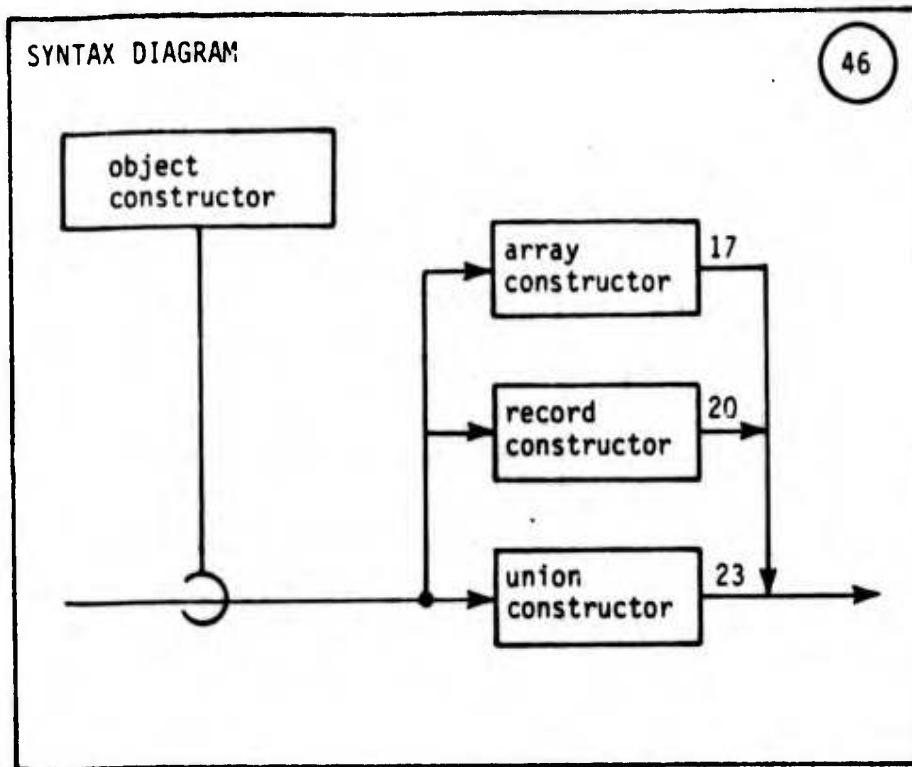
In all other situations the object is R-valued and not W-valued. In particular, compile-time identifiers, literals, and parenthesized <expression>s always produce R-values.

8.6.1 Object Component.



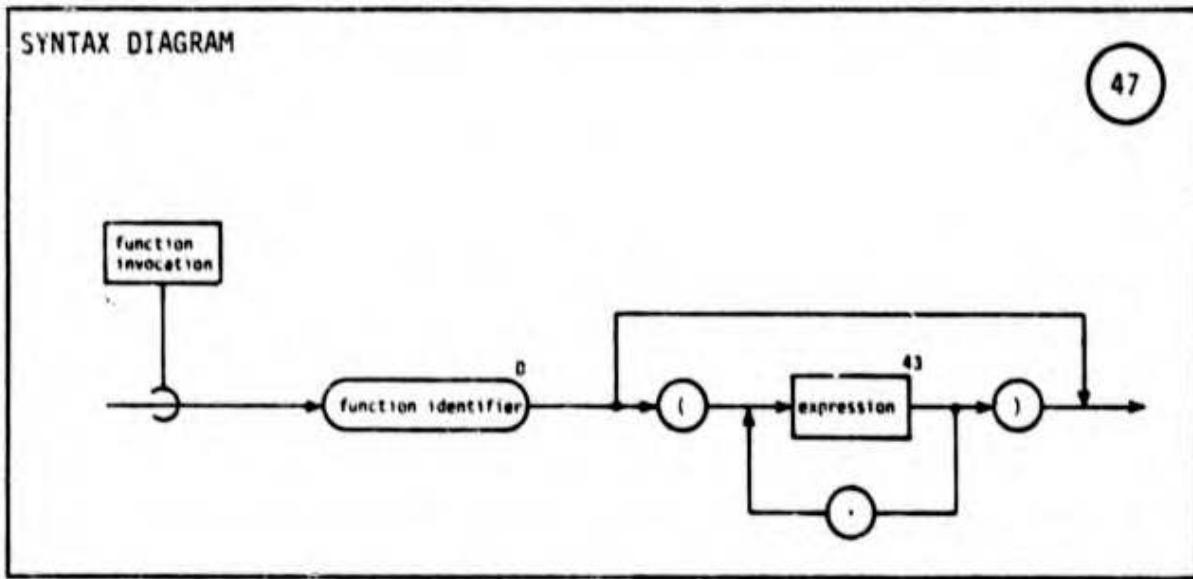
A component of an object of aggregate type may be selected by array subscripting or slicing, record qualifying, or union qualifying. The component is W-valued if and only if the aggregate object is.

8.6.2 Object Constructor.



An R-valued object of an aggregate type may be explicitly constructed via the three forms mentioned.

8.6.3 Function Invocation.

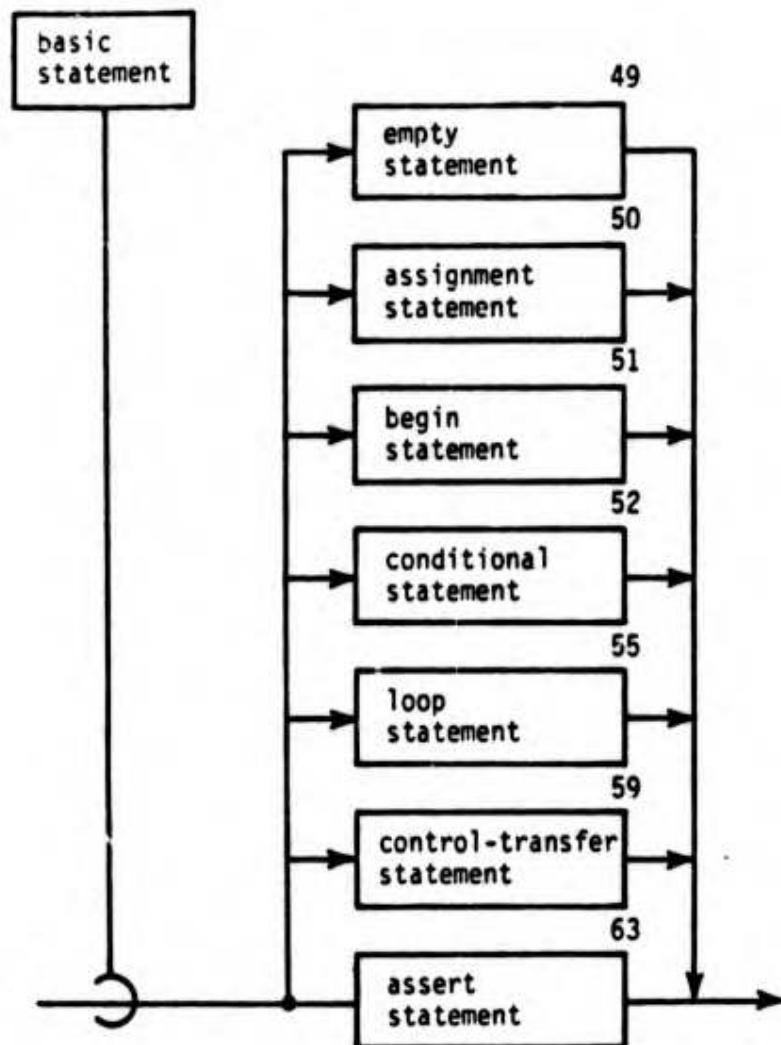


For the semantics of invocation, parameter passing, and return, see Sections 7.1.3 and 7.1.4. The object produced by a <function invocation> is always R-valued.

9.0 BASIC STATEMENTS

SYNTAX DIAGRAM

48



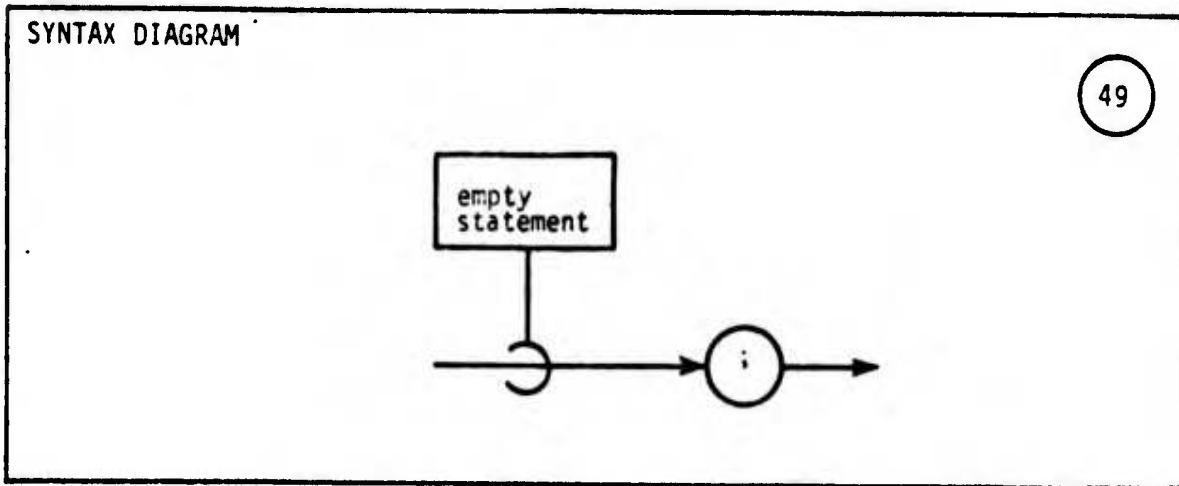
<Statement>s are executable; that is, <statement>s denote algorithmic actions to be performed at run-time. Any <statement> may optionally be prefixed by a label (see Section 4.1.4).

9.1 Comparison with Pascal.

The design of the <statement> in REDL is based heavily on Pascal. The main differences between the two are as follows:

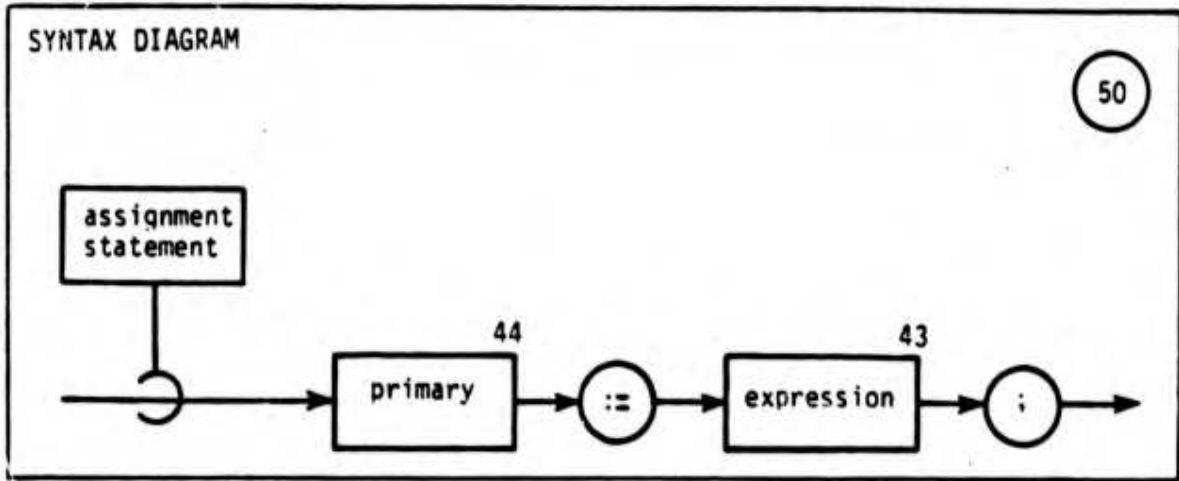
- (1) <Begin statement>s in REDL may contain declarations. Compound statements in Pascal cannot.
- (2) The END bracketing on conditional and loop <statement>s in REDL allows these <statement>s to contain <body>s and not simply single <statement>s. In Pascal, each of the if, case, for, and while statements has a single statement as its constituent.
- (3) REDL's <if statement> permits ELSEIF clauses, facilitating the nesting of <if statement>s in a readable fashion.
- (4) REDL's <select statement> is more general than Pascal's case statement (label ranges are allowed) and is more secure in its treatment of tag fields.
- (5) Unlike Pascal, REDL prohibits a <goto statement> from transferring control out of a routine.
- (6) The with statement of Pascal is absent from REDL.
- (7) REDL includes exit and assert <statement>s, absent from Pascal. For reliability and readability, the exit <statement> in REDL specifies a label of the construct to be exited.

9.2 Empty Statement.



The execution of an <empty statement> results in no action.

9.3 Assignment Statement.

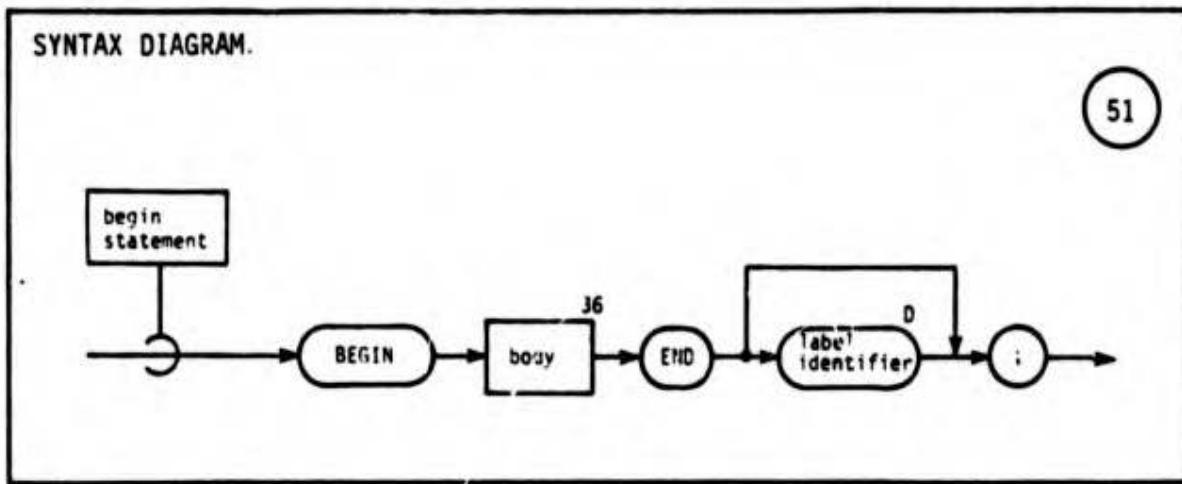


The execution of an <assignment statement> replaces the current value of a W-valued object by a new value produced by evaluating an <expression>. Evaluation of the <primary> must yield a

W-valued object, and the types of the <primary> and <expression> must be same. The rules for permissibility of the assignment are in Sections 5.1.2 and 5.6.4.

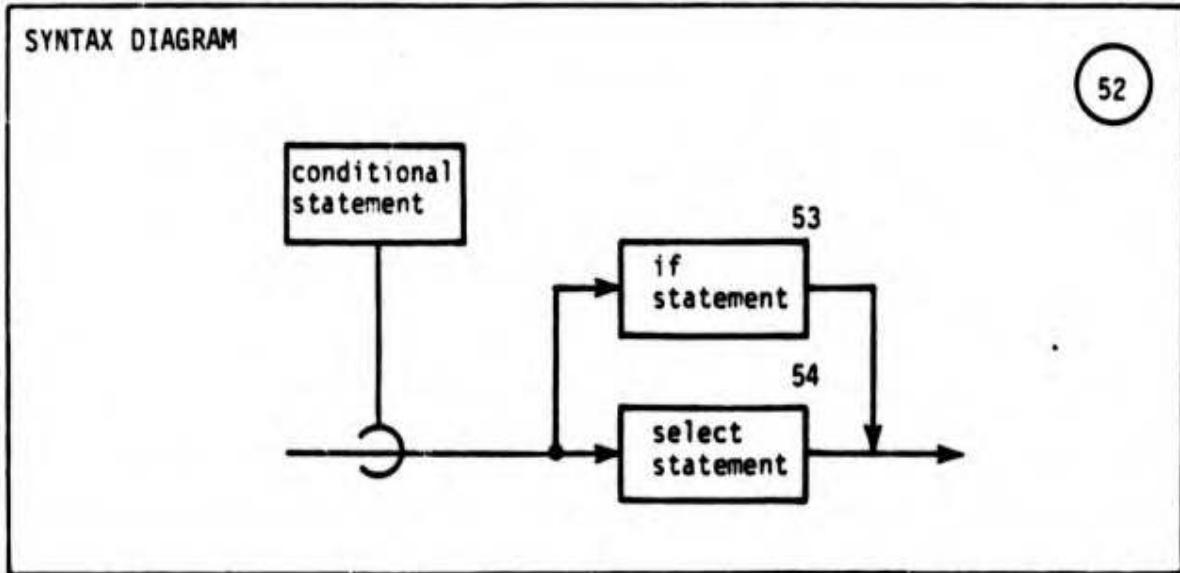
The relative order of evaluation of the <primary> and the <expression> is not specified.

9.4 Begin Statement.

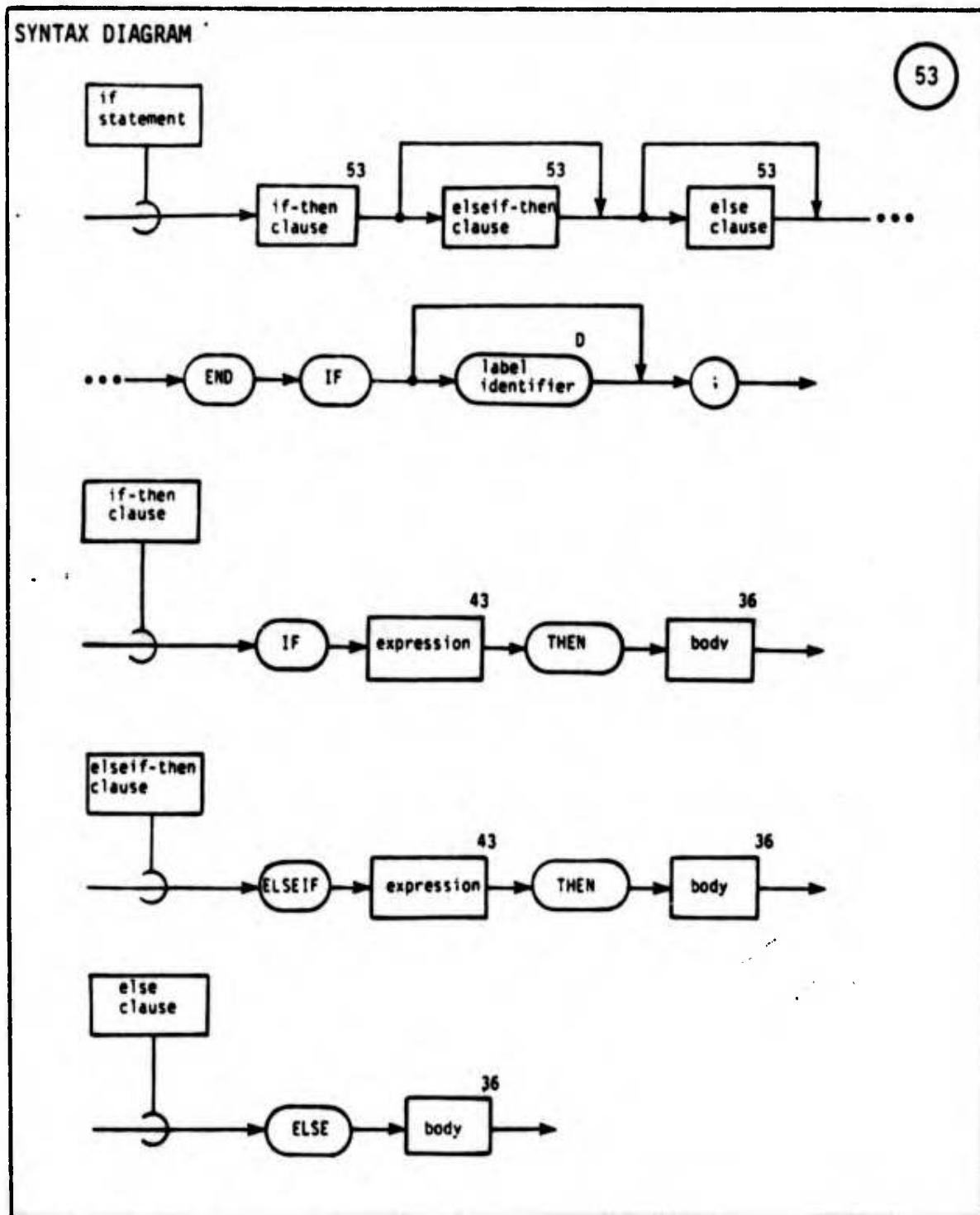


A <begin statement> is an open scope; it is a construct which contributes toward readability (since declarations can be localized at the place where they are needed) and efficiency (since data declared in a <begin statement> need only be allocated for the duration of the <statement>). Control leaves a <begin statement> either explicitly (via a <goto statement> or <exit statement>) or implicitly (via a raised exception or "dropping out the bottom").

9.5 Conditional Statement.



9.5.1 If Statement.



Each <expression> must be of BOOLEAN type. Execution of an <if statement> consists of the following steps:

(1) Evaluate the <expression> in the <if-then clause> and in each <elseif-then clause>, in order, until the first occurrence of a TRUE result; then execute the <body> in this clause.

(2) If the <expression> in the <if-then clause> and in each <elseif-then clause> is FALSE, then execute the <body> in the <else clause> (if this clause is present).

(3) If the expression in the <if-then clause> and in each <elseif-then clause> is FALSE, and there is no <else clause>, then no further action occurs.

Each <body> in the <if statement> is an open scope. It is thus not possible to transfer control into an <if statement> from outside, or to transfer control between the clauses.

The utility of the <elseif-then clause>s is to allow the nesting of if-then constructs without necessitating an accumulation of END IF delimiters.

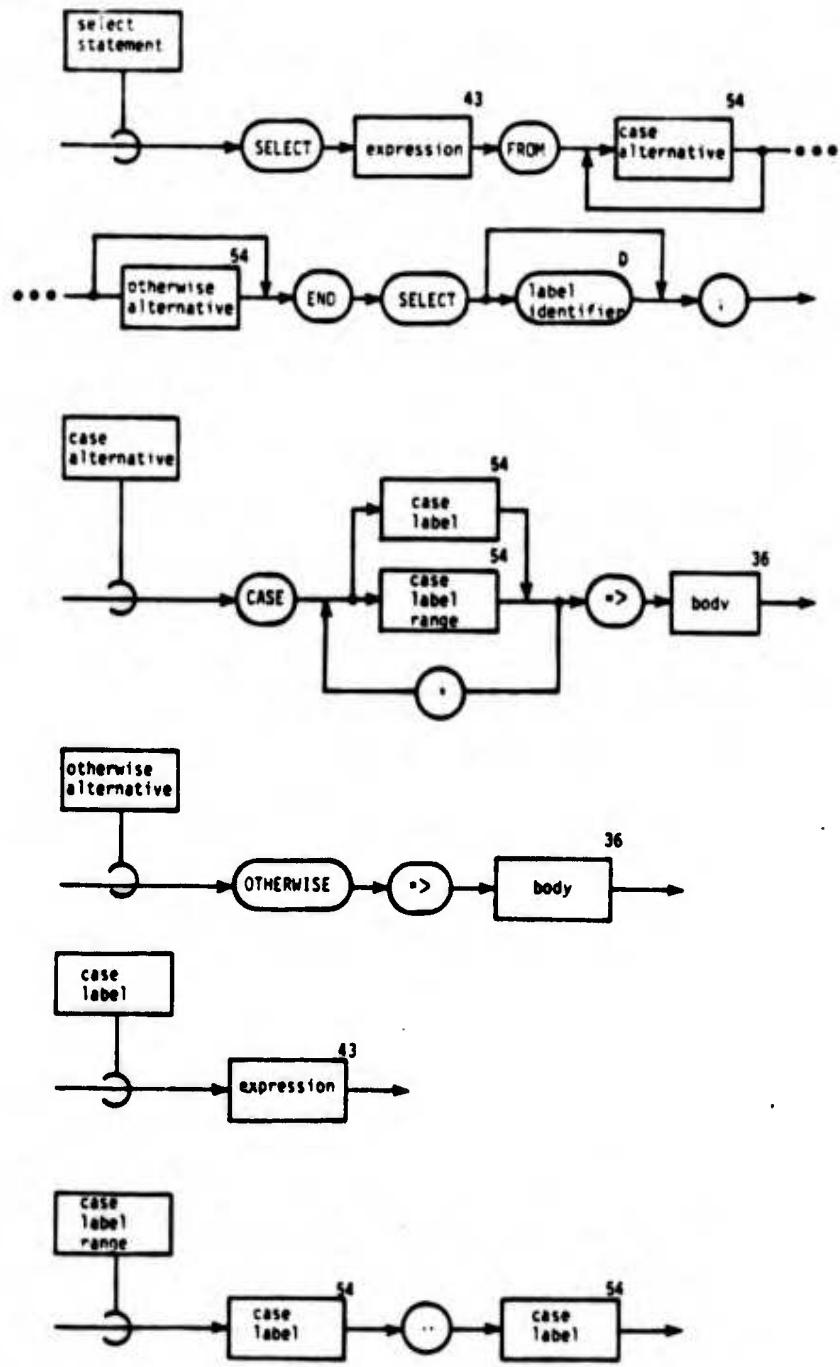
Example:

```
IF      X<0          THEN CALL P1;
ELSEIF (X=0) & (Y>0) THEN CALL P2; Z:=0;
ELSEIF  X>0          THEN CALL P3;
ELSE                      W:=Z+Y;
END IF;
```

9.5.2 Select Statement.

SYNTAX DIAGRAM

54



The `<select statement>` permits one of a set of alternative actions to be performed, based on a selection criterion embodied in the `<expression>` following `SELECT`. The possible criteria are:

- (1) the actual value of an object, if the `<expression>` is integer- or enumeration-type-valued (see Section 9.5.2.1),
- (2) the exception which caused a given handler to be invoked, if the `<expression>` is the special identifier `LAST_EXCEPTION` (see Section 11.5),
- (3) the value of the "tag" field of a union, if the `<expression>` is of some union type (see Section 9.5.2.2).

A feature common to all three is that the `<body>` of each alternative is a separate open scope. It is thus not possible to transfer control into or between alternatives.

9.5.2.1 Value Select. This is the case in which the `<expression>` following `SELECT` has integer or enumeration type. Whatever type this is, it must be same as the type of each `<case label>`. Moreover, each `<case label>` must be a compile-time-evaluable `<expression>`. No two labels may have the same value, no label may be included in any `<case label range>`, and no two `<case label range>`s may overlap.

Execution of the `<select statement>` consists of the following steps:

- (1) The `<expression>` following `SELECT` is evaluated.
- (2) The resulting value is compared with the `<case label>`s and `<case label range>`s appearing on the `<case alternative>`s. If a match is found, or if the value is within one of the `<case label range>`s, the `<body>` of the `<case alternative>` is executed (the constraints in the preceding paragraph ensure that there will be at most one such match).
- (3) If no match is found and the `<otherwise alternative>` is present, the `<body>` of this alternative is executed.

- (4) If no match is found and the <otherwise alternative> is absent, the X_SELECT exception is raised.

Example:

```
VAR CH: CHAR INIT ?;
VAR X: FIXED(0..15) INIT ?;
!
SELECT CH FROM
CASE ^0^.^9^ => X := FIXED(CH)-FIXED(^0^);
CASE ^A^.^F^ => X := 10+FIXED(CH)-FIXED(^A^);
OTHERWISE      => RAISE X_RANGE;
END SELECT;
```

9.5.2.2 Union Select. This is the case in which the <expression> following SELECT has a union type. The purpose of this <statement> is to allow a safe and efficient discrimination on the value of the tag field of a union data object. <Case label range>s are not permitted, and each <case label> must be a field name from the union type. No two <case label>s may be the same.

Execution of the <select statement> consists of the following steps:

- (1) The <expression> following SELECT is evaluated, and its tag field is retrieved.
- (2) The resulting field name is compared with the <case label>s appearing on the <case alternative>s. If a match is found (there can be at most one), the <body> of that <case alternative> is executed. If the <expression> following SELECT is a W-valued object, then whenever control resides in a <statement> contained within this <body>, the tag field of the object must be the same as when execution of the <body> began. The X_TAG exception is raised if this condition is violated.
- (3) If no match is found and the <otherwise alternative> is present, the <body> of this alternative is executed.
- (4) If no match is found and the <otherwise alternative> is absent, the X_SELECT exception is raised.

Example:

```
TYPE U: UNION
    A: CHAR;
    B: BOOLEAN;
END UNION;

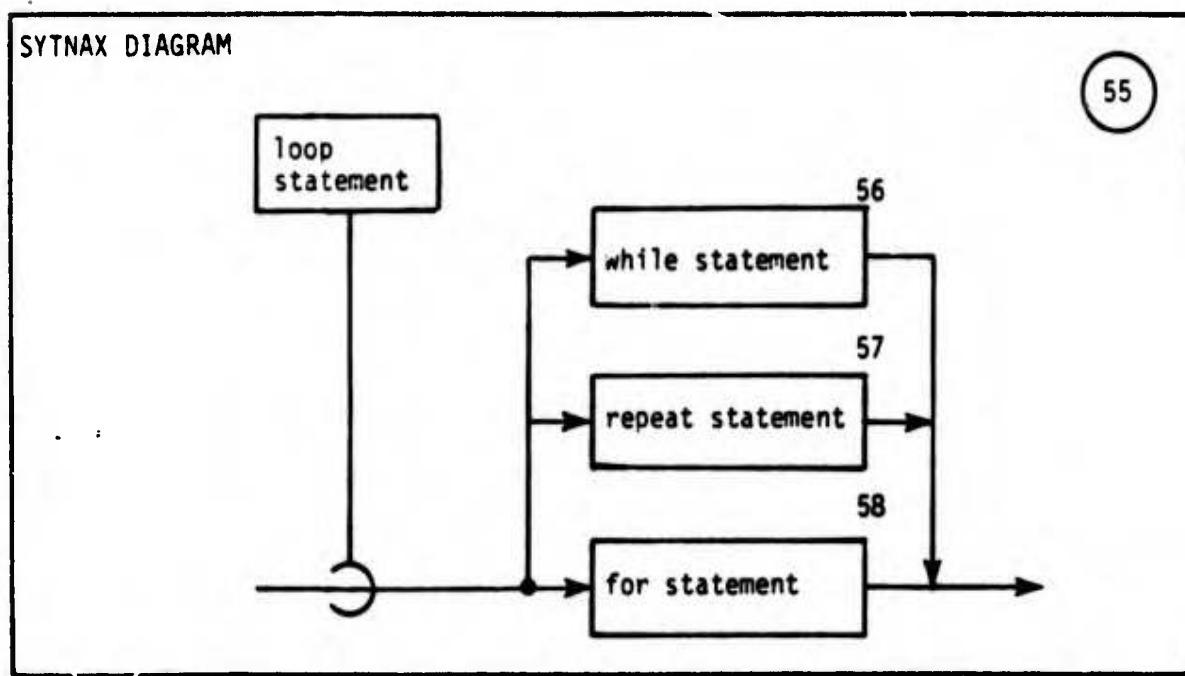
VAR V: U INIT U(A: ^Z^);

PROCEDURE TAG_CHANGE;
    !IMPORT V;
    V:= U(B:TRUE);
END PROCEDURE TAG_CHANGE;

SELECT V FROM
    CASE A => CALL TAG_CHANGE; <*Illegal*>
        V.A := PRED(V.A);
    CASE B => V.B := NOT V.B;
END SELECT;
```

A reasonable implementation is to perform tag checking inside <case alternative> <body>s only on return from procedures which could possibly change the tag. Thus the change of V's tag from A to B would be detected in the <body> of CASE A immediately after the return from TAG_CHANGE. Moreover, field selections from the union object which occur in the <body> of a <case alternative> with a single <case label> do not require run-time tag checking; V.A is either a correct reference (when it occurs in a <body> with A as the <case label>) or a compile-time error (when it occurs in a <body> which does not have A as a <case label>).

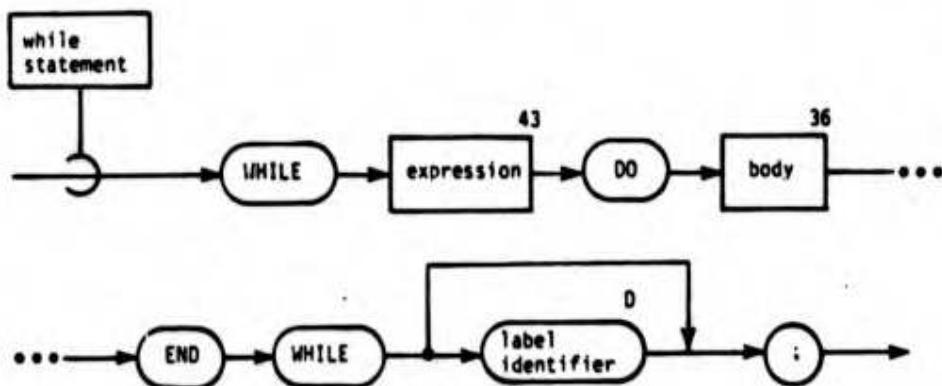
9.6 Loop Statements.



9.6.1 While Statement.

SYNTAX DIAGRAM

56



The <while statement> provides a means for executing a <body> zero or more times, as long as some condition (tested before each iteration) is true.

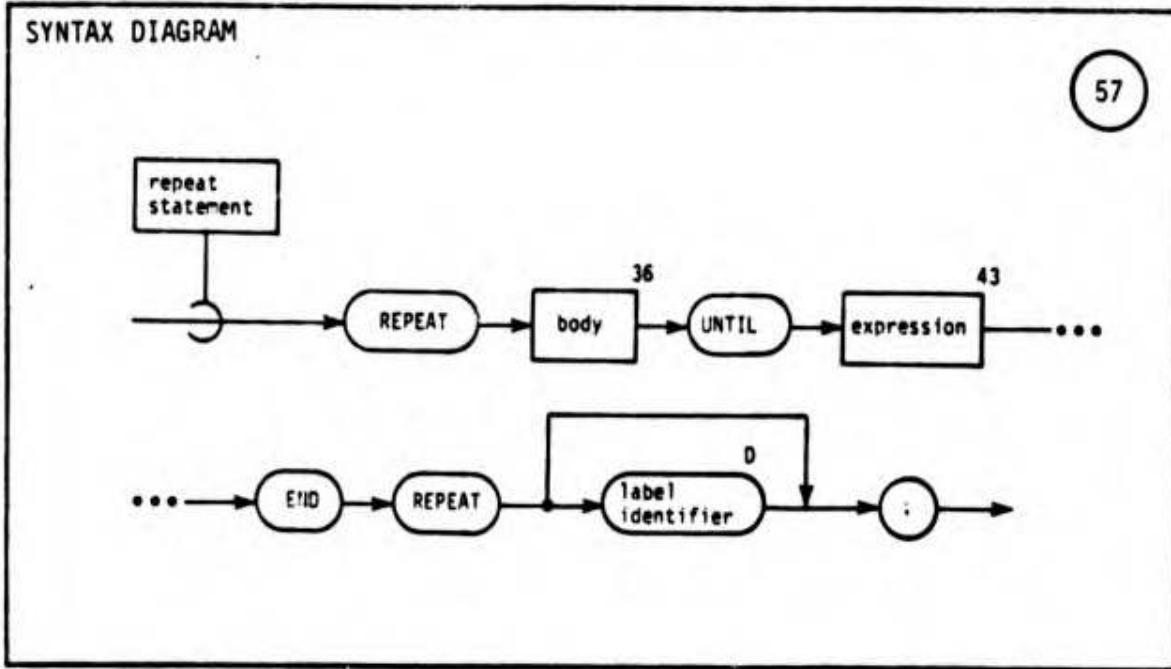
The <body> of the <while statement> is an open scope; it is thus not possible to transfer control into a <while statement>.

The <expression> must have BOOLEAN type.

The <statement> is executed as follows:

```
WHILE_STMT: BEGIN  
    L: IF NOT expression THEN EXIT WHILE_STMT; END IF L;  
    BEGIN body END;  
    GOTO L;  
END WHILE_STMT;
```

9.6.2 Repeat Statement.



The <repeat statement> provides a means for executing a <body> one or more times, as long as some condition (tested after each iteration) is false.

The <body> of the <repeat statement> is an open scope; it is thus not possible to transfer control into a <repeat statement>.

The <expression> must have BOOLEAN type.

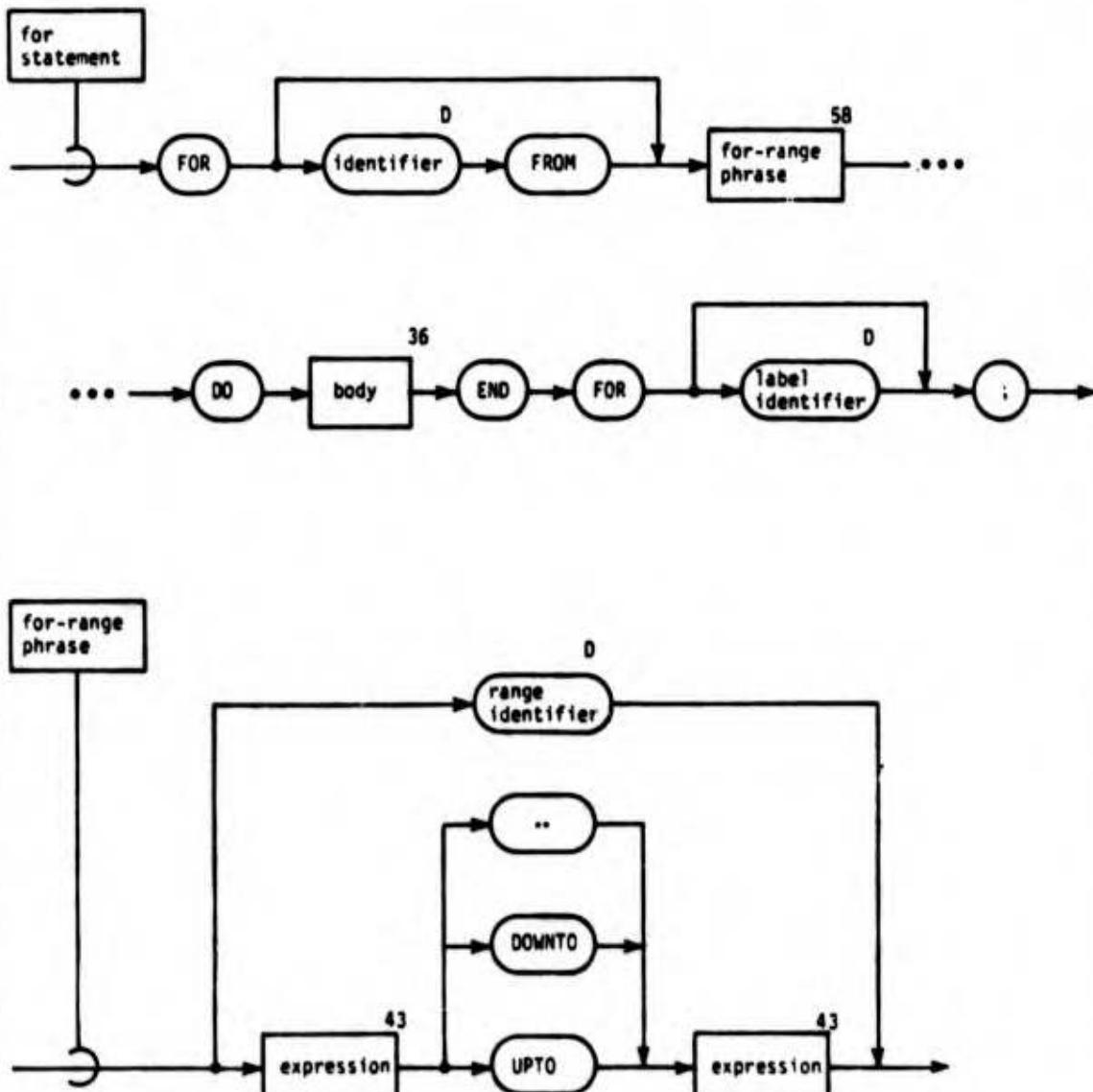
The <statement> is executed as follows:

```
REPEAT STM: BEGIN  
  L: BEGIN body END L;  
  IF NOT expression THEN GOTO L; END IF;  
 END REPEAT _STM;
```

9.6.3 For Statement.

SYNTAX DIAGRAM

58



The <for statement> provides a means for executing a <body> zero or more times, with the number of iterations established before the <body> is executed.

The <for statement> is an open scope. The identifier following FOR, if present, is called the for variable and is implicitly declared within this scope. It is not possible to transfer control into a <for statement>; nor is it possible to obtain the value of the for variable outside the <for statement>. Moreover, inside the <body> the for variable can only be used as a constant.

The "identifier FROM" phrase may be omitted if there is no reason to refer to the for variable within the <body>. For example, "FOR 1..10 DO body END FOR;" will cause body to be executed ten times.

The <for-range phrase> defines a range of values over which the <body> will be executed. The <body> is executed once for each value in the range; if the range is empty, the <body> is not executed at all. This range must be either integers or ordered enumeration type elements. If the "identifier FROM" phrase is present, then this has the effect of declaring the for variable to be of the type given by the range (the range of the variable, however, will not necessarily be the same as that implied by the <for-range phrase>).

Execution of the <for statement> is said to be either ordered or unordered, depending on the <for-range phrase>. If this phrase is either "range identifier" or "expression .. expression", then the execution is unordered. If the phrase is "expression UPTO expression" or "expression DOWNTO expression", then execution is ordered.

In the unordered case, the order in which the range is traversed is undefined.

In the ordered case, the order in which the range is traversed is either ascending (if "UPTO" is coded) or descending (if "DOWN TO" is coded). In the ascending case the for variable starts at the lower bound of the range and is incremented by 1 (or is replaced by its successor) after each iteration. In the descending case the for variable starts at the upper bound of the range and is decremented by 1 (or is replaced by its predecessor) after each iteration.

As an illustration of the semantics, the following <begin statement> has the same effect as the <for statement> "FOR id FROM expr1 UPTO expr2 DO body END FOR;" where expr1 and expr2 are integer-valued <expression>s:

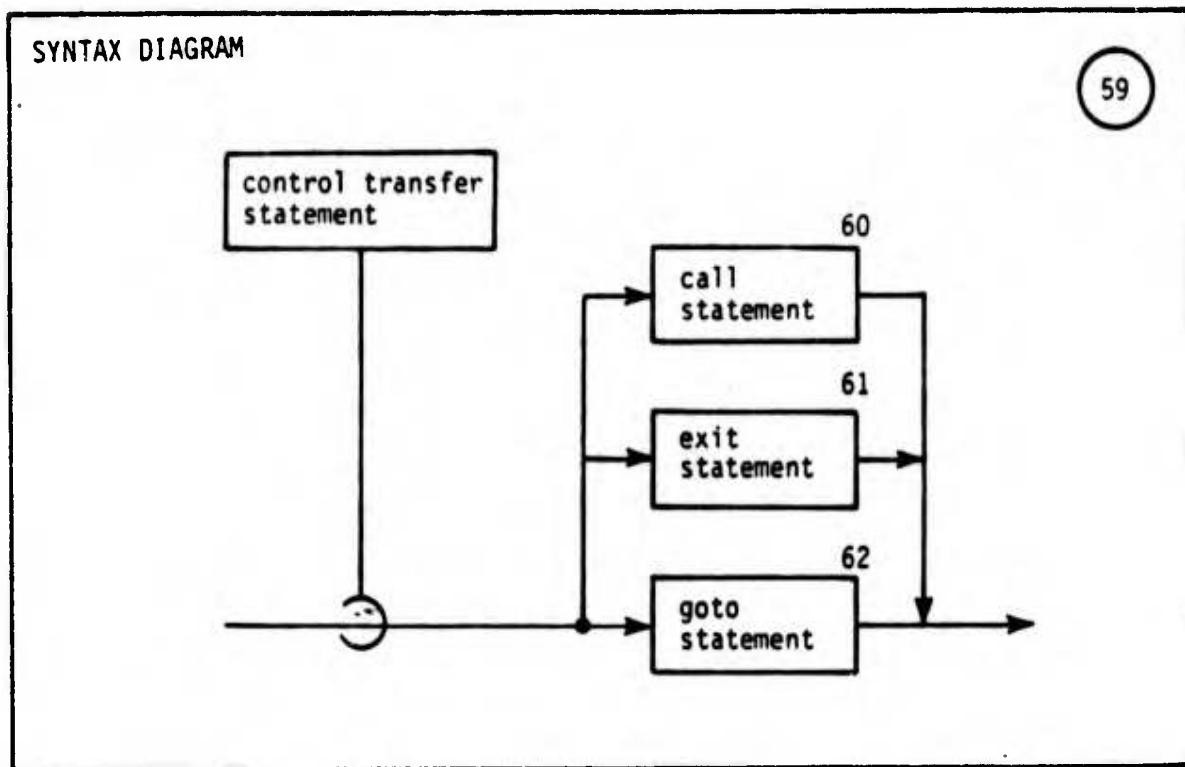
```
FOR STM: BEGIN
  CONST LOW: FIXED(MIN_INTEGER..MAX_INTEGER) INIT expr1;
  CONST HIGH: FIXED(MIN_INTEGER..MAX_INTEGER) INIT expr2;
  IF LOW <= HIGH THEN
    VAR id: FIXED(LOW..HIGH+1) INIT LOW;
    WHILE id <= HIGH DO
      BEGIN !READONLY id; body END;
      id := id+1;
    END WHILE;
  END IF;
END FOR STM;
```

Example:

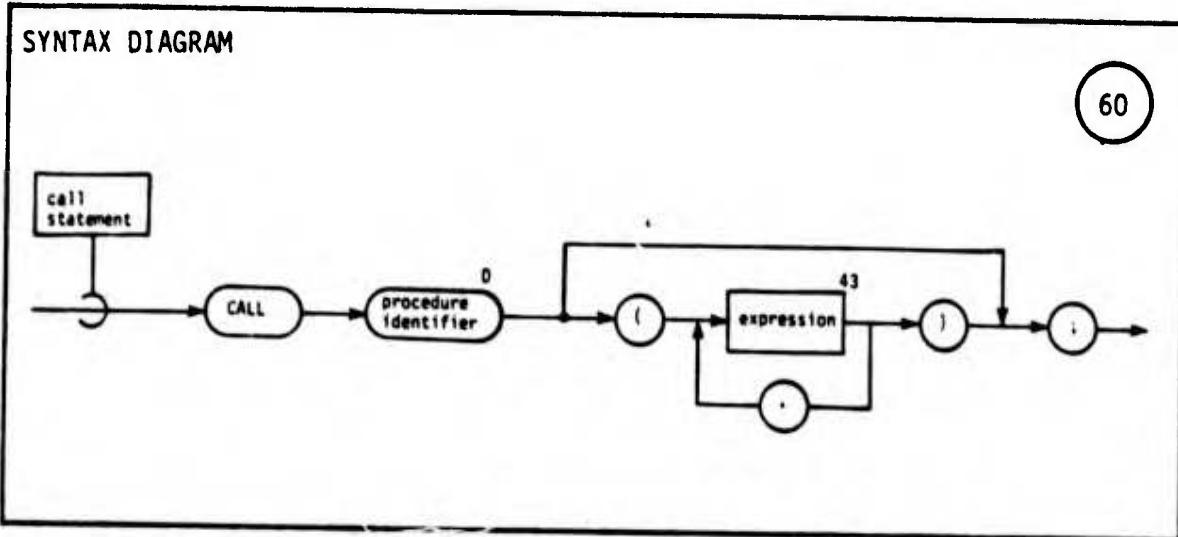
```
TYPE COLOR: ORDERED ENUM(RED, ORANGE, YELLOW, GREEN,
  BLUE, VIOLET);

RANGE WARM_COLOR: RED..YELLOW;
FOR WC FROM WARM_COLOR DO...END FOR;
FOR C FROM ORANGE UPTO BLUE DO...END FOR;
```

9.7 Control Transfer Statements.

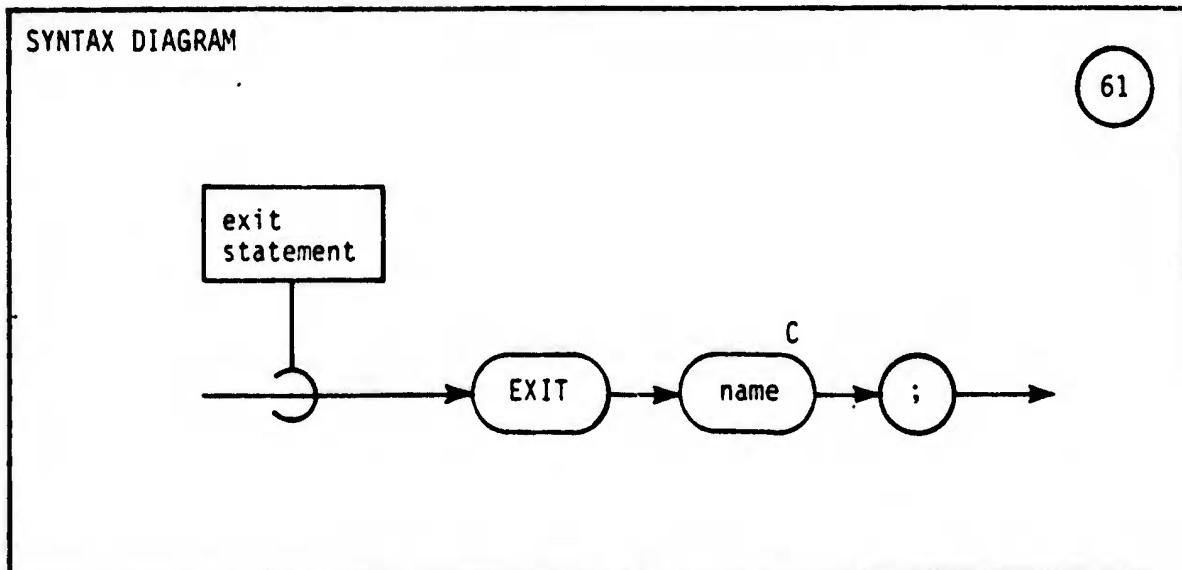


9.7.1 Call Statement.



A <call statement> is used to invoke a procedure. The semantics for this <statement> (i.e., the rules for parameter passing and aliasing) are described in Sections 7.1.4 and 7.2.2.

9.7.2 Exit Statement.



The execution of the <exit statement> transfers control out of a scope. The name following EXIT must either be that of a routine or path, or else the identifier of a label which precedes an open scope. In any of these cases, the scope that is named (we refer to this scope as S) must contain the <exit statement>. Moreover, there must not be any closed scope S' such that S contains S' and S' contains the <exit statement>. (This ensures that an <exit statement> may only transfer control out of a routine or path by explicitly mentioning the routine name or path identifier.)

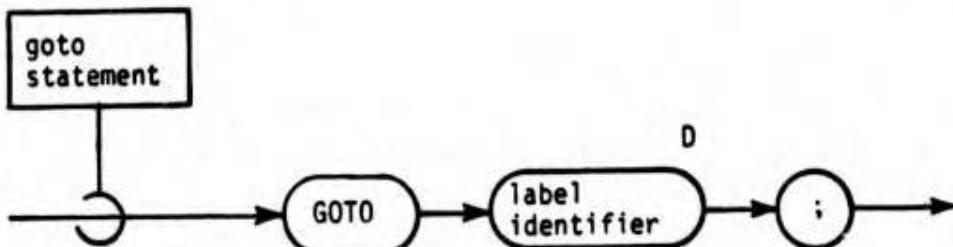
The semantics for exiting from a routine (i.e., the rules for control transfer and assignment to RESULT parameters) are given in Sections 7.1.4 and 7.1.5. Exiting from a path has the effect of terminating that path; see Section 12.2.

If the <exit statement> names the label of an open scope, then the effect of the <statement> is to cause execution to continue at the point following the open scope.

9.7.3 Goto Statement.

SYNTAX DIAGRAM

62



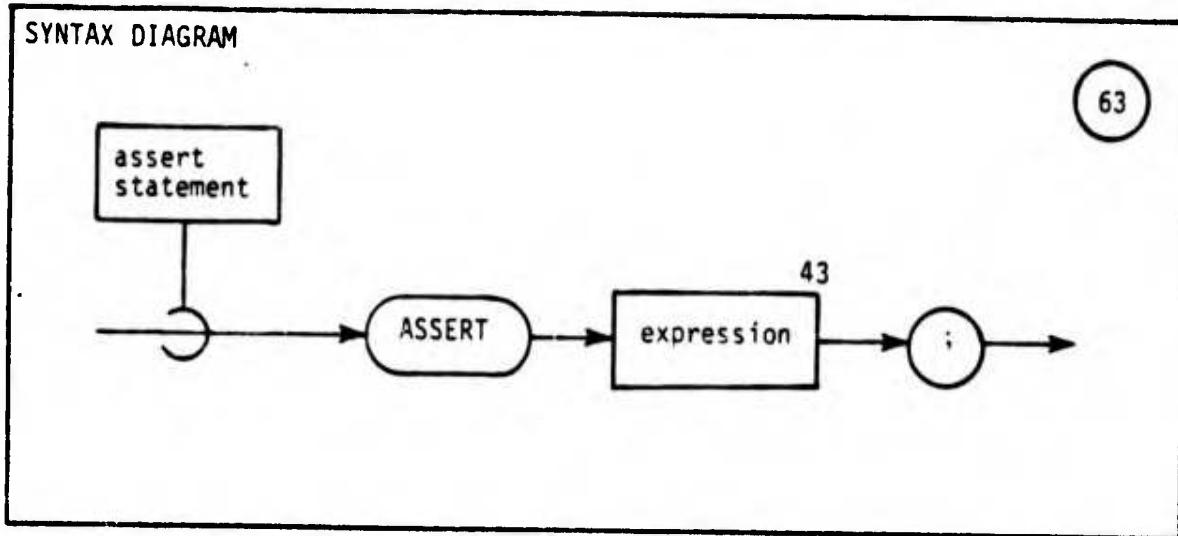
Execution of a <goto statement> transfers control to a specified <statement>. The <goto statement> may be used to transfer control out of an open scope or to another <statement> in the same immediately containing scope. It may not be used to transfer control out of a closed scope. It cannot transfer control into any scope from outside, since the label identifier would not be known.

It should be noted that the "high-level" control constructs provided by REDL--the conditional, loop, exit and exception <statement>s--remove much of the need for using a <goto statement>.

Examples:

```
Legal: BEGIN ...GOTO ALPHA; ...ALPHA:... END;  
      BETA: BEGIN  
      ...  
      IF cond THEN GOTO BETA; ELSE GOTO GAMMA; END IF;  
      ...  
      END;  
      GAMMA:...  
  
Illegal: DELTA: ...BEGIN PROCEDURE P; ...GOTO DELTA;  
        ... END PROCEDURE P; END;  
        /*It is illegal to transfer control out of  
        a routine*/  
        BEGIN ...EPSILON:... END; ...GOTO EPSILON; ...  
        /*It is illegal to transfer control into a scope*/
```

9.8 Assert Statement.



The <assert statement> is provided so that the user can specify, at various points in the <program>, conditions which must be true if the <program> is correct. The <statement> is useful for debugging, for formal proofs of program correctness, for optimization, and for documentation.

The <statement> "ASSERT expression;" has the same effect as "IF NOT expression THEN RAISE X_ASSERT; END IF;" but the distinctive syntax of the <assert statement> makes clearer to the reader and compiler its specialized purpose.

10.0 INPUT/OUTPUT

10.1 Basic Properties.

REDL's approach to I/O is embodied in two principles:

(1) The provision of a FILE type generator and a set of library routines so that general application-level I/O can be realized via standard protocols across all implementations.

(2) The provision of a set of definitional facilities so that device-dependent low-level and specialized application-level routines can be defined within the language and encapsulated in implementation-specific libraries.

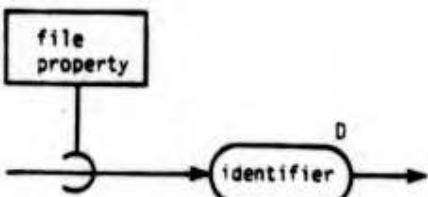
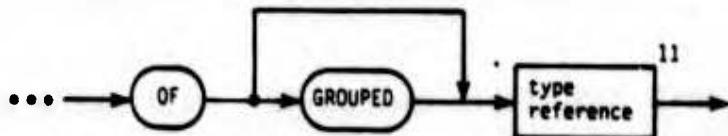
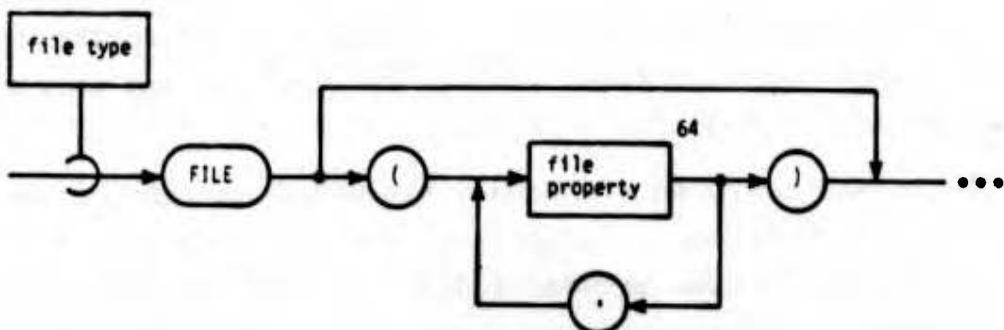
The general-purpose standard library I/O facilities are based heavily on Pascal. Differences are the inclusion in REDL of the association between a declared file and the physical medium which it represents (specified in the OPEN procedure), the provision in REDL for random-access file organization, and the use of "status" parameters to reflect the outcome of the routines.

The definitional facilities mentioned in (2) above are part of REDL's machine-dependent and compile-time features. These facilities may be used to define both the general-purpose libraries and the implementation-specific libraries. Moreover, asynchronous I/O can be realized via REDL's multipath facility.

10.2 Files.

SYNTAX DIAGRAM

64



The file type generator, like the array type generator, is used for declaring aggregate types whose objects are "homogeneous" (i.e., each component having the same type and representation) The differences between files and arrays are (1) the number of components of a file may change dynamically, and (2) there is a physical association between a file and a secondary storage device or a peripheral I/O unit. To obtain heterogeneous files, the programmer may use the UNION or MRECORD generator.

The type identified in the <type reference> is called the component type. A data object whose type is a file type is called a file. A component of a file is called a file-record. (The term traditionally used is simply "record", but this has another meaning in REDL.)

The <file property>s in the parenthesized list will depend on the file organizations supported by the target machine environment; they will include SEQUENTIAL, RANDOM, BUFFERED, and UNBUFFERED. In the absence of an explicit list, the defaults are SEQUENTIAL and BUFFERED (i.e., the conventions adopted by Pascal).

The identifier NIL_FILE denotes a value of each file type. The only routine which may be passed the value NIL_FILE is the procedure OPEN.

The library routines which take a file parameter must have a means of examining the "internal structure" of the storage comprising the file. (As an analogue, if the floating point operations were to be written in the language as library routines, they would have to be able to access a FLOAT object's mantissa and exponent separately.) To allow for this, REDL

provides the FILE_REP type, whose meaning is implementation-defined and whose use is restricted to programs which are machine-dependent. Typically, FILE_REP will be a record type with such fields as FILE_ORGANIZATION, DEVICE_ADDR, and COMPONENT_SIZE.

If T is either a pointer type or a file type, then it is illegal for T to be used as the type of a file-record (or as the type of a component of a file-record).

10.3 File-Record I/O Routines.

10.3.1 OPEN(F,P1,...,Pn,S) and CLOSE(F,S) (Procedures)

The procedures OPEN and CLOSE each take a VAR file parameter F and a RESULT integer parameter S. OPEN additionally takes parameters P1,...,Pn of implementation-determined number and types; these parameters give the specification of the file or device as known to the target system and define such characteristics as whether this is a new or old file; whether the file is to be open for input, output, or update; type density; file record blocking. Upon return from OPEN, the file parameter will denote a file, and, for both OPEN and CLOSE, the integer parameter will reflect the status (i.e., success or failure) of the call. The returned status value will be in accordance with the following conventions:

- (1) 0 designates unqualified success.
- (2) A positive value designates success but encodes an implementation-defined qualification.
- (3) A negative value designates failure and encodes the reason for failure in an implementation-defined manner.

10.3.2 READ(F,R,S) (Procedure)

F is a VAR file parameter, R is a RESULT parameter whose type is F's component type, and S is a RESULT integer parameter. The effect of READ, if successful, is to copy the next file-record of file F into R, advancing the current position of the file. S is a status parameter as described above. If EOF(F) is TRUE, a failure status will be given.

10.3.3 WRITE(F,R,S) (Procedure)

F is a VAR file parameter, R is a CONST parameter whose type is F's component type, and S is a RESULT integer parameter. The effect of WRITE, if successful, is to copy R as the next file-record of file F, advancing the current position. S is a status parameter as described above. If EOF(F) is FALSE, a failure status will be given and no output will be performed.

10.3.4 EOF(F) (Function)

EOF is a function which takes a file parameter F and returns a BOOLEAN value: TRUE if F is positioned at the end-of-file (i.e., with no file-record following the current file position), and FALSE otherwise. If F is not ready to be checked for end-of-file position (e.g., if it has not been OPENed), the X_FILE exception is raised.

10.3.5 REWIND(F,S) (Procedure)

F is a VAR file parameter, and S is a RESULT integer parameter. The effect of REWIND, if successful, is to set the current position of F to precede the first file-record (if any) of F. S is a status parameter as described above.

10.3.6 GET_POSITION(F,I,S) (Procedure)

F is a VAR file parameter, and I and S are RESULT integer parameters. The effect of GET_POSITION, if successful, is to return in I the current position of file F. S is a status

parameter as described above. The first file-record of a file has position 1, the (nonexistent) file-record preceding the first has position 0, and the (nonexistent) file-record following the last has position -1.

When a file is opened, it may be positioned at 0, at 1, or at -1 depending on the manner of opening.

The READ procedure advances the position before reading. A file open for sequential input will therefore be opened at position 0. The procedure WRITE writes into the current position and then advances that position. A file open for output will therefore be positioned initially at -1. (An existing file may be extended in this way. When a newly created file is positioned at -1, the first file-record to be written will be at position 1.)

10.3.7 SET_POSITION(F,I,S) (Procedure)

F is a VAR file parameter, I is a CONST and S a RESULT integer parameter. The effect of SET_POSITION, if successful, is to position a file at the beginning or end of the file-record sequence (by use of the position designations 0 and -1) or, if the file is open for random access, at other positions. The desired position is specified by the position parameter I and information about success or failure of the call is returned by the status parameter S.

10.3/8 OVERWRITE(F,R,S) (Procedure)

The effect of OVERWRITE is the same as WRITE, with the important difference that EOF(F) must be FALSE (i.e., OVERWRITE is used to overwrite an existing file-record as opposed to extending the file). OVERWRITE may only be used for random access files open for update.

10.4 Textfile I/O.

10.4.1 The TEXT_FILE Type.

Analogous to Pascal, REDL provides the built-in type TEXT_FILE:

```
TYPE TEXT_FILE: FILE OF GROUPED CHAR;
```

A data object of type TEXT_FILE is called a textfile. A textfile may be treated as either a sequence of CHARs, in which case the routines of the preceding section apply, or as a sequence of CHARs partitioned into lines by the appearance of implementation-defined end-of-line CHARs as separators. For example, if the line separator is the <CR><LF> sequence and the contents of a textfile F are 'ABC<CR><LF>DEFG<LF>HI<CR><LF>JKL<CR><LF>' then F may be regarded as containing three lines: 'ABC', 'DEFG<LF>HI', and 'JKL'.

10.4.2 Line-Oriented I/O Routines.

10.4.2.1 READLN(F,S). F is a VAR file parameter, and S is a RESULT integer parameter. The effect of the READLN(F,S) procedure, if successful, is to advance the current position of the file to the point immediately following the next end-of-line separator. S is a status parameter.

10.4.2.2 WRITELN(F,S). F is a VAR file parameter, and S is a RESULT integer parameter. The effect of the WRITELN(F,S) procedure, if successful, is to write out an end-of-line separator at the current position of the file, advancing the current position. S is a status parameter. If EOF(F) is FALSE, a failure status will be given and no output will be performed.

10.4.2.3 EOLN(F). EOLN is a function which takes a file parameter F and returns a BOOLEAN value: TRUE if F is positioned at (the first CHAR of) an end-of-line separator, and FALSE otherwise. If F is not ready to be checked for end-of-line position (e.g., if it has not been OPENed), the X_FILE exception is raised.

10.4.3 The Standard Files INPUT and OUTPUT.

The textfiles INPUT and OUTPUT usually represent the standard I/O media of a computer installation (such as the card reader and the line printer). Because of the frequency of their use, they are considered as "default values" in textbox routines when the textbox is not explicitly indicated. Thus, if CH is an object of type CHAR and S is a W-valued integer object, the routine invocations READ(CH,S), WRITE(CH,S), READLN(S), WRITELN(S), EOF, and EOLN may be used instead of READ(INPUT,CH,S), WRITE(OUTPUT,CH,S), READLN(INPUT,S), WRITELN(OUTPUT,S), EOF(INPUT), and EOLN(INPUT).

By using INPUT or OUTPUT in a call on OPEN, the user may associate with these file names either the media defined for them by the implementation, or other media desired for the specific application.

10.4.4 Formatted I/O.

REDL provides for formatted I/O by allowing STRINGS (and not just CHARs) to be read from or written to textfiles and by establishing library procedures for explicit conversions between STRING and the other built-in types.

If F is a textbox, then the procedures READ(F,R,S) and WRITE(F,R,S) may be called with R having type STRING instead of CHAR. The value of R.SIZE will determine the number of CHARs read or written.

The procedures for conversion to and from STRING include BOOLEAN_TO_STRING, STRING_TO_BOOLEAN, FIXED_TO_STRING, STRING_TO_FIXED, FLOAT_TO_STRING, STRING_TO_FLOAT, BITS_TO_STRING, and STRING_TO_BITS. As an illustration of a calling sequence for these routines, the FIXED_TO_STRING conversion takes a CONST parameter X of type FIXED (the value to be converted), a RESULT parameter R which is a STRING(*), a CONST parameter SC of type FIXED (the scale to be exhibited in the output), and a RESULT parameter S which is an integer (the status of the conversion in terms of success or failure). If X has the value 1B-6 (decimal value 0.015625), R is a STRING(6), and SC is 1, then the effect of FIXED_TO_STRING(X,R,SC,S) is to assign to R the STRING '0.0156' and to assign to S a status value indicating qualified success (the qualification being the loss of rightmost digits due to insufficient STRING size in R).

10.5 Library Definition Facilities.

The definitional facilities needed so that the library routines (both low-level and application-level) can be defined in REDL are described in other portions of this document. In summary, the ANY formal parameter specification (Section 14.5.1), the CALLER_ASSERT and REDECLARE directives (Sections 14.5.2 and 14.5.4), compile-time type interrogation predicates (Section 14.5.3), and the FILE REP type (Section 10.2.1) are applicable. Their use is illustrated in Appendix A.

11.0 EXCEPTION HANDLING FACILITIES

11.1 Introduction.

In embedded-computer applications, the occurrence of exceptional conditions during program execution must be treatable in an orderly way. REDL provides a group of capabilities, referred to as exception-handling facilities, to enable programs to specify the processing which is to occur when exceptional conditions arise.

In REDL, an exception may be regarded as a named two-state entity, which is in the raised state or the lowered state. The execution of a <raise statement> for a given exception places that exception in the raised state, and causes execution control to be transferred to a handler, in a manner described below. Exceptions are initially in the lowered state, and are implicitly placed in the lowered state at the completion of the transfer of control to the handler.

REDL provides a set of pre-defined exceptions. The compiler will supply code necessary to raise these exceptions under the conditions defined for them. Exceptions may also be defined by the user: user-defined exceptions are raised only by user-written <raise statement>s.

The pre-defined exceptions in REDL are listed in Table 11-1.

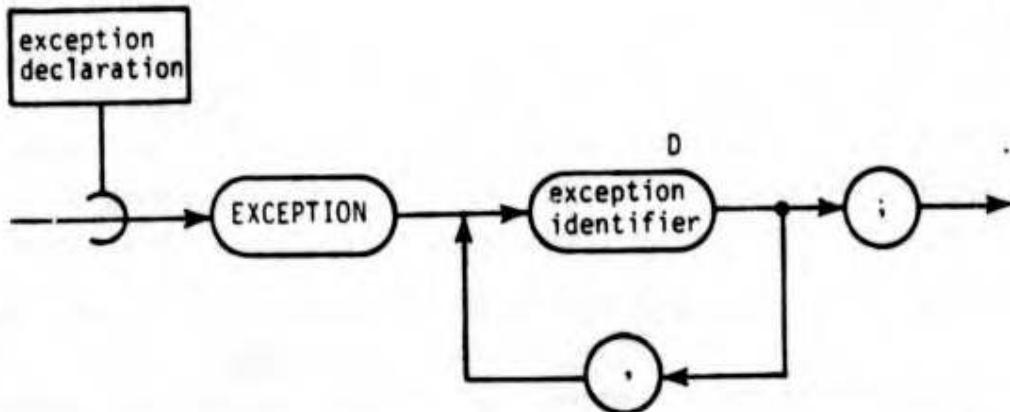
X_ALIAS	X_RANGE
X_ASSERT	X_SELECT
X_FILE	X_SIZE
X_INIT	X_SUBSCRIPT
X_MULTIPATH	X_TAG
X NIL_POINTER	X_TERMINATE
X_NON_FREEABLE	X_UNDERFLOW
X_OVERFLOW	X_ZERO_DIVIDE

Table 11-1. Pre-Defined Exceptions in REDL

11.2 Exception Declarations.

SYNTAX DIAGRAM

65

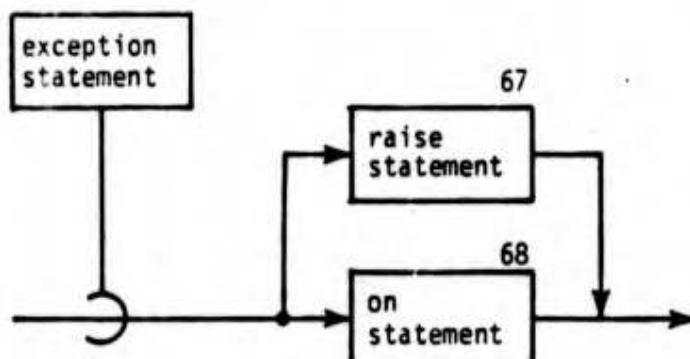


Identifiers of exceptions obey the same lexical scope and lifetime rules as other identifiers. A user-written declaration defines one or more new exceptions, and associates a specified identifier with each.

11.3 Exception Statement.

SYNTAX DIAGRAM

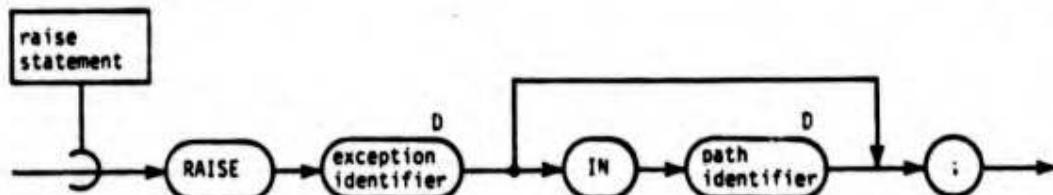
66



11.3.1 Raising of Exceptions: The Raise Statement.

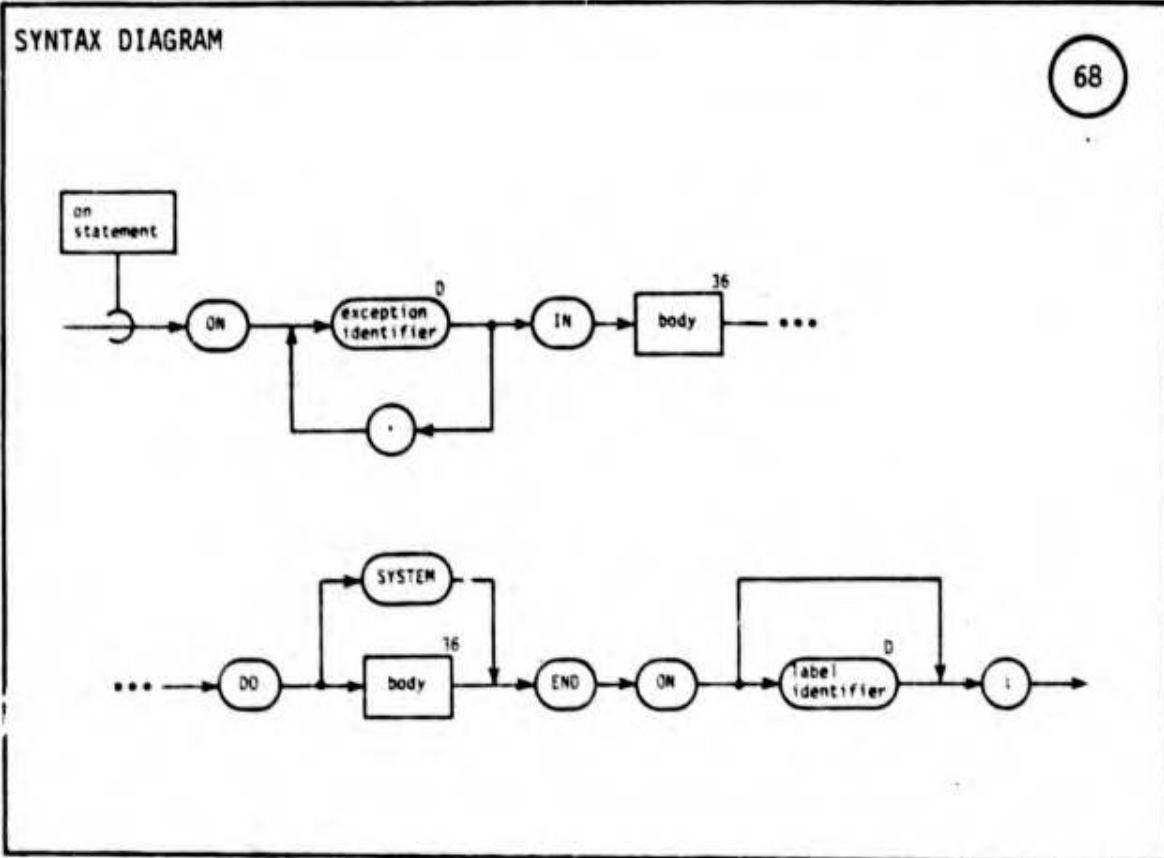
SYNTAX DIAGRAM

67



An exception is placed in the raised state by the execution of a <raise statement> which specifies its identifier, within the scope of its declaration. Pre-defined exceptions may also be raised by execution of <raise statement>s contained in the code of the routines which implement certain system-defined operations. When the optional IN-phrase is coded in the <raise statement>, the path in which the exception is to be raised is specified explicitly by the path identifier given. (The only exception permissible in such a context is X_TERMINATE; see Section 12.2.1.) Otherwise, the exception is raised in the path which executes the <raise statement>.

11.3.2 Handling of Exceptions: The On Statement.



The execution of a <raise statement> both raises the named exception, and causes a transfer of control to a so-called handler for that exception. Upon entry of the handler, the exception is implicitly lowered.

A handler for an exception is associated with a section of <program> by means of an <on statement>. An <on statement> contains three user-specified components: (1) a specification of one or more exceptions either explicitly through a list or implicitly through X_ANY, (2) the section of <program> with which the handler is to be associated (the <body> following IN), called the guarded <body>, and (3) a handler (the part following DO), to which control is to be transferred should one of the exceptions in the list be raised during execution of the guarded <body>.

The handler may be a <body>, or, alternatively, the system-defined handler may be designated by coding the word SYSTEM in place of a <body>. Both the guarded <body> and the handler are open scopes.

As will be seen below, the X_ANY identifier has the effect of enabling the containing <on statement> to handle any exception raised. If X_ANY appears, then this must be the last identifier in the list following ON.

The handling of an exception depends on the dynamic history of the path in which it is raised. The remainder of this section provides an informal description of the semantics of exception handling.

Corresponding to a path in execution are a sequence of executable program constructs whose executions have begun but not yet ended; e.g., routines which have been called but from which control has not returned, <expression>s whose evaluations have begun but not yet finished. Let the sequence of these unfinished constructs be listed in a "stack" as follows:

first ... R ... OS_iG₁ ... R ... OS_nG_n ... last

where "first" is either the main procedure or a <path clause> of a <fork statement>, where "last" is the executable construct which raised the exception, where the R's denote routines, and where the OS_iG_i denote <on statement>s whose guarded <body>s are still in execution.

An <on statement> potentially provides a handler for the exception if its guarded <body> is on the "stack" and if it either contains X_ANY or explicitly lists the exception. If the "stack" contains one or more potential handlers for the exception, then the most recently invoked (i.e., the rightmost) is chosen to handle the exception.

In this case, the executions of all elements on the stack to the right of the chosen OS are terminated abnormally, control of the path is transferred to the handler part of the chosen <on statement>, and the exception is lowered. The resulting "stack" looks like this (where H_n denotes the handler of OS_n):

first ... R ... $OS_1 GB_1 \dots R \dots OS_n H_n$

In the event that there is no potential handler for the exception on the "stack", all the executable elements of the "stack" are terminated abnormally, including "first".

In this case, if "first" was the main procedure whose execution began the execution of the linked <program>, then the system-defined handler will be executed and then the <program> will terminate. If "first" was a <path clause> of a <fork statement>, then the path is said to have terminated "on an unhandled exception".

When all of the paths of a <fork statement> have terminated, an exception will be raised in the path containing the <fork statement> under the following circumstances:

(1) If the X_TERMINATE exception had been raised in the path containing the <fork statement> during its fork-wait, then X_TERMINATE remains raised.

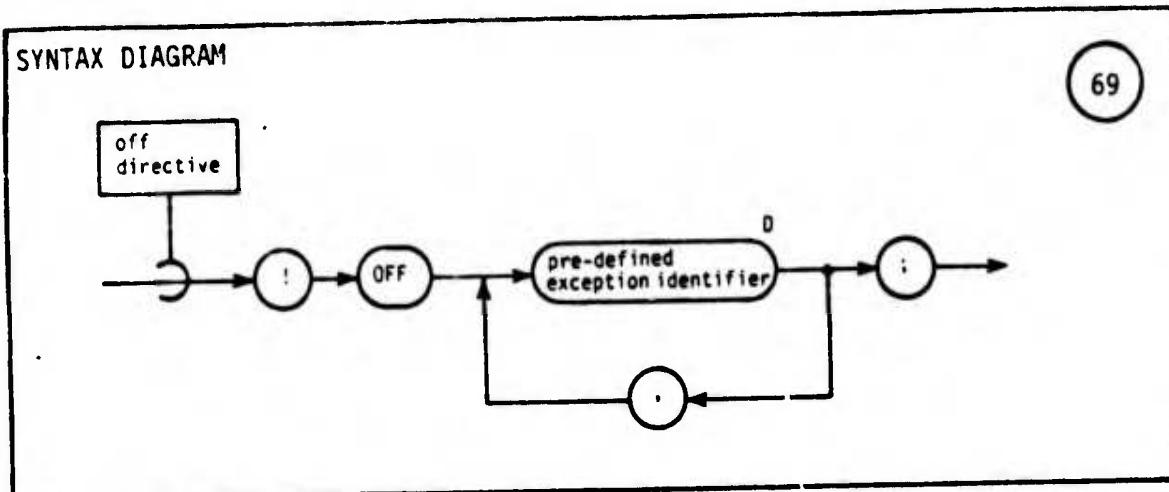
(2) Otherwise, if one or more of the paths of the fork have terminated "on an unhandled exception", an exception will be raised in the path containing the <fork statement> as follows:

(a) If exactly one path so terminated, then the exception on which it terminated will be raised;

(b) If more than one such path so terminated, then the X_MULTIPATH pre-defined exception will be raised.

An exception so raised in the path containing the <fork statement> is handled in that path as described in this section.

11.4 Suppression of Exception Checking: The Off Directive.



Some pre-defined exceptions require space and/or time overhead at execution time for checking. The <off directive> is provided in REDL to allow the user to inform the implementation of a desire to minimize the checking overhead for those exceptions listed in the <directive>, possibly at the expense of safety.

The effect of an <off directive> is to suppress the generation of code by the compiler which will check for the conditions which cause the designated exceptions to be raised. The scope of suppression is the scope in which the <directive> appears and all inner scopes excepting those in which it is overridden by an <on statement> referencing the same exception identifier(s). The behavior of a <program> is undefined when a condition occurs whose detection is suppressed.

Although checking code is suppressed when an <off directive> appears, it is still possible for an exception identified in the <directive> to be raised in the scope containing the <directive>. For example, the hardware might raise the exception directly, the user might raise the exception via an explicit <raise statement>, or a called routine (in whose scope the checking is turned on) might return abnormally (i.e., without handling the exception). In such cases the behavior of the <program> is as though the exception identifier had not occurred in the <off directive>.

11.5 Determination of Raised Exception.

Since an <on statement> may list a set of exceptions which it will handle (and may also specify that any exception will be handled), it is useful for the handler of an <on statement> to be able to determine which exception was in fact raised. The <select statement> may be used for this purpose. Before describing the semantics, we provide the following example:

```
TYPE ARITH_STATUS: ENUM(OK, OVERFLOW, UNDERFLOW);  
VAR S: ARITH_STATUS INIT OK;  
VAR A,B,X: FLOAT(5, -1.0E7 .. 1.0E7) INIT 0.0;  
:  
ON X_OVERFLOW, X_UNDERFLOW, X_ZERO_DIVIDE, X_ANY  
    IN      X := (A*F(B) - B*F(A))/(F(B) - F(A));  
    DO      SELECT LAST_EXCEPTION FROM  
            CASE X_OVERFLOW, X_ZERO_DIVIDE => S := OVERFLOW;  
            CASE X_UNDERFLOW => S := UNDERFLOW;  
            OTHERWISE           => RAISE LAST_EXCEPTION;  
    END SELECT;  
END ON;
```

In this example, the raising of X_OVERFLOW, X_UNDERFLOW, or X_ZERO_DIVIDE during execution of the guarded <body> will cause the appropriate alternative of the <select statement> to be executed. If any other exception is raised, the OTHERWISE alternative is executed. In the above example, this exception is then raised again. (The determination of a handler in this case is not illustrated in the example.)

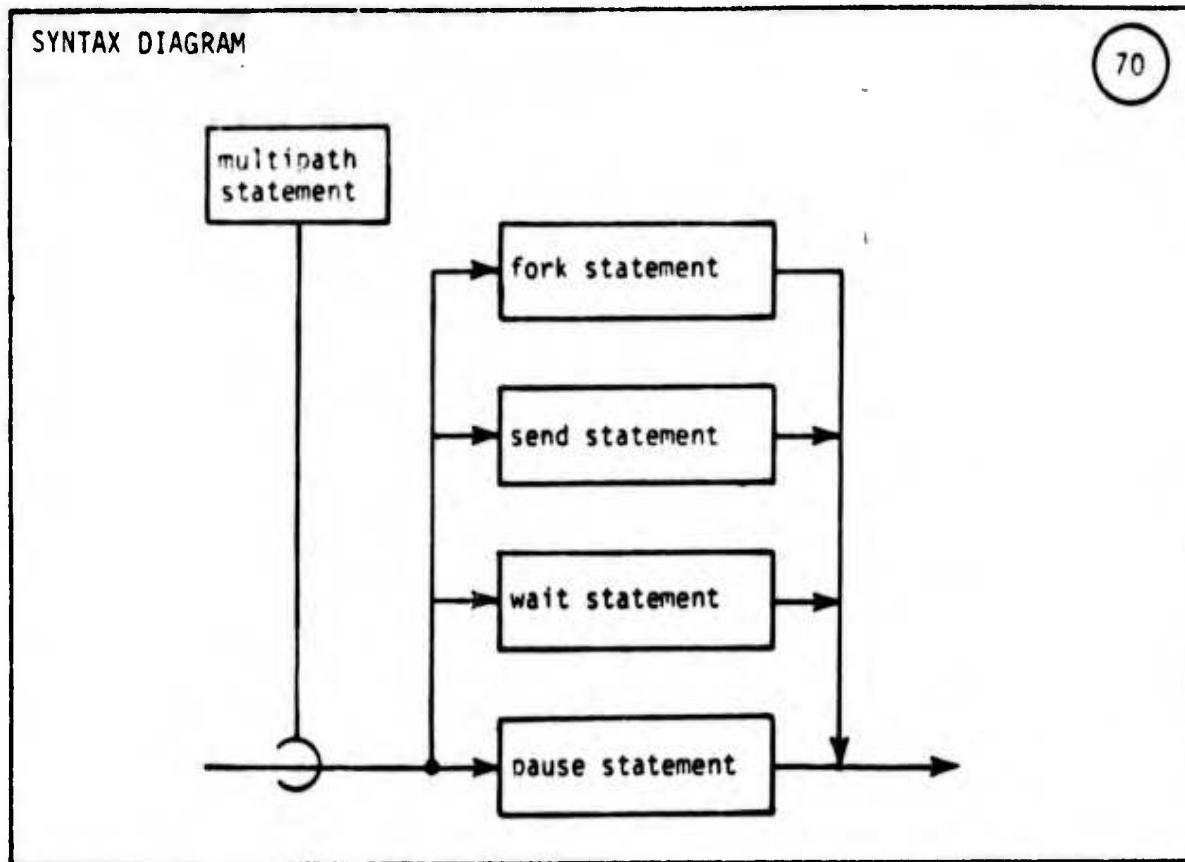
In general, to perform a discrimination upon the exception which caused a handler to be invoked, the user supplies in the <body> of the handler a <select statement> with the special identifier LAST_EXCEPTION following SELECT. On entry to the handler, LAST_EXCEPTION is bound to the exception which caused the handler to be invoked. The semantics of selecting an alternative is the same as that described in Section 9.5.2.1,

with the qualification that the <case label>s must all be exception identifiers, and that <case label range>s are prohibited.

We note that the only permitted occurrences of the identifier LAST_EXCEPTION are within a handler <body> of an <on statement>. It may be used in such a <body> either as the object of selection in a <select statement> (i.e., as the <expression> following SELECT), or in a <raise statement>. Both uses are illustrated in the example earlier in this section.

12.0 MULTI-PATH AND REAL-TIME FACILITIES

12.1 Introduction.



70

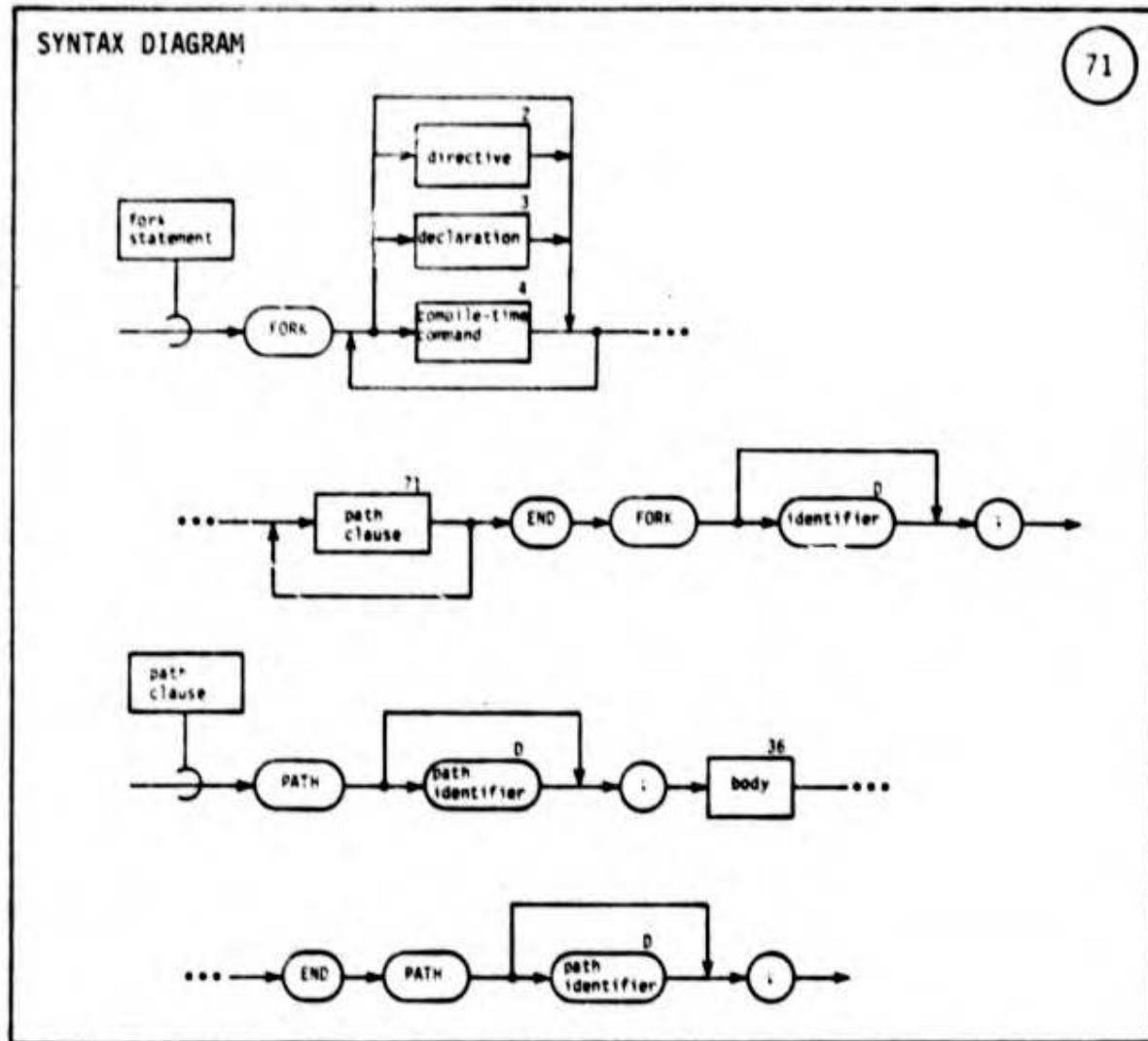
REDL contains a facility for defining parallel control paths (the <fork statement>); for allowing paths to cooperate in mutually exclusive accesses to shared data (the monitor capsule); for permitting paths to synchronize their execution, either time-independently (via events) or time-dependently (via the <pause statement>); and for dealing with failures on any control path (the X_TERMINATE exception).

Programming examples for the multi-path and real-time facilities may be found in Appendix A.

12.2 Paths.

The unit of execution in REDL is the path. The execution of a REDL <program> constitutes a main path. Other paths capable of parallel execution come into existence through the execution of <fork statement>s.

12.2.1 Fork Statement.



The <fork statement> has the effect of bringing into parallel (or pseudo-parallel) execution a set of paths, each path corresponding to a <path clause> in the <fork statement>. (Whether execution is truly parallel or instead interleaved depends on whether the number of hardware processors is at least as great as the number of paths.)

A <fork statement> is an open scope. If a <compile-time command> appears before the <path clause>s, it may produce only <directive>s and <declaration>s; see Sections 14.3 and 14.4.

When a <fork statement> is executed, the path in which the <fork statement> exists enters a waiting state (called fork wait) until all the constituent paths of the fork have terminated. The END FORK may thus be regarded as a rejoin point for further sequential execution of the scope continuing the <fork statement>. The execution of the paths of a fork may cause an exception to be raised at the rejoin point, as described in Section 11.3.2.

12.2.2 Execution of a Path Clause.

A <path clause> (or "path", for short) is a closed scope. Thus, it is not possible to reference a label declared outside a path from within the <body> of the path. The path identifier, if given, is declared in the scope of the <fork statement>; thus, the path identifiers which are specified must be distinct. (The path identifier must be specified if the path is to be explicitly terminated, i.e., via the raising of the X_TERMINATE exception.)

The execution of a path begins when the enclosing <fork statement> begins the parallel execution of all of its paths and ends when the path reaches a terminated state. Termination of a path occurs under any of the following circumstances:

- (1) Execution of an <exit statement> specifying the path identifier.
- (2) Execution of the last <statement> in the path <body> (unless this is a <goto statement>).
- (3) Raising an exception which is not handled on the path (see Section 11.3.2).

It is possible for a path P1 to raise the X_TERMINATE exception in another path P2 which is in the waiting state (see Section 12.2.3). The effect depends on the reason for the wait. If P2 has requested a monitor or executed a wait or pause <statement>, then it is removed from the appropriate queue and placed in the running state for the purpose of finding a handler for the X_TERMINATE exception. If P2 is in a fork wait, then the X_TERMINATE exception is raised in each of its constituent paths; when control reaches the rejoin point, a handler for the X_TERMINATE exception raised in P2 is sought.

We note that the X_TERMINATE exception is intended to allow paths to deal with "catastrophic" failures. By raising this exception in a path P when such a failure has been detected, another path can give P the opportunity to perform the necessary "cleanup" before P is terminated.

12.2.3 Path States.

REDL distinguishes between three possible states for a path: running, when it either is in execution by a processor or would be in execution if a processor were available; waiting, when it is postponed until an EVENT is sent, a monitor capsule is released, a specified time interval elapses, or a <fork statement> which it contains reaches its rejoin point; and terminated, when it has terminated. A summary of state transitions appears in Table 12-1.

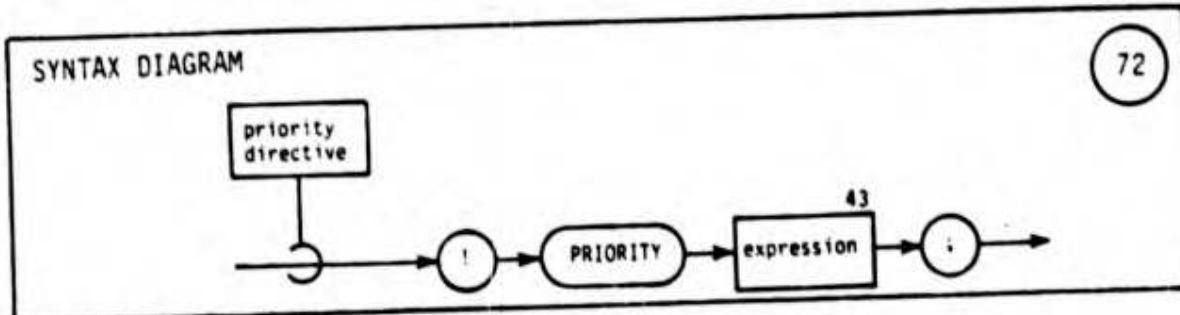
FROM	TO	REASON	REFERENCE
running	waiting	1. Request a monitor capsule held by some other path 2. Execute a <wait statement> 3. Execute a <pause statement> 4. Execute a <fork statement>	12.3 12.4.3 12.5.1 12.2.1
running	terminated	Terminate the path	12.2.2
waiting	running	1. Another path releases a monitor capsule 2. Another path executes a <send statement> 3. Time specified in <pause statement> elapses 4. All paths of a <fork statement> terminate 5. Another path raises X_TERMINATE exception	12.3 12.4.2 12.5.1 12.2.1 12.2.2
waiting	terminated	Cannot occur	--
terminated	running	Cannot occur	--
terminated	waiting	Cannot occur	--

Table 12-1. Summary of Path State Transitions

12.2.4 Priorities of Paths.

12.2.4.1 Introduction. Associated with each path is a value known as its priority. This value is represented as an integer in the range 1.. MIN_PRIORITY, where MIN_PRIORITY is an implementation-defined constant, with smaller integers indicating higher priority. The priority of a path is used when several paths are in contention for the same shared resource--either a user-created resource such as an event or a monitor capsule, or a system-provided resource such as an I/O device or a processor. If several paths are contending for a resource, the path with the highest priority obtains it. Within the same priority level, the path which has been waiting the longest will obtain the resource.

12.2.4.2 Priority Directive.



There are two ways in which a priority value becomes associated with a path. The first is through a <priority directive> in the path <body>. This <directive> specifies an integer-valued <expression> which is the initial priority value for the path. If the <directive> is absent, then the default initial priority value for the path is MIN_PRIORITY.

12.2.4.3 CHANGE_PRIORITY Procedure. The second way in which a priority value is associated with a path is the CHANGE_PRIORITY procedure. This procedure allows a path to modify its own or another path's priority dynamically. The arguments to the routine are a path identifier (designating the path whose priority is to be altered) and an integer in the range 1.. MIN_PRIORITY (the new priority).

12.2.4.4 PRIORITY Function. The PRIORITY function takes a single argument (a path identifier) and returns as its result the priority of the specified path.

12.3 Monitor Capsules.

A monitor capsule (or "monitor", for short) provides a safe mechanism for sharing data among paths and may be used to guarantee mutually exclusive and deadlock-free accesses to shared data. A monitor is a specialized form of capsule (see Section 6.1.1) designated by the MONITOR token, and is like a capsule in all respects except as additionally specified below.

The mutual exclusion facility of a monitor arises from its central property: that at most one path at a time can "hold" a monitor. The remainder of this section will be devoted to discussion of the "requesting", "holding", and "releasing" of monitors by paths.

Some terminology will be helpful. With respect to a given monitor, a path is in one of three states: "holding" the monitor, "waiting" for the monitor, or "ignoring" the monitor. If no path holds a monitor, then the monitor is said to be "free". If a path holds a monitor, then the monitor is said to be "busy", and has associated with it a (possibly empty) queue of paths waiting for the monitor.

A monitor is initially free. A path requests a monitor by invoking a routine exported from the monitor. We shall see in Section 12.4 that a path can also request a monitor through the execution of a send or wait <statement> having a RELEASE clause. When a path requests a monitor and the monitor is free, the path is permitted to hold the monitor and execution of the path continues. When a path requests a busy monitor, the path enters a waiting state and is put on the queue of the monitor.

A path holding a monitor releases the monitor when it exits from the routine of the monitor by which it originally requested the monitor. Monitors can also be released through execution of

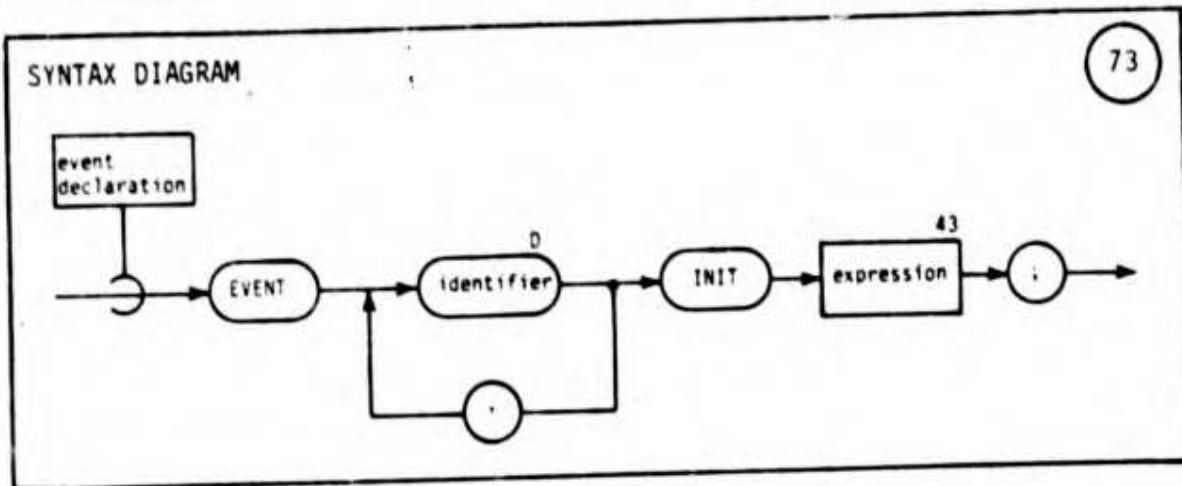
send or wait <statement>s having a RELEASE clause. When a path releases a monitor, and the monitor's queue is not empty, then one of the paths on the queue is chosen (according to their priorities) and the chosen path is then put in the running state and holds the monitor. If a monitor is released and no paths are waiting for it in its queue, then the monitor becomes free.

In order to prevent the possibility of a user's bringing about a deadlock through mutual dependencies between monitors, REDL imposes two simple restrictions (enforceable at compile-time or link-time).

We say that a monitor M can "reach" another monitor N if an exported routine of N is reachable from an exported routine of M (see Section 7.1.7). The first restriction is that, for any two monitors M and N, if M can reach N then it is a compile-time error if N can also reach M. (This rule prevents the deadlock where a path P holds monitor M and waits for monitor N while another path, Q, is holding N and waiting for M.)

The second restriction makes it a compile-time error if a send or wait <statement> having a RELEASE clause names a monitor other than the monitor most immediately containing it lexically. (This rule addresses the situation of nested monitors, M containing N. If path P holds both M and N and then releases M while still holding N, then path Q may enter M (and hold M) and then wait on N. This leads to a deadlock when P requests M after execution of its send or wait <statement>.)

12.4 Events.



Events are provided as the basic low-level language construct for path synchronization. Conceptually, an event consists of an integer value in the range 0..MAX_SENDERS and a queue of waiting paths. (MAX_SENDERS is an implementation-defined constant.) If the integer is positive, then the queue is empty and the integer is the excess of the number of times the event was sent over the number of times the event was waited for. If the integer is zero, then the queue contains the paths waiting for the event.

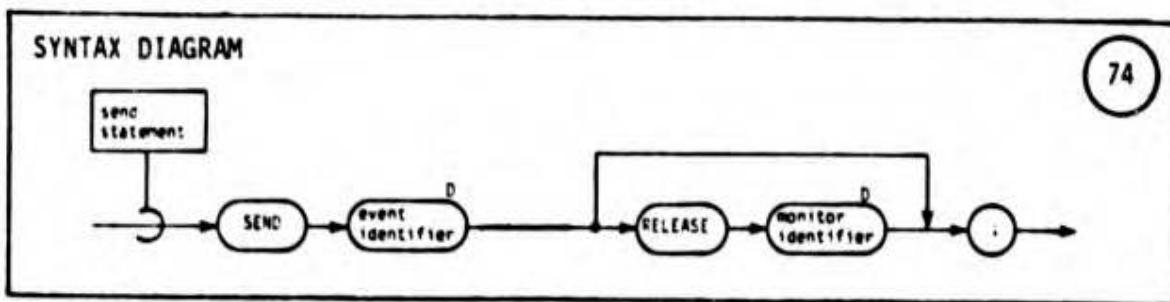
Events behave in many respects like data objects. The send and wait <statement>s modify an event (effectively, they treat it as a W-valued object), whereas the functions SENDERs and WAITERS treat an event as though it were R-valued. As a result, events are regarded as data objects with respect to the rules for capsule export and usage in functions, operators and safe procedures:

- (1) Capsules (including monitors) may not export an event.
- (2) Event identifiers may not be included in a <pervasive directive>.
- (3) If a function, operator, or safe procedure imports an event, then the routine must do so "readonly".

12.4.1 Initialization.

The INIT phrase is mandatory in event declarations and must contain an <expression> whose value is a non negative integer, say N, in the range 0..MAX_SENDERS. The effect of the initialization is to start with an event with an empty queue and a value of N for the integer.

12.4.2 Send Statement.

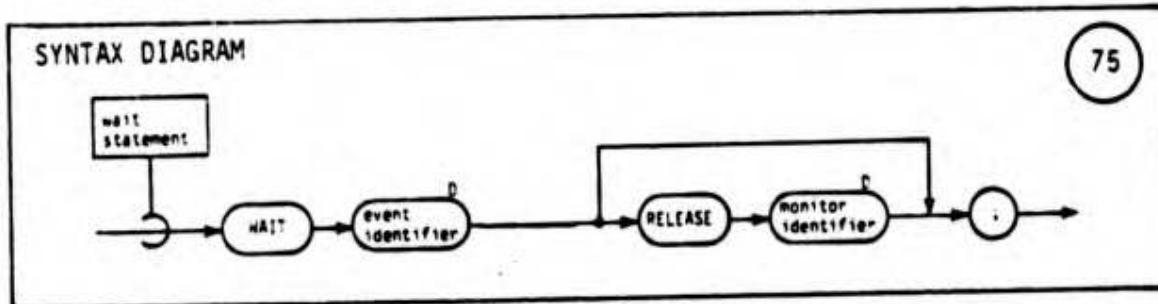


The effect of the <send statement> is to perform the following actions, in an indivisible manner:

- (1) If the queue of paths associated with the event is non-empty, choose a path based on the priority scheme described in Section 12.2.4 and change its state from waiting to running. Depending on priorities and the number of processors, this may preempt the path which executed the <send statement>.
- (2) If the queue of paths is empty, increment by one the integer associated with the event.
- (3) If no RELEASE clause is specified or if the path executing the <send statement> does not hold the named monitor, then execution of the <send statement> is complete.
- (4) If the RELEASE clause is specified, then the <send statement> must be lexically contained in the named monitor (and in no lexically smaller monitor). In this case, the sending

path releases the named monitor, which it held, immediately after completing steps (1) and (2) above and then immediately requests the monitor again. As a result it may happen that the sending path immediately regains hold of the monitor, but it may instead happen that another path requesting the monitor, perhaps the path awakened, will get hold of the monitor. Execution of the <send statement> is complete when the sending path again gets hold of the named monitor, at which point its execution proceeds.

12.4.3 Wait Statement.



The effect of the <wait statement> is to perform the following actions, in an indivisible manner:

- (1) If no RELEASE clause is given, then proceed to step (3).
- (2) If a RELEASE clause is given, then the <wait statement> must be lexically contained in the named monitor (and in no lexically smaller monitor). If the path executing the <wait statement> holds the named monitor, then it releases the monitor.
- (3) If the integer associated with the event is positive, it is decremented by one and execution continues in the scope which contains the <wait statement>.

(4) If the integer is zero, the path is placed on the queue associated with the event and its state is changed from running to waiting. While the path is in this state, its execution is suspended; it will remain in this state until another path executes a <send statement> to "awaken" it or raises the X_TERMINATE exception in it to terminate it.

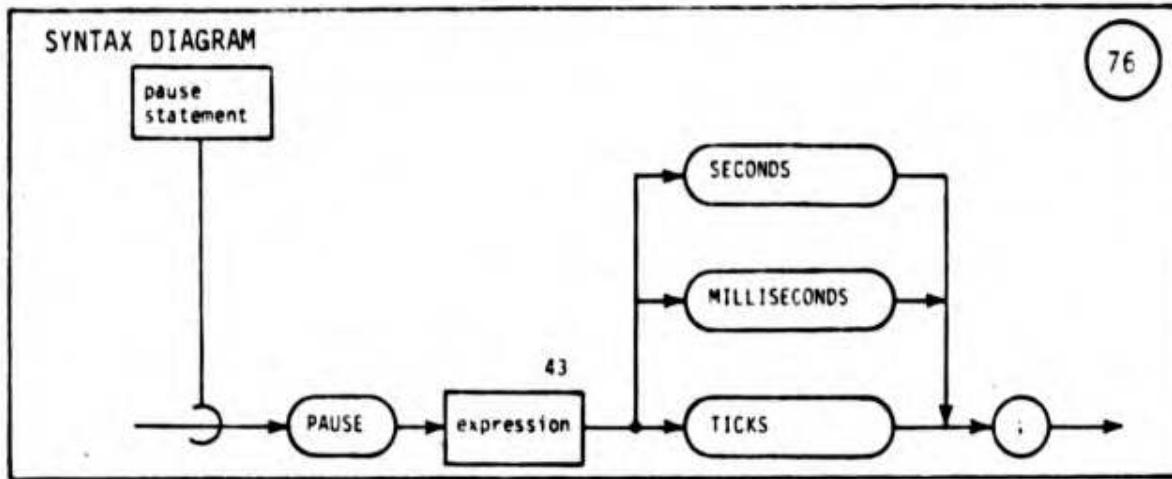
(5) Unless the waiting path released a monitor in step (2), the execution of the <wait statement> is complete. If a monitor was released in step (2), then it is now requested by the waiting path. Execution of the <wait statement> is complete when the path regains hold of the named monitor.

12.4.4 SENDERS and WAITERS.

There are two functions defined for events: if E is an event then SENDERS(E) is the non-negative integer associated with E, and WAITERS(E) is the length of the queue associated with E (i.e., the number of paths currently waiting for E).

12.5 Real-Time Facilities.

12.5.1 Pause Statement.



The <pause statement> is used to postpone further execution of the path until a specified time interval has elapsed. The <expression> in the <pause statement> must be integer valued; it represents the minimum number of units of time which must occur before the path may continue its execution. The units may be MILLISECONDS, SECONDS, or machine-dependent units called TICKS. The relationship between TICKS and SECONDS is given by the integer configuration constant TICKS_PER_SECOND.

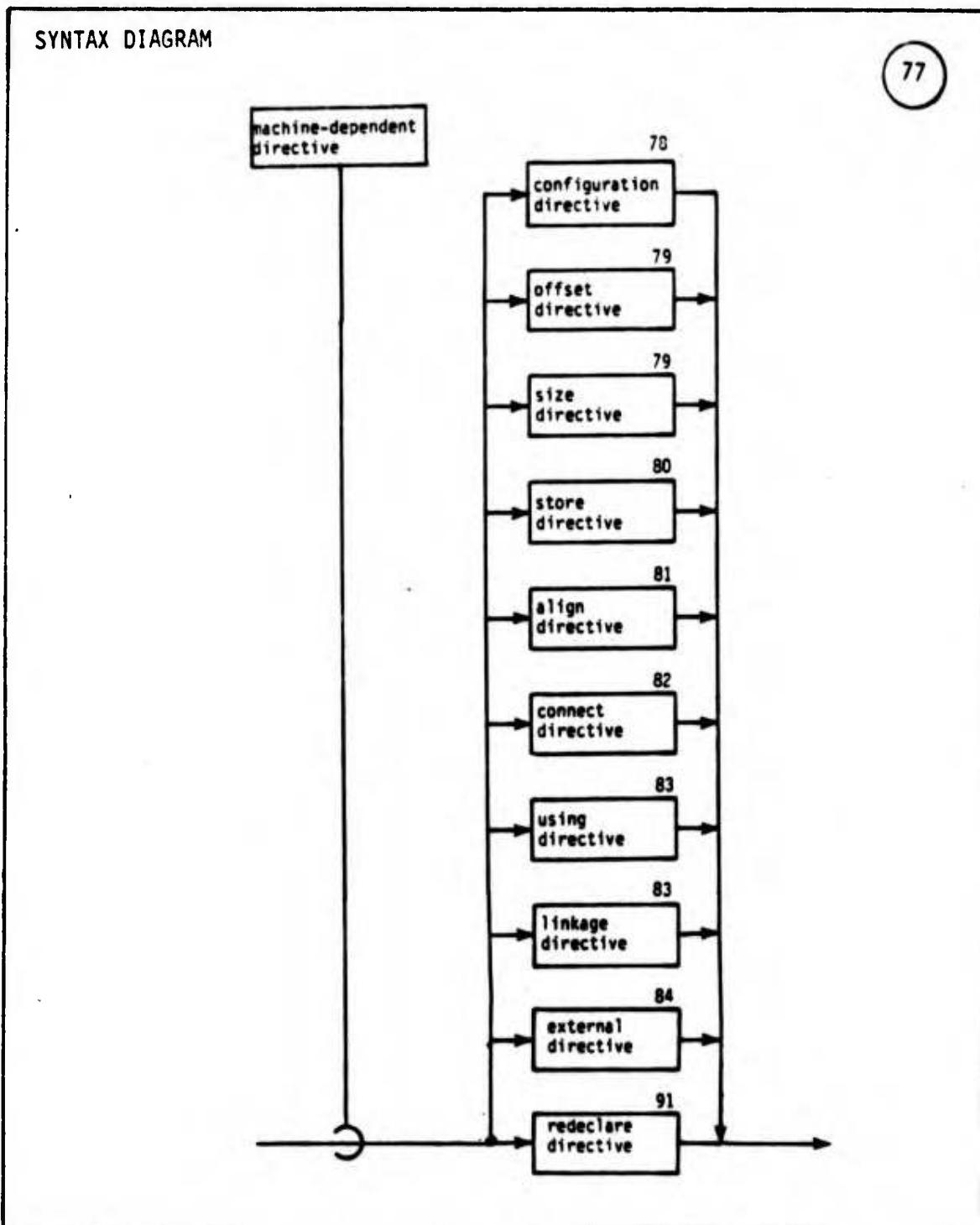
Execution of the <pause statement> changes the state of the path from running to waiting.

12.5.2 Cumulative Processing Time.

Each of the functions CUM_SECONDS, CUM_MILLISECONDS, and CUM_TICKS takes no arguments and returns an integer representing the total number of time units spent by the processor in executing the path from which the function was called.

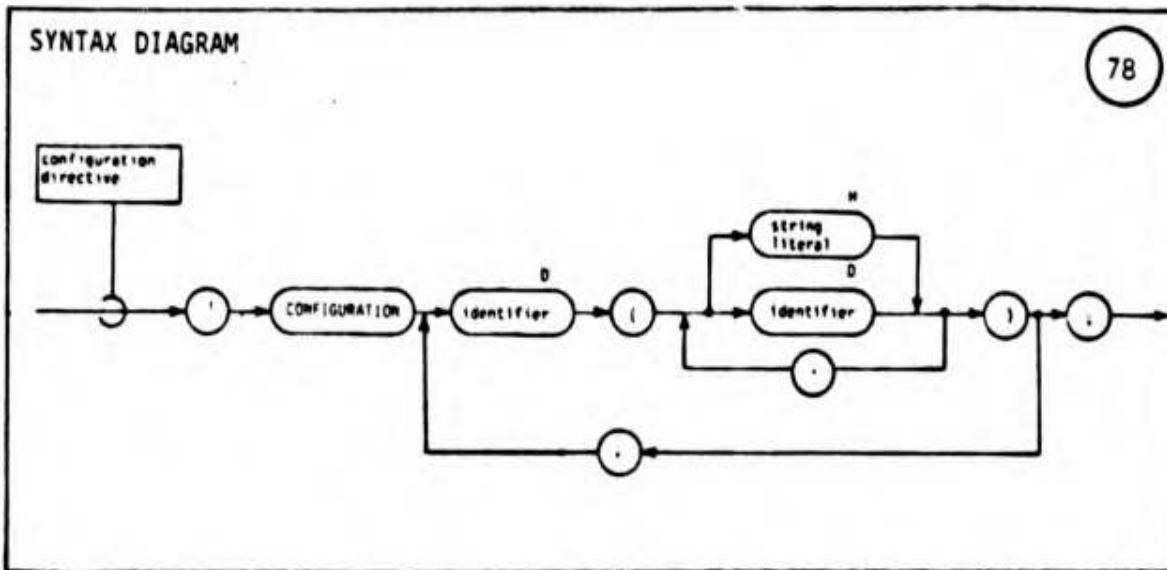
13.0 MACHINE-DEPENDENT FACILITIES

13.1 Introduction.



In response to the requirements of embedded applications to deal with hardware-specific properties, REDL supplies a set of machine-dependent features. In this chapter we categorize these features into two classes: data-oriented and routine-oriented machine dependencies. Both classes use forms which are based on <directive>s to announce the specific machine dependencies. the relevant <directive>s are given in Syntax Diagram 77.

13.2 Configuration Directive.



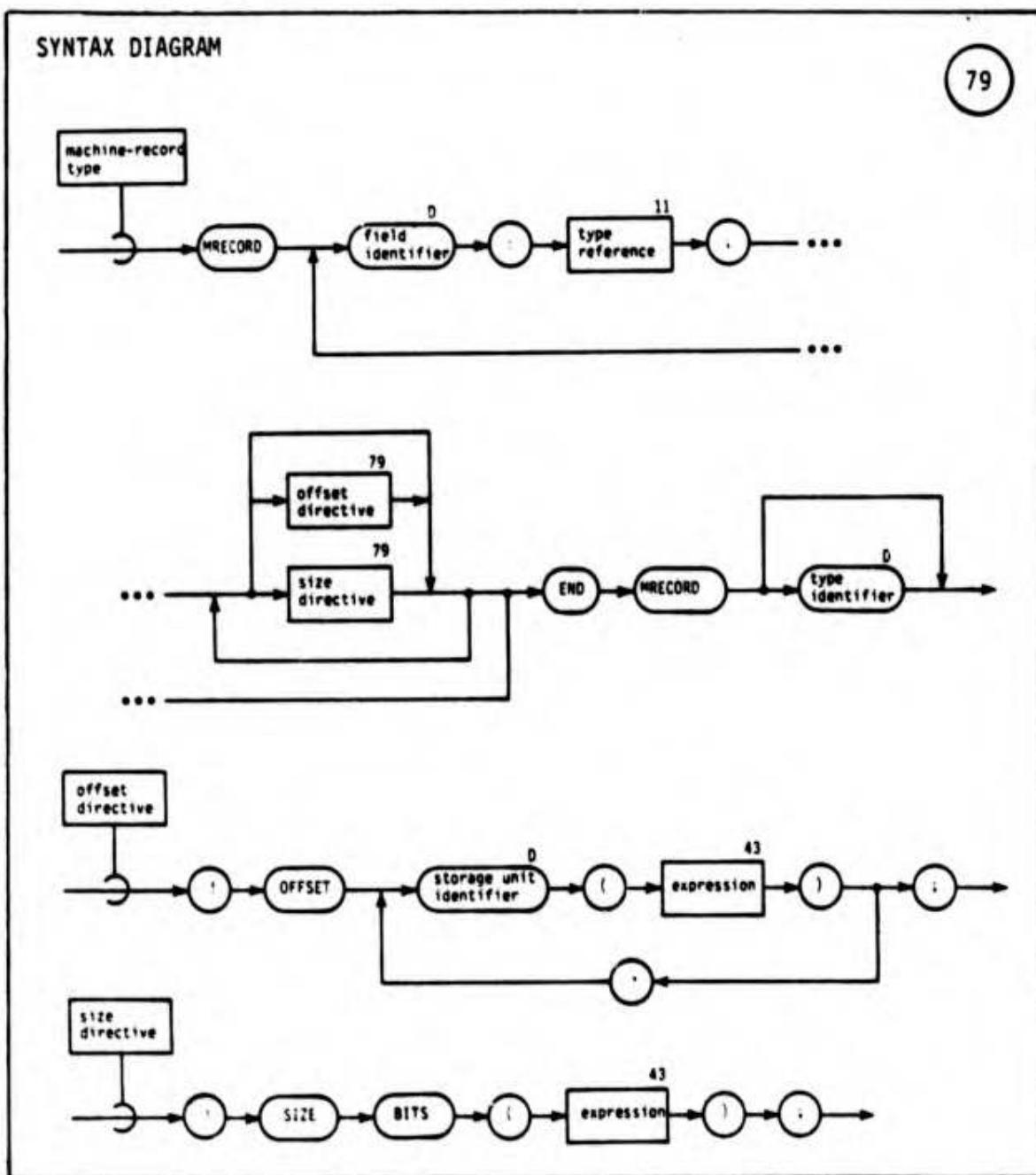
The <configuration directive> must appear among the <directive>s at program level for any <program> which uses any of the machine-dependent facilities. The list of identifiers and string literals itemizes such characteristics as the machine model, memory size, special hardware options, and/or operating system upon which the <program> is dependent.

Example:

```
! CONFIGURATION MACHINE('PDP10'), OP_SYS('TENEX'), CPU('KL10');
```

13.3 Data-Oriented Machine Dependencies.

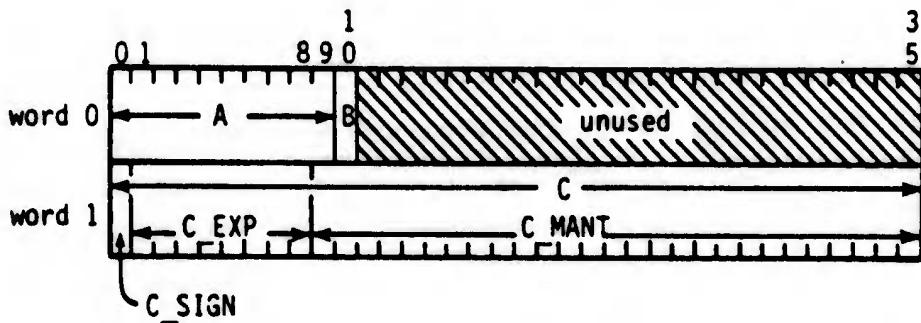
13.3.1 Machine Record.



Example:

```
TYPE T: MRECORD
  A: FIXED(0..1023);
    !OFFSET WORD(0), BIT(0);
    !SIZE BITS(10);
  B: BOOLEAN;
    !OFFSET WORD(0), BIT(10);
    !SIZE BITS(1);
  C: FLOAT(3, -1.0E5..1.0E5);
    !OFFSET WORD(1), BIT(0);
    !SIZE BITS(36);
  C_SIGN: BITS(1);
    !OFFSET WORD(1), BIT(0);
  C_EXP: BITS(8);
    !OFFSET WORD(1), BIT(1);
  C_MANT: BITS(27);
    !OFFSET WORD(1), BIT(9);
END MRECORD;
```

The storage layout for an object of type T is as follows, where the word-size is assumed to be 36 bits:



The machine-record type generator, a generalization of the record type generator, is used for the declaration of objects whose storage layout is to be specified by the programmer. This specification is made via one or two <directive>s following the declaration of a field identifier. The <offset directive> determines the start of the component's storage in terms of implementation-defined storage unit identifiers such as DOUBLEWORD, WORD, HALFWORD, BYTE, BIT. The first DOUBLEWORD or WORD is defined to be at offset 0, but within a WORD the numbering scheme for HALFWORDS, BYTES and BITS is implementation-defined (i.e., whether offsets start at 0 or 1 and whether they ascend left-to-right or right-to-left).

The only attribute of a machine-record type is the grouping attribute.

Each component for which an <offset directive> is given is defined to be grouped.

Machine-records offer a mechanism for circumventing the language's strong type checking, since components may be overlaid and thus stored according to one type and retrieved according to another. An illustration appears in the declaration of T above; the C component is declared as a FLOAT but is overlaid with three BITS components which correspond to the sign, exponent, and mantissa of C.

The size <directive> is used for specifying the size (in bits) of a component. If the size is not that which the compiler would determine for the component, then the effect is implementation-dependent and the compiler will issue an appropriate warning.

In the absence of the <size directive> or <offset directive> for a component, the compiler will determine the actual size and offset.

The <expression>s in the size and offset <directive>s must be compile-time evaluable and yield integers.

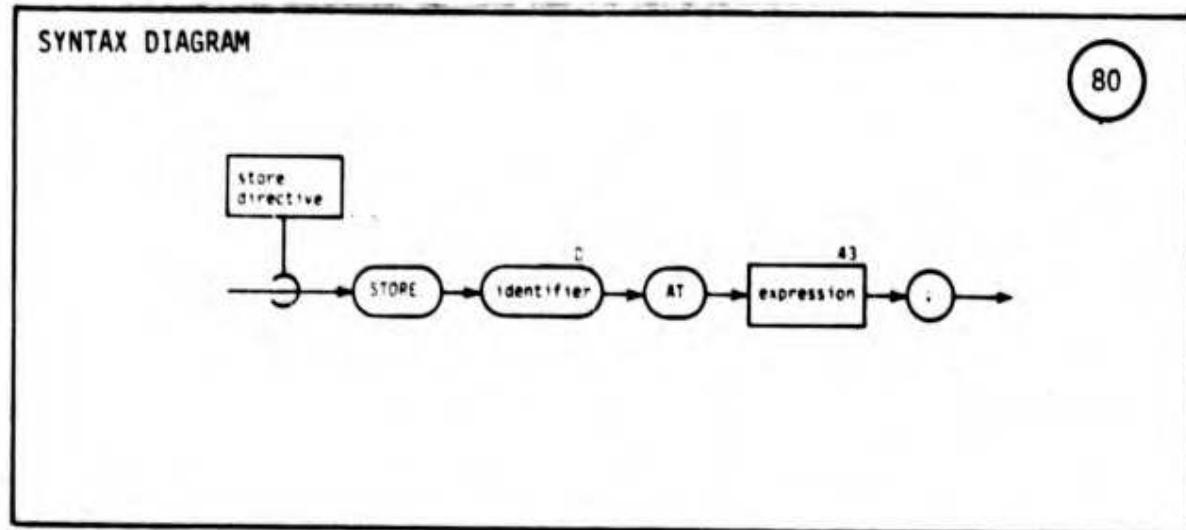
The forms for machine-record component selection and machine-record construction are the same as for records.

13.3.2 Absolute Addresses and Alignment.

13.3.2.1 The ADDRESS Type. The built-in machine-dependent type ADDRESS is used in the <store directive> to specify an absolute storage location, and is the result type of the LOC function. The <type reference> for ADDRESS is simply the identifier ADDRESS itself. There are no literals of type ADDRESS. The routines defined for the type are the = and # operations and an explicit conversion function (the type identifier ADDRESS itself) from integers and STRINGS.

13.3.2.2 The LOC Function. The LOC function takes as its parameter an arbitrary data object which is ungrouped, or the identifier of a routine which is not overloaded and not expanded inline. The function returns a value of type ADDRESS; viz., the absolute storage address at which the object or routine is located. LOC is link-time evaluable if its argument is either a data object declared at program level or a routine.

13.3.2.3 The Store Directive.

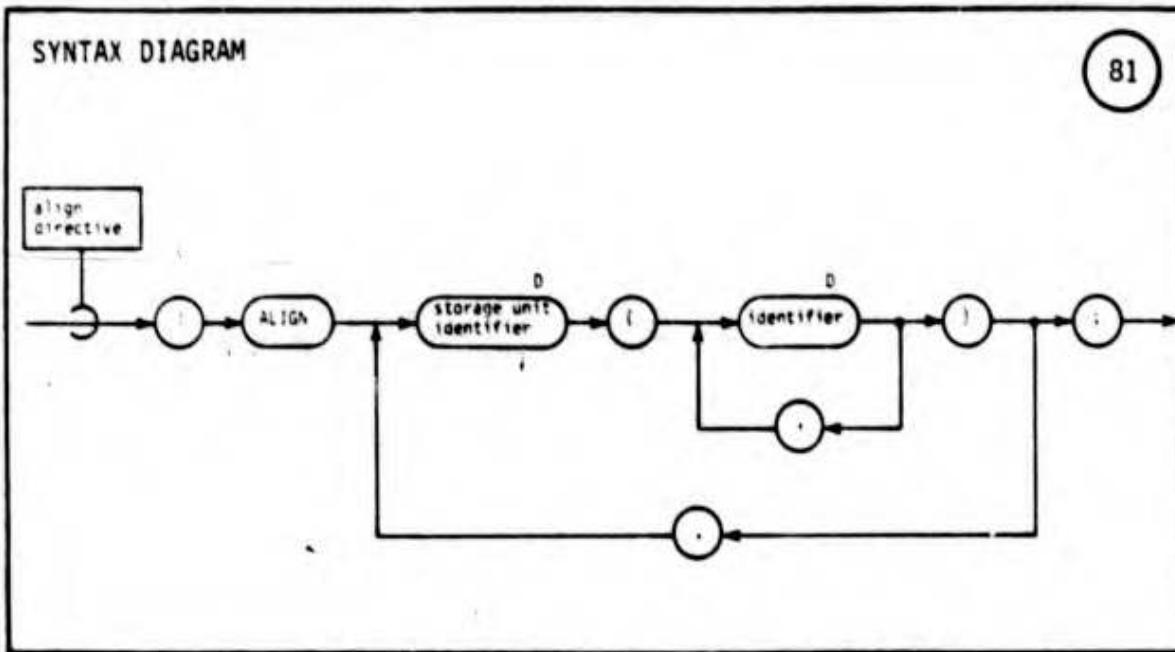


The <store directive> is used for specifying absolute storage addresses for data objects and/or routines. The identifier must denote either a data element declared at program level (in which case the <directive> must likewise appear at program level), or a routine which immediately contains the <directive>. The <expression> must have type ADDRESS.

Example:

```
:STORE X AT ADDRESS.'0177';
<*Now LOC(X) = ADDRESS('0177')*>
```

13.3.2.4 The Align Directive.



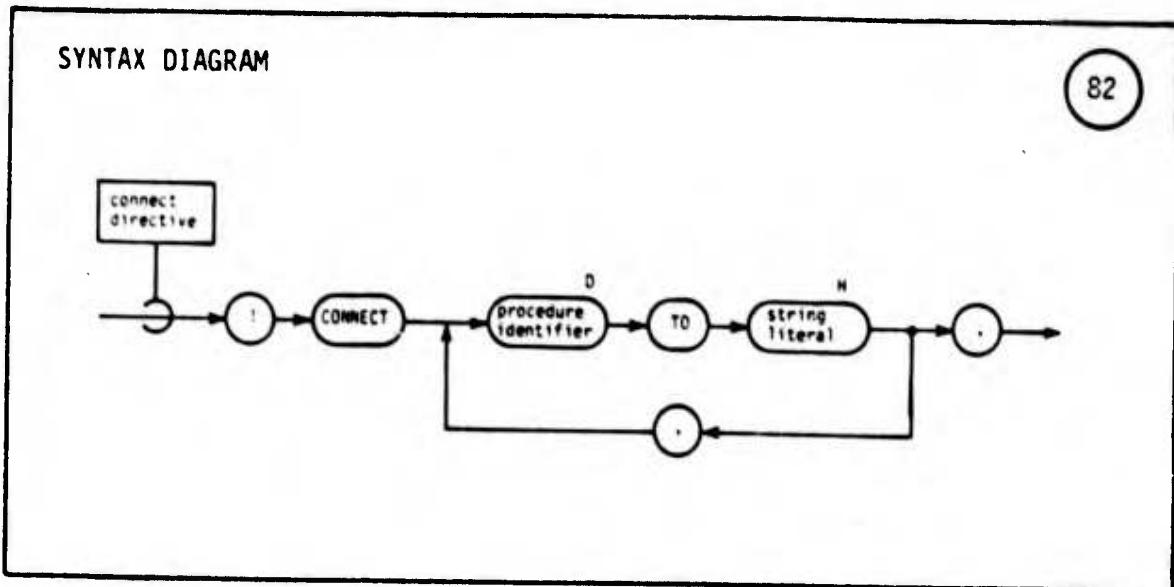
The <align directive> is used for specifying alignment of data elements on boundaries which are a multiple of a programmer-specified storage unit. The identifiers in the parenthesized list may denote either constants or variables or types. If a constant or a variable, then alignment will be as specified by the storage unit. If a type, then each constant or variable of that type will be so aligned.

The <align directive> must be immediately contained in the scope in which the declarations of the identifiers referenced in the <directive> appear.

Example:

```
:ALIGN DOUBLEWORD(X,Y,Z,T), WORD(A,B);
```

13.3.3 The Connect Directive.



The <connect directive> is used to establish an association between a programmer-designated procedure and an asynchronous hardware interrupt. This association is defined so that the occurrence of the interrupt causes the invocation of the procedure as the "handler" for the interrupt. The string literal specifies the interrupt in an implementation-defined manner.

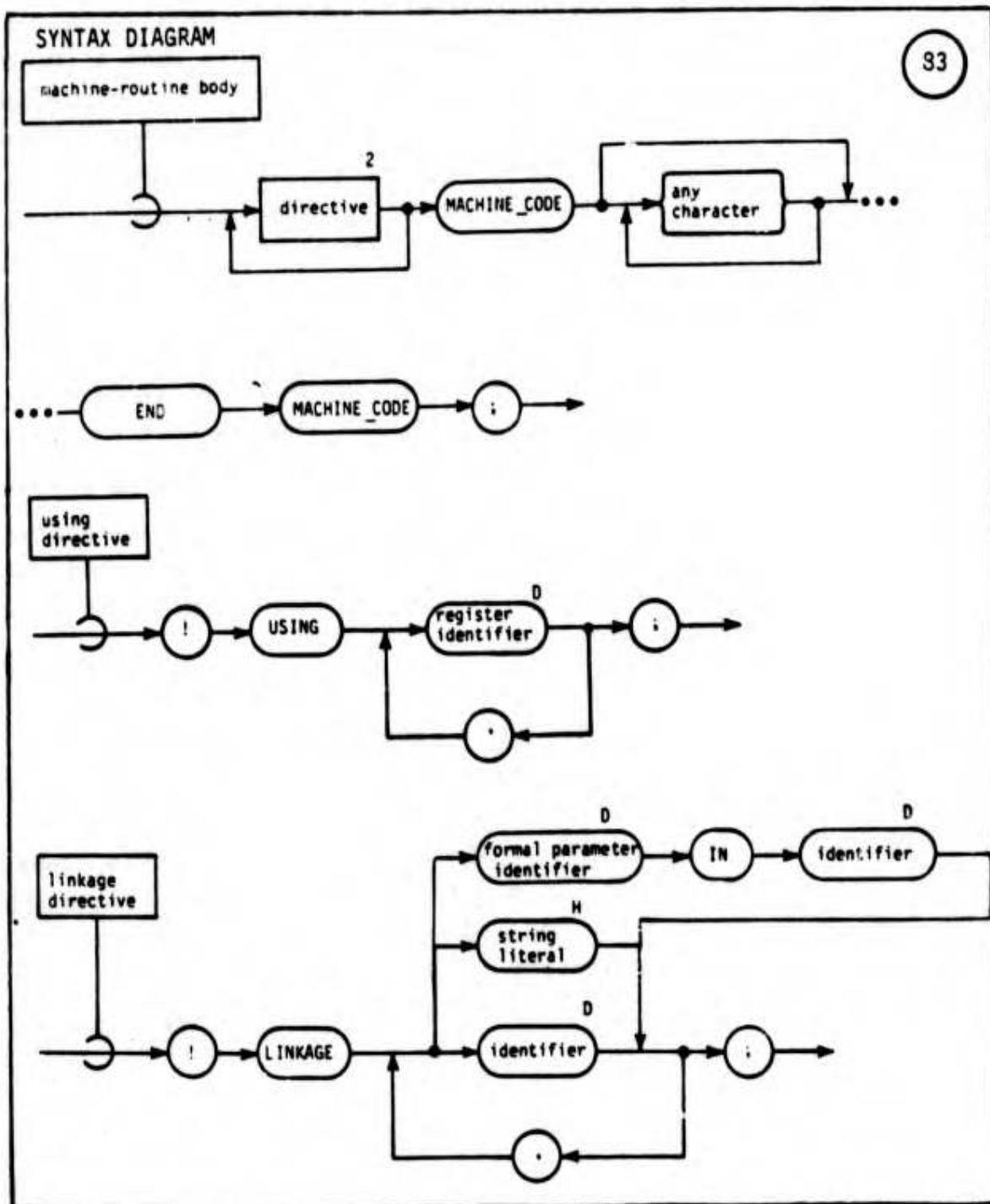
We note that the use of the <connect directive> to establish a handler for asynchronous interrupts is analogous to the use of the <on statement> to establish a handler for those pre-defined exceptions corresponding to synchronous interrupts. A difference is that asynchronous interrupts do not necessarily imply program errors, whereas the pre-defined exceptions do.

Upon entry to a scope immediately containing a <connect directive>, the handler(s) (if any) formerly associated with the interrupt(s) named in that <directive> are saved ("pushed down"); upon exit from the scope, all the saved handlers are restored for their associated interrupts.

Any procedure named as an interrupt handler in a <connect directive> must be declared at program level with LINKAGE 'INTERRUPT' and may not take parameters.

13.4 Routine-Oriented Machine Dependencies.

13.4.1 Machine Routines.



If it is necessary for a user to write portions of a <program> in assembly or machine language, this may be accomplished via REDL's machine-routine facility. The header of a machine-routine has the same form as a routine header, and type-checking is carried out in the normal way on calls. The difference is that a <machine-routine body> contains <directive>s establishing the linkage and register usage conventions, and a sequence of characters bracketed by MACHINE_CODE and END MACHINE_CODE. These characters are interpreted in an implementation-defined manner as constituting the assembly or machine language statements. The only constraint on the sequence of characters is that the first occurrence of the consecutive tokens END MACHINE_CODE will serve as terminator.

The only <directive>s permitted in the declaration of a machine-routine are the inline, using, and linkage <directive>s.

The <using directive> identifies all registers whose contents are modified by the machine-routine. This <directive> is useful as an optimization, since there is no need for the implementation to save any registers at the point of call other than those which the <using directive> identifies.

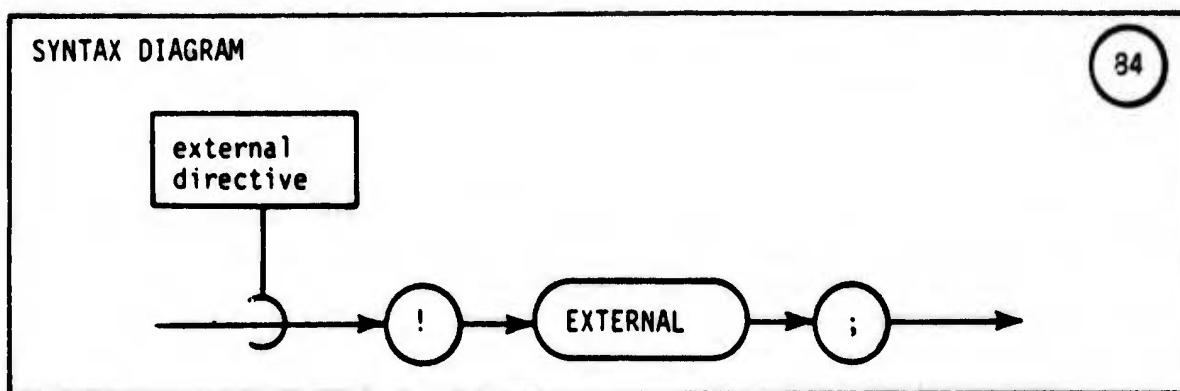
The <linkage directive> establishes the conventions to be used regarding parameter passing and return address saving. The user may specify explicit registers for such information; alternatively, identifiers or string-literals reflecting implementation-defined linkages (such as passing parameters on a run-time stack) may be used. We note that, if parameters are passed in registers, then such registers will contain addresses for VAR and RESULT parameters, and either addresses or values for CONST parameters (depending on the individual machine-routine and the size of the object).

If a machine-routine is specified with the <inline directive>, then formal parameter identifiers may be referenced within the assembly language statements. These identifiers will be replaced at the point of call by the appropriate addresses, registers and/or literals.

Example:

```
!CONFIGURATION MACHINE('AYK-14);
PROCEDURE ROTATE_LEFT(VAR V: BITS(16), CONST N: FIXED(0..15));
  !INLINE;
  !LINKAGE V IN R1, N IN IMMEDIATE;
  MACHINE_CODE
  LCLS V, N
  END MACHINE_CODE;
END PROCEDURE ROTATE_LEFT;
VAR Y: BITS(16) INIT X'FF00';
  :
CALL ROTATE_LEFT(Y,1);
<*Now Y is X'FE01'*>
```

13.4.2 External Routines.



An external-routine is a procedure or function written in a language other than REDL and using parameter linking conventions different from those of REDL. An external-routine declaration provides an interface between a REDL <program> and an external-routine in the implementation environment. An external-routine declaration specifies the name (and possibly the directory location) of an external-routine and the linkage conventions necessary for invocation.

A <routine declaration> is designated as an external-routine declaration by the appearance of the <external directive>.

Each external-routine declaration must include a <linkage directive> and may include a <using directive>. No other <directive>s may appear.

An external-routine declaration does not include a <body>, since this is specified elsewhere (in the non-REDL definition of the routine). External-routines are invoked in the same way as REDL routines. Type checking is performed at the point of invocation as far as the <type reference>s in the external-routine declaration are concerned, but there is no guarantee that these types correspond to those required by the actual non-REDL routines. Similarly, the effect of binding class specifications not supported by the non-REDL language is implementation-dependent.

The identifier appearing in the header of the declaration is the identifier of the routine as it will be known inside the REDL program. An association between the declaration and an externally-supplied definition of a routine is established by the <linkage directive>. The identifiers for the formal parameters and/or result variables appearing in the routine header may be employed in this <directive> to specify parameter passage via registers. Alternatively, the parameter passage conventions may be designated

by other items in the <linkage directive>, such as the language name. The interpretation of other identifiers and string literals comprising the list appearing in this <directive> is implementation-dependent, but information contained here includes the name of the language in which the external-routine is written (e.g., FORTRAN) and an identification of the location of the routine for purposes of linking (e.g., '<FTNLIB>SINE.REL'). The <using directive> specifies register usage as explained in Section 13.4.1.

Example:

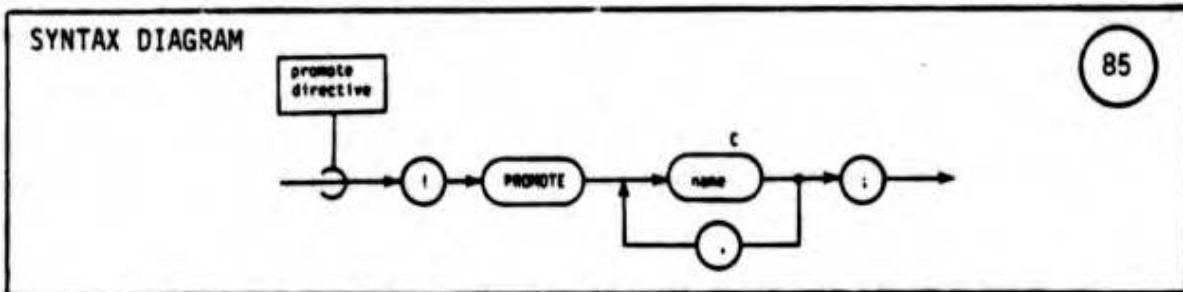
```
FUNCTION SIN(ANGLE: FLOAT(8, -1.0E3..1.0E3))
    RESULT SIN_ANGLE: FLOAT(8, -1.0..1.0);
!LINKAGE FORTRAN, '<FTNLIB>SINE.REL';
!USING R0,R1,R2,R14,R15;
END FUNCTION SIN;
```

14.0 COMPILE-TIME FACILITIES

14.1 Separate Compilation.

REDL allows <program>s to be constructed and compiled separately. The features provided for this purpose--the <promote directive> and the <access directive>--permit decomposition of large systems into manageable-sized <program>s without sacrifice of compile-time checking. As special cases of such decomposition, REDL allows the writing of <program>s containing only type and data declarations (similar to "com pools"), the writing of <program>s containing declarations of routines ("libraries"), and the writing of <program>s consisting solely of <capsule declaration>s (for data encapsulation and/or data base management).

14.1.1 Promote Directive.



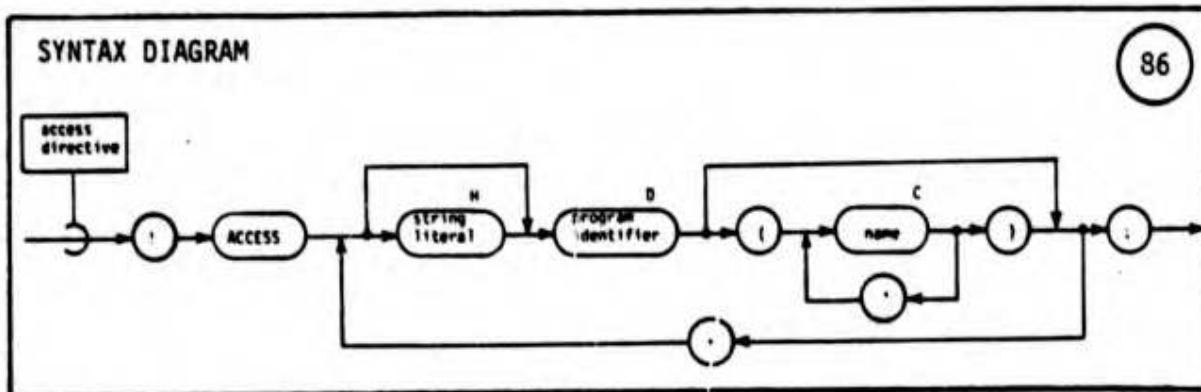
The <promote directive> makes a name in one <program> accessible to other <program>s, separately compiled. The names in the <promote directive> must denote elements declared at program level in the <program> containing the <directive>. The <promote directive> may only appear at the program level.

A name listed in a <promote directive> is said to be promoted.

If a capsule identifier is promoted, then any names exported from the capsule are automatically promoted.

If an overloaded routine name is exported, then each program-level declaration for the routine name is promoted. Thus the user may overload a built-in operator and then promote the operator name.

14.1.2 Access Directive.



The <access directive> makes it possible for the <program> containing the <directive> to refer to elements promoted by another <program>. The <access directive> may only appear at the program level.

A name appearing in a parenthesized list in the <directive> is said to be accessed.

The string literal, if it appears, has an implementation-defined interpretation which specifies the host machine file on which the accessed <program> is stored. If the string literal is omitted, an implementation-defined default will be used.

If no parenthesized list appears after the program identifier, then all promoted names from the designated <program> are accessed.

The effect of an <access directive> is to make the declaration of each accessed name (as established in the <program> from which the name was promoted) known in the program-level scope in which the <access directive> appears.

We point out that an accessed name may in turn be promoted from the accessing <program>. The name also may be made pervasive, or, if a variable, may be made readonly, through use of the corresponding <directive>s.

14.1.3 Linking Separately-Compiled Programs.

A problem associated with separate compilation is that of incompatible <program> updates, illustrated by the following situation. <Program> A declares and promotes routine R which takes a parameter of type FIXED. <Program> A is compiled. <Program> B accesses R, and some routine in B invokes R. <Program> B is compiled, with type checking performed at the invocation of R to ensure that the actual parameter has type FIXED. Now the source <program> A is changed, so that R takes a parameter of type FLOAT, and A is recompiled. If the linker were to permit A and B to be linked, the language's type checking would be defeated. On the other hand, if R were changed in some other manner (e.g., inserting some additional <statement>s) there would be no reason to prevent A and B from being linked, since such a change would not affect the compiler-checkable interface between the <program>s.

The general problem is as follows: given a set of inter-dependent, separately-compiled <program>s (we refer to compiled <program>s as object modules), when is linking permissible? The language rule for this situation can be stated straightforwardly: linking is permissible provided that a recompilation of the set of source <program>s which correspond to the object modules would not cause any errors.

(For reasons of link-time and object-module storage efficiency, implementations are unlikely to enforce the above rule exactly as it was stated. Instead, equivalent but more

efficient techniques are anticipated. One such approach is for the compiler to generate a "template" for each promoted program element as part of the object module for the promoting <program>, and to copy this template into the object module of the accessing <program>. The template specifies those aspects of the element which, if changed, will force recompilation of <program>s which accessed the promoted program element before the change. Thus the template for a <routine declaration> will include the types of the formal parameters but not the <body> of the routine (unless the routine is to be expanded inline). The template for a <type declaration> will be the declaration in its entirety. When two object modules are linked, the templates for corresponding program elements must be the same in both the accessing and the accessed module; otherwise a link-time error will result, informing the user of the template incompatibility.)

14.1.4 Example.

In this section we illustrate the use of the promote and access <directive>s in the definition of <program>s which contain mutually recursive routines.

(1) Write (and compile) <program> A, with a "stub" for the eventually recursive procedure PA:

```
PROGRAM A;  
    !PROMOTE PA;  
    PROCEDURE PA(I:FIXED(0..100),RESULT R:FIXED(0..1E5));  
    END PROCEDURE PA;  
END PROGRAM A;
```

(2) Write (and compile) another <program> containing the procedure PB which calls PA:

```
PROGRAM B;  
    !PROMOTE PB;  
    !ACCESS A(PA);  
    PROCEDURE PB(J:FIXED(0..100);RESULT R:FIXED(0..1E5));  
        !IMPORT PA;      !RECURSIVE;  
        ...CALL PA(J-1,R);...  
    END PROCEDURE PB;  
END PROGRAM B;
```

(3) Add the <body> of PA and recompile A, thereby introducing the mutual recursion:

```
PROGRAM A;  
    !PROMOTE PA;  
    !ACCESS B(PB);  
    PROCEDURE PA(I:FIXED(0..100),RESULT R:FIXED(0..1E5));  
        !IMPORT PB;      !RECURSIVE;  
        ...CALL PB(I-1,R);...  
    END PROCEDURE PA;  
END PROGRAM A;
```

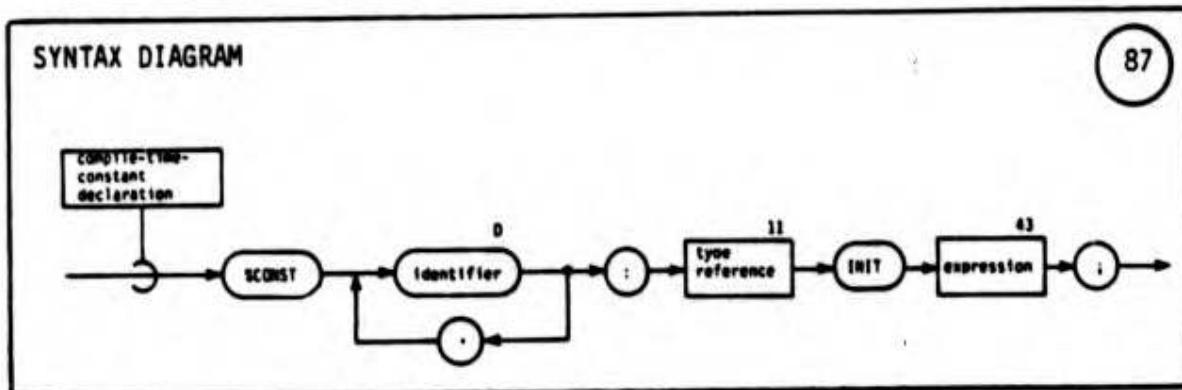
(4) Link <program>s A and B to generate a new object module.

We observe from this example that a modification to a <program> (A in the example above) does not necessarily imply a link-time error when the object module for the revised version is linked with an object module of another <program> (B) which accessed elements from the previous version.

14.2 Compile-Time Constants and Expressions.

REDL provides both a facility for user declaration of compile-time constants and a set of language-defined compile-time constants which are accessible from a standard library. These constants may be used in <expression>s whose compile-time evaluability is guaranteed.

14.2.1 Compile-Time Constant Declarations.



A <compile-time constant declaration> has the effect of declaring one or more compile-time constants, each having as its value the result of the <expression> in the INIT phrase. A compile-time constant differs from other constants in the following respects:

(1) Its type must be one of the built-in types (see Figure 5-1).

(2) Its initialization <expression> must be compile-time evaluable.

(3) It may be used as an operand in a compile-time evaluable <expression>.

Example:

```
%CONST      PI:FLOAT(6)INIT 3.14159;
%CONST    TWO_PI:FLOAT(5)INIT 2*PI;
```

14.2.2 Language-Defined Compile-Time Constants.

REDL provides a variety of compile-time constants which reflect configuration-dependent information. Table 14-1 gives the name and interpretation of each such constant. To be used in a <program>, the constants must be accessed from a standard library to be provided by the implementation. Each constant is an integer.

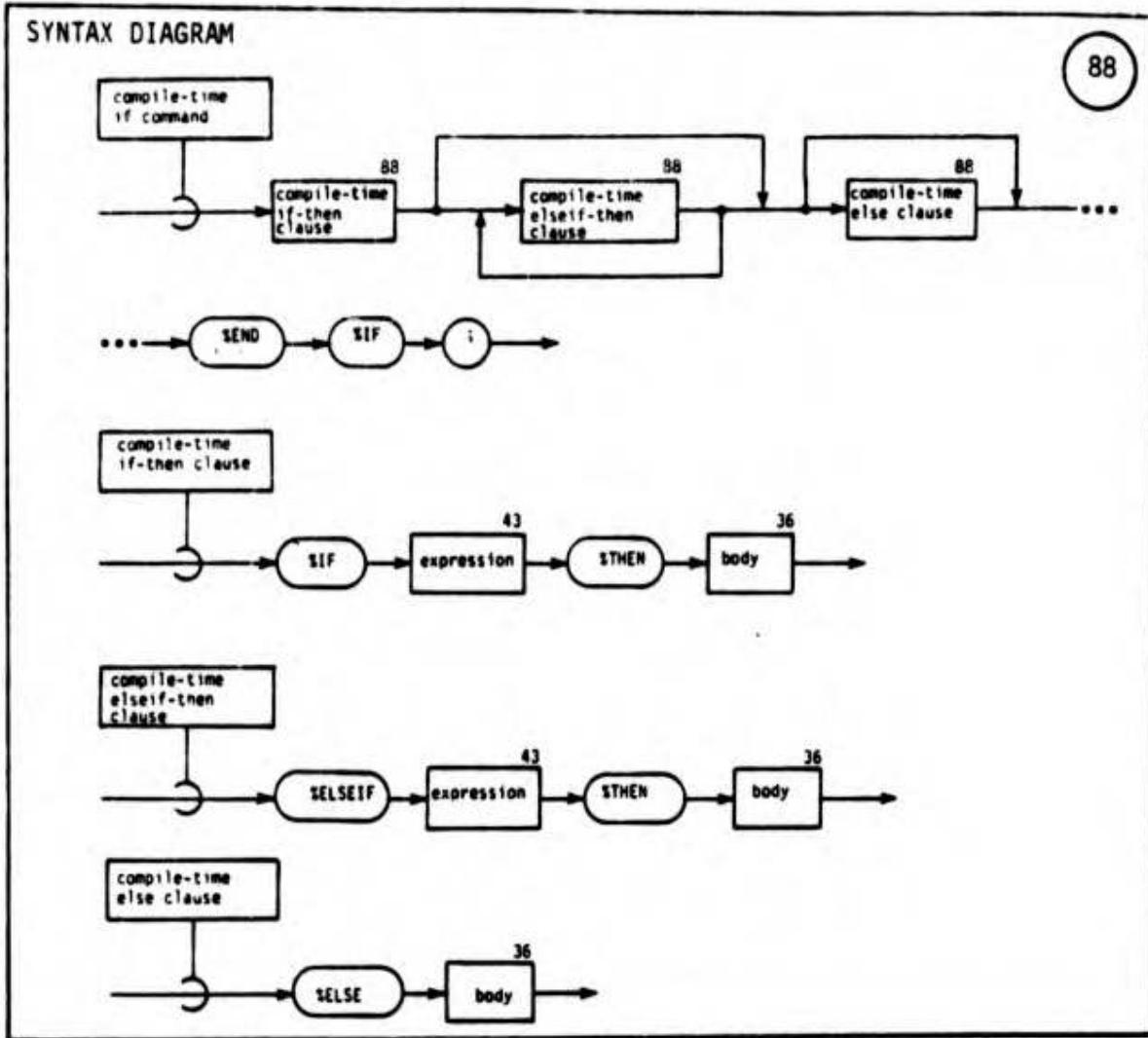
Name	Interpretation
MIN_INTEGER	Smallest integer representable
MAX_INTEGER	Largest integer representable
MAX_PRECISION	Largest floating-point precision
RADIX	Floating-point radix
MIN_EXPONENT	Smallest floating-point exponent
MAX_EXPONENT	Largest floating-point exponent
MIN_PRIORITY	Lowest path priority
MAX_SENDERS	Maximum number of senders of an event

Table 14-1. Configuration Constants in REDL

14.2.3 Compile-Time Expressions.

A compile-time <expression> is an <expression> whose value is determinable at compile-time. Operators appearing in the <expression> must be built-in (i.e., an operator overloaded for user-defined types is not permitted). Operands must either be literals, compile-time constants, or invocations of built-in functions. If the invocation of any of these functions does not result in a compile-time determinable value, and the context of the invocation requires such a value, then a compile-time error occurs.

14.3 Conditional Compilation.



The <compile-time if command> provides a conditional compilation facility. Each of the <expression>s must be of BOOLEAN type and be compile-time evaluable. A <compile-time if command> is processed by replacing the entire command by a selected <body> of the command. The <body> is selected as follows:

(1) Evaluate the <expression> in the <compile-time if-then clause> and the <expression>s in each <compile-time elseif-then clause> until the first occurrence of a TRUE result. The selected <body> is the <body> of this clause.

(2) If all <expression>s evaluate to FALSE, then select the <body> in the <compile-time else clause> (if this clause is present).

(3) If all <expression>s evaluate to FALSE and no <compile-time else clause> is present, no <body> is selected and the entire <compile-time if command> is deleted.

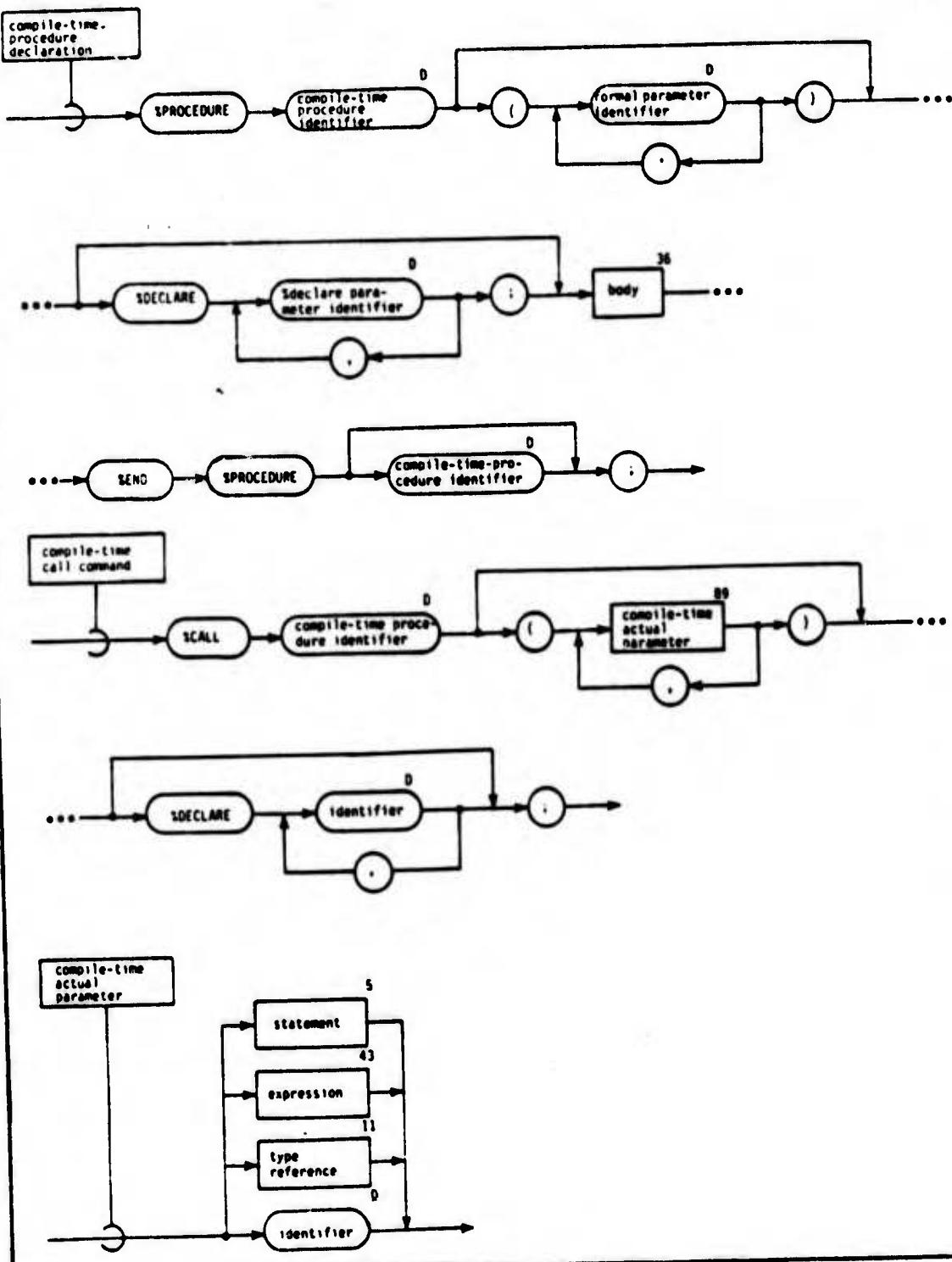
An example of the compile-time facilities described thus far appears below. In this example, assume that P, Q, and R are program elements existing in two forms: one version for debugging purposes, in <program> LIBDEB, and another version for production purposes, in <program> LIBPRO. The <program> being defined consists of a main routine which uses the version indicated by the value of the compile-time constant DEBUGGING.

```
PROGRAM SAMPLE_PROG;
  !PROMOTE MAIN_PROC;
  %CONST DEBUGGING: BOOLEAN INIT TRUE;
  %IF DEBUGGING
  %THEN !ACCESS LIBDEB(P,Q,R);
  %ELSE !ACCESS LIBPRO(P,Q,R);
  %END %IF;
  PROCEDURE MAIN_PROC;
    !MAIN;
    ...P...Q...R
  END PROCEDURE MAIN_PROC;
END PROGRAM SAMPLE_PROG;
```

14.4 Compile-Time Procedures.

SYNTAX DIAGRAM

89



Compile-time procedures provide a semantic macro facility, in accordance with Ironman 12D. (As a notational convenience, we will frequently find it useful to abbreviate "compile-time" by the acronym "CT" in the remainder of this section.) The declaration of a CT procedure includes a set of formal parameters (called generic parameters in Ironman 12D) and a <body> which references these parameters. Just as a procedure may only be invoked via a <call statement>, so a CT procedure may only be invoked via a <compile-time call command>. The behavior of the two, however, is different: the <call statement> is executed at run-time to yield run-time effects, whereas the <compile-time call command> is processed at compile time and is, in effect, replaced by a copy of the <body> of the CT procedure, with parameter substitution performed as described below. The resulting copy is said to be produced by the processing of the <compile-time call command>.

The following example is a CT procedure which, when invoked in a <compile-time call command>, produces a declaration of a (run-time) procedure. (Thus the <body> of the CT procedure is simply a <procedure declaration>.)

```
%PROCEDURE P(T,F) %DECLARE PROC ID;
    PROCEDURE PROC_ID(CONST C:T, RESULT R:T INIT ?);
        R:=F(C);
    END PROCEDURE PROC_ID;
%END %PROCEDURE P;
```

When P is CT-invoked with a <type reference>, a function identifier, and an undeclared identifier as the CT actual parameters, it will produce a <procedure declaration> of the identifier; this new procedure will take two parameters, each having the type supplied

in the CT-invocation's first CT actual parameter. For example, the <compile-time call command>

```
%CALL P(FLOAT(4,-PI..PI),SIN) %DECLARE P1;
```

produces the following:

```
PROCEDURE P1(CONST C:FLOAT(4,-PI..PI),
             RESULT R:FLOAT(4,-PI..PI));
    R:=SIN(C);
END PROCEDURE P1;
```

As another example, the following <program> fragment shows a protocol for achieving mutually exclusive access to shared data, without the use of monitors:

```
FORK
EVENT MUTEX INIT 1;
%PROCEDURE EXCLUSIVE(S);
    !IMPORT MUTEX;
    WAIT MUTEX;
    S;
    SEND MUTEX;
%END %PROCEDURE EXCLUSIVE;
VAR X:FIXED(-1000..1000) INIT ?;
PATH P1; !IMPORT X, EXCLUSIVE; ...%CALL EXCLUSIVE(X:=X+1);...
END PATH P1;
PATH P2; !IMPORT X, EXCLUSIVE; ...%CALL EXCLUSIVE(X:=X-1);...
END PATH P2;
END FORK;
```

Without a mechanism to produce unique names, the processing of a <compile-time call command> which produces a new declaration would produce multiple declarations for the same name when multiple CT calls occurred in the same scope. To prevent this, REDL provides a %declare parameter mechanism. %Declare parameters (the identifiers

following %DECLARE) permit the specification of unique names for each produced declaration at each <compile-time call command>. In the first example above, PROC_ID is the formal %declare parameter in P, and the identifier Pl was passed as the actual substitution parameter in the CT call of P.

A CT procedure is somewhat like a macro facility in that it has a name, takes parameters, and is expanded when called so that the actual parameters replace the formals. There is, however, an important difference. The replacement is not that of character text, but rather that of semantic units; the environment of the <compile-time procedure declaration> is used to resolve name references in the <body> of the CT procedure, and the environment of the <compile-time call command> is used to resolve name references in the actual parameters to the CT call. The following example illustrates these environment rules.

```
VAR X: FIXED(-10..10) INIT-3;
%PROCEDURE H(F,V) %DECLARE PROC_ID;
  !IMPORT X;
  PROCEDURE PROC_ID(RESULT R: FIXED(-10..10) INIT ?);
    !IMPORT X,F,V;
    R:=F(X)+V;
  END PROCEDURE PROC_ID;
%END %PROCEDURE H;

PROCEDURE P;
  !IMPORT H;
  %CONST X: FIXED INIT 5;
  %CALL H(ABS,X) %DECLARE Q;
  VAR Y: FIXED(-10..10) INIT?
  :
  CALL Q(Y);
  <*This has the effect of assigning ABS(-3)+5, i.e., 8, to Y*>
  :
END PROCEDURE P;
```

When Q is declared in the CT call of H, ABS replaces F in the <body> of H, and the value of X in the environment of the CT call (viz., 5) replaces V. The reference to X in the <body> of H (in F(X)) is a reference to the imported X in the environment of H's declaration.

The <compile-time actual parameter>s supplied in a <compile-time call command> may be any <statement>, <expression>, <type reference>, or any declared identifier. The actual parameters supplied for the formal &declare parameters must be identifiers which are not declared, since the CT call will have the effect of establishing declarations for them in the scope of the <compile-time call command>.

The <body> of a <compile-time procedure declaration> is a closed scope with respect to the environment of the declaration; thus any non-pervasive names from this environment must be imported.

<Compile-time if command>s may appear in the <body> of a CT procedure. In this case the <compile-time if command>s are not processed at the point where the <compile-time procedure declaration> appears. These commands are retained in the copy of the <body> of the CT procedure that is produced as a result of processing a <compile-time call command> that references the CT procedure. This permits the <expression>s in <compile-time if command>s to depend on <compile-time actual parameter>s.

The following example shows the use of compile-time procedures to define a "stack generator" capsule.

```

$PROCEDURE STACK_GENERATOR(N,T) $DECLARE CAPSULE_ID, TYPE_ID;
  CAPSULE CAPSULE_ID;
  !IMPORT N,T;
  !EXPORT TYPE_ID, STACK_INIT, PUSH, POP;
  !PERVASIVE TYPE_ID,N,T;
  TYPE T_ARRAY: ARRAY(1..N) OF T;
  TYPE TYPE_ID: RECORD
    VAL: T_ARRAY;
    TOP: FIXED(0..N);
  END RECORD;
  OVERLOAD PROCEDURE STACK_INIT(VAR S: TYPE_ID);
    S.TOP:=0;
  END PROCEDURE STACK_INIT;
  OVERLOAD PROCEDURE PUSH(VAR S: TYPE_ID, CONST V: T);
    IF S.TOP=N THEN RAISE X_OVERFLOW;
    ELSE S.TOP:=S.TOP+1;
      S.VAL(S.TOP):=V;
    END IF;
  END PROCEDURE PUSH;
  OVERLOAD PROCEDURE POP(VAR S: TYPE_ID, RESULT V: T INIT ?);
    IF S.TOP=0 THEN RAISE X_UNDERFLOW;
    ELSE V:=S.VAL(S.TOP);
      S.TOP:=S.TOP-1;
    END IF;
  END PROCEDURE POP;
  END CAPSULE CAPSULE_ID;
$END $PROCEDURE STACK_GENERATOR;
$CALL STACK_GENERATOR(100, FIXED(-100,100))
  $DECLARE INT_STACK_CAPSULE, INT_STACK;
VAR IS1: INT_STACK INIT ?;
VAR IS2: INT_STACK INIT ?;
$CALL STACK_GENERATOR(50, BOOLEAN)
  $DECLARE BOOL_STACK_CAPSULE, BOOL_STACK;
VAR BS: BOOL_STACK INIT ?;
CALL STACK_INIT(IS1);
CALL STACK_INIT(IS2);
CALL STACK_INIT(BS);
CALL PUSH(IS1,10);
CALL PUSH(BS,TRUE);

```

14.5 Type-Unresolved Parameters.

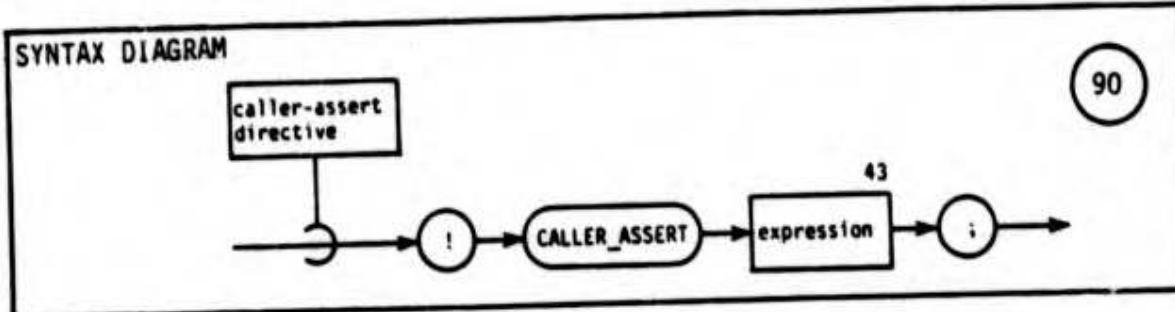
14.5.1 ANY

ANY may be used in place of a <type reference> in the declaration of a formal parameter to a routine. The effect is a generalization of the overload facility. There is no restriction on the corresponding actual parameter which may be supplied in an invocation of the routine; i.e., the actual parameter may be of any type.

In conjunction with the <caller assert directive> and <redeclare directive> described below, the ANY facility allows the user to write general routines which can perform compile-time discrimination to establish the appropriate <body>. An example of ANY's utility is the writing of the I/O library routines, as shown in Appendix A.

. It should be noted that ANY is not a type, but instead corresponds to "unresolved type" in much the same way that an asterisk in a <type reference> corresponds to "unresolved bound". Thus, if X is a formal parameter specified as ANY, then the type of X is resolved at each invocation of the routine; i.e., at different invocations of the routine, differently typed actual parameters may be supplied for X. (We point out that an implementation may choose to expand inline a routine with an ANY parameter, in order to guarantee that type-checking is performed at compile-time.)

14.5.2 Caller Assertions.



The <caller-assert directive> is a compile-time feature analogous to the run-time <assert statement>. It allows the writer of a routine which takes a formal ANY parameter to specify restrictions on the actual parameters which may be passed in an invocation of the routine. The <caller-assert directive> must be immediately contained in a <routine declaration>. The <expression> must be compile-time evaluable. If a formal ANY parameter is referenced in the <expression>, the corresponding actual parameter will be used. If the <expression> evaluates to FALSE or is not compile-time evaluable, a compile-time error occurs at the point of invocation.

Example:

```
PROCEDURE P(X,Y: ANY, RESULT B: BOOLEAN INIT ?);
  !CALLER_ASSERT IS_ARRAY(X) AND IS_ARRAY(Y) AND TYPE_EQ(X,Y);
  :
END PROCEDURE P;
```

The caller-assertion above checks that the types of X and Y are the same array type.

14.5.3 Type Comparison Functions.

REDL provides several BOOLEAN-valued functions which are useful in conjunction with ANY parameters and caller assertions.

(1) `TYPE_EQ(X,Y)` returns TRUE if and only if X and Y are data objects of the same type. `TYPE_EQ` is always compile-time evaluable.

(2) `REP_EQ(X,Y)` returns TRUE if and only if X and Y are data objects of the same type and representation. Depending on X and Y, `REP_EQ` may or may not be compile-time evaluable.

(3) `IS_ENUM(X)`, `IS_ARRAY(X)`, `IS_RECORD(X)`, `IS_UNION(X)`, `IS_POINTER(X)`, `IS_FILE(X)` and `IS_MRECORD(X)` return TRUE if and only if the type of data object X is an enumeration type, array type, record type, union type, pointer type, file type, or machine-record type, respectively. Each of these functions is always compile-time evaluable.

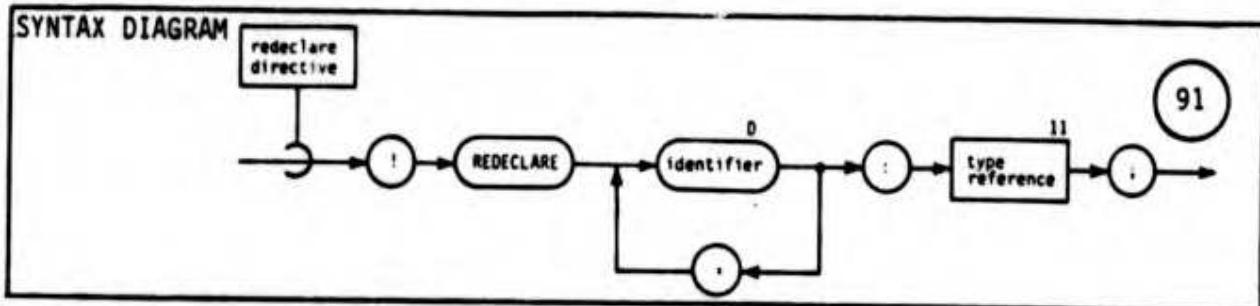
(4) `FILE_COMPATIBLE(X,Y)` is a specialized function which facilitates the writing of I/O routines. This function returns TRUE if and only if X is a file and Y is a data object whose type and representation are equal to the type and representation of X's components. This function may or may not be compile-time evaluable.

As a notational convenience, REDL permits `TYPE_EQ`, `REP_EQ`, and `FILE_COMPATIBLE` to be invoked with arguments which are type identifiers or resolved <type reference>s.

Example:

```
VAR X: FIXED(-100..100) INIT 0;  
TYPE_EQ(X,FIXED) is TRUE  
REP_EQ(X,FIXED(0..100)) is FALSE
```

14.5.4 Redeclare Directive.



The <*redeclare directive*> is a machine-dependent facility and thus may only be used in <*program*>s containing a <*configuration directive*>. The most frequent anticipated usage of the <*redeclare directive*> is in connection with ANY parameters and caller assertions.

A <*redeclare directive*> must be immediately contained within a <*routine declaration*>, and the identifier referenced in such a <*directive*> must be a formal parameter or result variable of the routine. The effect of a <*redeclare directive*> is to cause the containing routine to be compiled as follows:

- (1) The <*type reference*> (or ANY specification) which appears in the header for the formal parameter is used for type-checking at invocations of the routine.
- (2) The <*type reference*> appearing in the <*redeclare directive*> is used for type-checking in the <*body*> of the routine.
- (3) The X_SIZE exception is raised if the storage requirements for the formal parameter are different before and after redeclaration (this is implementation-dependent).
- (4) The effect is implementation-dependent when, through a <*redeclare directive*>, the storage occupied by an object of one type is treated as though of another type.

As an example of the <redeclare directive>, the following function takes two parameters of type BITS, interprets them as integers, obtains their sum, and returns this value as an object of type BITS.

```
FUNCTION BITS_SUM(A,B: BITS(8)) RESULT R: BITS(8) INIT ?;
!REDECLARE A,B,R: FIXED(0..255);
R:=(A+B)MOD 256;
END FUNCTION BITS_SUM;
```

APPENDIX A: REDL EXAMPLES

A.1 I/O for Particular Machines

A.1.1 Illustrates a channel-I/O program for the AN/UYK-20.

This is an example of I/O processing done in REDL on the UNIVAC AN/UYK-20. It illustrates the use of many machine dependent features of the language, plus others:

```
CONFIGURATION
MACHINE_CODE routine
MRECORD with field overlays
ALIGN
%CONST
ADDRESS and LOC

/* This example will write N characters of data on a UNIVAC 1532
/* teletype and then write a carriage return (CR) and line feed (LF)

PROGRAM channel;
!PROMOTE ccm, write_line;
!CONFIGURATION MACHINE('AN/UYK-20'), TELETYPE('1532');

<**** io op codes *** *>
%CONST ock:BITS(16) INIT H'E706';
%CONST io :BITS(16) INIT H'E300';
%CONST ipr:BITS(16) INIT H'EC10';
%CONST hcr:BITS(16) INIT H'EC00';

TYPE ccm:MRECORD <* Channel Control Memory *>
    op:BITS(16);
    !OFFSET WORD(0);
    r:BITS(4);
    !OFFSET WORD(0), BIT(8);
    <* OVERLAYS PART OF OP *>
    s:ADDRESS;
    !OFFSET WORD(1); !SIZE BITS(16);
END MRECORD ccm;
TYPE io_chain:RECORD i1,i2,i3:ccm; i4,i5:BITS(16);
    END RECORD io_chain;
!ALIGN DOUBLEWORD(ccm, io_chain);
```

```

CONST effad:BITS(16) INIT H'0002';
CONST effwd:ccm INIT
    ccm(op:H'4001', r:H'0', a:LOC(effad));
CONST endlhead:STRING(2) INIT '^CR'^LF';
CONST endlne:ccm INIT
    ccm(op:H'4002', r:H'0', a:LOC(endlhead));
VAR outputad:ccm INIT ?;
CONST outchain:io_chain INIT
    io_chain(i1:ccm(op:io, r:H'2', a:LOC(effwd)),
              i2:ccm(op:io, r:H'1', a:LOC(outputad)),
              i3:ccm(op:io, r:H'1', a:LOC(endlne)),
              i4:iPr,
              i5:hcr);
CONST outinst:ccm INIT
    ccm(op:lock, r:H'1', a:LOC(outchain));
PROCEDURE write_line(buff:ADDRESS, outinst:ccm,
                     n:FIXED(0..300), VAR out:ADDRESS);
!INLINE;
!LINKAGE buff IN R3, /* second word of SD R2 */
    outinst IN R4, /* and R5, since ccm is 2 words */
    n IN R2, /* number of chars */
    out IN R6; /* where to put ccm */
!USING R2,R3,R4,R5,R6;
MACHINE_CODE
. LK R2,040000 .SET UP CHANNEL CONTROL MEMORY FOR WRITE,
. OR ONTO n IN R2.
    SDI R2,R6 . STORE ccm IN R2+R3(buff) INTO out
    SD R4,0140 . STORE outinst INTO 140
    IOCR . INITIATE I/O
END MACHINE_CODE;
END PROCEDURE write_line;

CONST mes:STRING INIT 'Welcome to REDL.';

CALL write_line(LOC(mes), outinst, SIZE(mes), LOC(outputad));
/* write message to tty */

END PROGRAM channel;

```

A.1.2. Illustrates an Interrupt-driven I/O Program for the PDP-11.

This program illustrates several machine-dependent features of REIDL as applied to an interrupt driven I/O program for the PDP-11.

Illustrated are:

STORE ... AT
ADDRESS
CONNECT
LINKAGE (for interrupts)

```
PROGRAM tty_input_line;
  *
<**>
<* Simple PDP-11 interrupt driven read-one-line program
<* Ignores all errors.
<* Echoes all characters read.
<* Reads till line full or "CR" is read.
<* Echoes the "CR" and then echoes a "LF" as well.
<* Stores and counts the "LF" if it will fit.
<**>

  ! CONFIGURATION MACHINE('PDP-11');

  EVENT keyhit INIT 0;    <* says a character was typed
  EVENT printed INIT 1;   <* says that an echo is completed

<* Terminal data and status words, for input and for output

  VAR tty_in_data , tty_out_data : CHAR INIT ? ;
  VAR tty_in_status , tty_out_status : BITS(16) INIT ?;

  ! STORE tty_in_data AT ADDRESS ('0777562');
  ! STORE tty_out_data AT ADDRESS ('0777566');
  ! STORE tty_in_status AT ADDRESS ('0777560');
  ! STORE tty_out_status AT ADDRESS ('0777564');

  %CONST tty_inactive : BITS(16) INIT X'0000';
  %CONST tty_active : BITS(16) INIT X'0005';
```

```

<* The interrupt handler for the terminal responding to input

PROCEDURE tty_in_int_handler;
  ! IMPORT keyhit;
  ! LINKAGE INTERRUPT ;
  SEND keyhit;
END PROCEDURE tty_in_int_handler;

<* The interrupt handler for the terminal responding to output finished

PROCEDURE tty_out_int_handler;
  ! IMPORT printed;
  ! LINKAGE INTERRUPT ;
  SEND printed;
END PROCEDURE tty_out_int_handler;

<* The read-a-line procedure, called by the user
<* deactivates the terminal, sets up interrupt handlers, and
<* then activates the handlers.
<* Waits for enough input to fill the buffer or until
<* a CR is recognized. Echoes input.
<* Adds in a final LF if a CR was recognized and if there is room for it

PROCEDURE tty_read_line(
  VAR space : STRING(*), <* the buffer
  RESULT len : FIXED(MIN_INTEGER..MAX_INTEGER) INIT SIZE(space));
IMPORT tty_in_status , tty_out_status , tty_inactive ,
      tty_active , tty_in_int_handler , tty_out_int_handler ,
      tty_in_data , tty_out_data , keyhit , printed ;

  VAR ch : CHAR INIT ? ;

  <* deactivate terminal, set up interrupt handlers,
  <* and then activate the terminal.

  tty_in_status := tty_inactive;
  tty_out_status := tty_inactive;

BEGIN

  <* in this begin-block, the interrupt handlers are set up
  <* set up of the handlers had to await de-activation of TTY

  ! CONNECT tty_in_int_handler TO 'tty_input_int';
  ! CONNECT tty_out_int_handler TO 'tty_out_int';

  tty_in_status := tty_active;
  tty_out_status := tty_active;

  <* now loop, filling up the buffer 'space'

```

```

fill_buff_loop:
  FOR i FROM 1 UPTO SIZE(space) DO
    WAIT keyhit; /* Wait until user types a character
    ch := tty_in_data;
    space(i) := ch;

    WAIT printed; /* Wait until the previously echoed character
                     /* has been printed.
    tty_out_data := ch; /* this 'store' in itself initiates the
                         /* echo of the
                         **'ch'

  IF (ch = "cr")
    THEN
      WAIT printed; /* wait until the CR has been echoed
      tty_out_data := "lf"; /* add LF to make CR-LF
      IF (i < SIZE(space))
        THEN
          space(i+1) := "lf";
          len := i+1;
        END IF;
        EXIT fill_buff_loop;
      END IF;
    END FOR fill_buff_loop;

  /* now shut down the terminal
  /* must wait for output to terminate and must
  /* prepare for next call on this routine

  tty_in_status := tty_inactive;
  WAIT printed;
  tty_out_status := tty_inactive;
  SEND printed; /* so that the initial WAIT on 'printed' will be OK
END;
END PROCEDURE tty_read_line;
END PROGRAM tty_input_line;

```

A.2 Methods for Construction of Standard I/O Library Procedures.

This program fragment illustrates the machine-dependent techniques that may be used to code the standard I/O library procedures. In particular, the following are illustrated:

```
the ANY parameter
REDECLARE
CALLER_ASSERT
INLINE
and very minimal use of MACHINE_CODE

<* fragment ... *>

RANGE status_range : -100..100;

PROCEDURE read (
  VAR f : ANY ,
  RESULT v : ANY INIT ? ,
  RESULT s : FIXED(status_range) INIT ? );
  ! INLINE;
  ! IMPORT file_rep, read_dispatch;
  ! CALLER_ASSERT FILE_COMPATIBLE(f,v);
  ! REDECLARE f : file_rep;    <* the insider's view of files
  ! REDECLARE v : BITS(f.component_size);
                           <* opening f has provided the component_size
IF ( f.open & f.read )
  THEN
    CALL read_dispatch(f,v,s);
  ELSE
    s := -1;
  END IF;
END PROCEDURE read;

PROCEDURE read_dispatch (
  VAR f : file_rep,
  RESULT v : BITS(*) INIT ? ,
  RESULT s : FIXED(status_range) INIT ? );
  ! INLINE;
  ! LINKAGE STANDARD; <* with machine code its best to specify
                           the linkage *>
```

MACHINE_CODE

Here, insert machine code to accomplish the call:

CALL f.driver(f,v,s)

where the address of the driver appropriate to the file f
has been stored in f.driver by OPEN.

Please note that the driver may itself be written
principally in REIDL with only minor use of MACHINE_CODE.

END MACHINE_CODE;
END PROCEDURE read_dispatch;

A.3 Parallel Processing Examples.

A.3.1 Illustrates use of MONITORS for Synchronization in a standard parallel processing problem.

This program illustrates the use of the parallel processing facilities in REDL. The example which is coded here is the 'Producers and consumers' problem, in which multiple producers and consumers share a buffer, the consumers extracting an item from the buffer at will, and the producers adding items to the buffer at will, as long as the buffer is not empty or full, respectively.

It should be noted that the use of integers in events (rather than booleans) makes it very easy to associate the events which signal FULL and EMPTY buffers with the buffer size.

This example demonstrates the use of MONITORS, FORKS, PATHs, and EVENTS.

```
PROGRAM produce_and_consume;

!PervasivE produce, consume;
;

MONITOR CAPSULE buffer_manager;

    ! EXPORT produce, consume;
    ! PervasivE buffer_not_full , buffer_not_empty , buffer_manager;

%CONST buffer_size: FIXED ( 1 ) INIT 100 ;
TYPE buffer_type:ARRAY (1..buffer_size:) OF CHAR;
VAR buffer:buffer_type INIT ?;
EVENT buffer_not_full INIT buffer_size ;
EVENT buffer_not_empty INIT 0 ;

PROCEDURE produce;
    WAIT buffer_not_full RELEASE buffer_manager;
        /* produce an item into the buffer
        SEND buffer_not_empty;
END PROCEDURE produce;

PROCEDURE consume;
    WAIT buffer_not_empty RELEASE buffer_manager;
        /* consume an item from the buffer
        SEND buffer_not_full;
END PROCEDURE consume;

END CAPSULE buffer_manager;

PROCEDURE start_here; !MAIN;
```

```
FORK
  /* The paths which follow represent
  /* two Producers and two consumers

  PATH;
    CALL Produce;
  END PATH;

  PATH;
    CALL Produce;
  END PATH;

  PATH;
    CALL consume;
  END PATH;

  PATH;
    CALL consume;
  END PATH;

END FORK;

END PROCEDURE start_here;

END PROGRAM produce_and_consume;
```

A.3.2 Illustrates Parallel Processing Where One Job is a Real-time Job Driven at 100-millisecond Intervals.

This program performs some 'WORK' beginning at 11AM and repeating every 100 milliseconds while a condition holds.

The program illustrates the use of FORNs, PRIORITYs, PATHs, and the PAUSE statement, and is an example of a cyclically rescheduled task.

```
PROGRAM cycle ;

! CONFIGURATION MACHINE('XYZ');

FUNCTION clock RESULT c : FIXED(0..1E8) INIT ? ;
  ! LINKAGE c IN R0;
  MACHINE_CODE
    Here is machine code to read the system
    'Wall-time' clock. Its readout is
    in milliseconds since midnight.
  END MACHINE_CODE;
END FUNCTION clock;

FUNCTION condition_is_true RESULT x: BOOLEAN INIT ? ;
.....
END FUNCTION condition_is_true;

PROCEDURE cycle_Proc ;
  !IMPORT condition_is_true , work , clock ;
  VAR clock_t, next_time : FIXED(0..1E8) INIT 0 ;
  %CONST eleven_am : FIXED INIT 1000*60*60*11 ;
    /* millisecs since midnight */
  %CONST one_hundred_ms: FIXED INIT 100 ;

  clock_t := clock;
  IF clock_t < eleven_am
    THEN PAUSE (eleven_am - clock_t) MILLISECONDS;
  END IF;

  WHILE condition_is_true DO
    next_time := clock + one_hundred_ms;

    CALL work;  /* This is the external job ... */


```

```

        clock_t := clock;
        IF clock_t < next_time
            THEN PAUSE (next_time - clock_t) MILLISECONDIS;
        END IF;

    END WHILE;

END PROCEDURE cycle_Proc;

PROCEDURE work;
...
END PROCEDURE work;

PROCEDURE main_body : !MAIN ;
    !IMPORT cycle_Proc;

    FORK

        PATH;
        /* THE CODE FOR ALL OF THE OTHER
        /* OPERATIONS TO BE PERFORMED WHILE
        /* THE CYCLICAL PROCEDURE IS RUNNING ON THE
        /* SIDE IS INSERTED HERE
    END PATH;

    PATH;
        ! PRIORITY 10;
        CALL cycle_Proc;
    END PATH;

    END FORK;

END PROCEDURE main_body;

END PROGRAM cycle;

```

A.4 Examples From Graphics and Communications Applications.

A.4.1 Graphics Example.

This example demonstrates how complex algorithms can be coded in REIL. The algorithm performs clipping on lines for graphics applications, in order to remove parts of lines that are not contained in a given region. Some features to note:

```
abstract types: Point and line
RECORD construction and assignment
overloaded operators for points
RANGE
VAR, CONST, and RESULT parameters
ASSERT
PERVasive
PROCEDURES and FUNCTIONS
short circuited operators

PROGRAM clipping_algorithm;
!PERVasive fr, pt, line, rectangle;
!PERVasive is_pt_in, slope, slope_recip, clip;
RANGE fr:-2B15..2B15-1 ; /* range for fixeds */
TYPE pt:RECORD x, y:FIXED(fr); END RECORD pt;
TYPE line:ARRAY (1..2) OF pt;
TYPE rectangle:RECORD
    left, right, bottom, top:FIXED(fr);
    END RECORD rectangle;
/* left<right, bottom<top */

OVERLOAD OPERATOR + (a,b:pt) RESULT c:pt INIT
    pt(x:a.x + b.x, y:a.y + b.y);
    !ASSOCIATIVE; !COMMUTATIVE;
END OPERATOR +;

OVERLOAD OPERATOR DIV (p:pt, d:FIXED(fr)) RESULT pd:pt INIT
    pt(x:p.x DIV d, y: p.y DIV d);
END OPERATOR DIV;

FUNCTION is_pt_in(p:pt, r:rectangle) RESULT inside:BOOLEAN
    INIT p.x >= r.left & p.x <= r.right & p.y >= r.bottom
    & p.y <= r.top ;
    /* determines if pt is inside rectangle */
END FUNCTION is_pt_in;

/* need slope and slope_recip to avoid zero divide */

FUNCTION slope(l:line) RESULT s:FIXED(fr) INIT
    (l(1).y - l(2).y) DIV (l(1).x - l(2).x);
END FUNCTION slope;

FUNCTION slope_recip(l:line) RESULT s:FIXED(fr) INIT
    (l(1).x - l(2).x) DIV (l(1).y - l(2).y);
END FUNCTION slope_recip;
```

```

PROCEDURE clip(VAR l:line, CONST region:rectangle,
RESULT valid:BOOLEAN INIT TRUE);

<* if any part of l is inside region, valid is set TRUE
<* and that part is returned in l. otherwise, valid is
<* set to FALSE. note valid is initially TRUE.*>

!PERVERSIVE region;

FUNCTION quick_elim(l:line) RESULT elim:BOOLEAN INIT ?;
<* determines if line is completely above, below, or to
<* the side of region. *>
elim := l(1).x < region.left & l(2).x < region.left
    ! l(1).x > region.right & l(2).x > region.right
    ! l(1).y < region.bottom & l(2).y < region.bottom
    ! l(1).y > region.top & l(2).y > region.top ;
END FUNCTION quick_elim;

PROCEDURE one_pt_fix_up(VAR l:line, CONST inside, outside:
FIXED(1..2));
<* l(inside) is inside region, and
<* l(outside) is outside region.
<* on return, l(outside) is the point of intersection
<* of the line and the region boundary.*>
ASSERT is_pt_in(l(inside), region) AND
    NOT is_pt_in(l(outside), region);
IF l(outside).x < region.left
THEN
    l(outside) := pt(x:region.left,
        y:l(inside).y + (region.left - l(inside).x) *
        slope(l));
ELSEIF l(outside).x > region.right
THEN
    l(outside) := pt(x:region.right,
        y:l(inside).y + (region.right - l(inside).x) *
        slope(l));
END IF;
IF l(outside).y < region.bottom
THEN
    l(outside) := pt(x:l(inside).x +
        (region.bottom - l(inside).y) * slope_recip(l),
        y:region.bottom);
ELSIF l(outside).y > region.top
THEN
    l(outside) := pt(x:l(inside).x +
        (region.top - l(inside).y) * slope_recip(l),
        y:region.top);
END IF;
ASSERT is_pt_in(l(outside), region);
END PROCEDURE one_pt_fix_up;

<* ***** beginning of code for clip ***** *>

```

```

IF is_pt_in(l(1), region)
THEN
    IF NOT is_pt_in(l(2), region)
    THEN
        CALL one_pt_fix_UP(l, 1, 2);
    END IF;
ELSEIF is_pt_in(l(2), region)
THEN
    CALL one_pt_fix_UP(l, 2, 1);
ELSEIF quick_elim(l) THEN valid := FALSE;
ELSE
    loop:
    WHILE TRUE DO
        CONST mid_pt:pt INIT (l(1) + l(2)) DIV 2 ;
        VAR temp;line INIT line(l(1), mid_pt);
        IF is_pt_in(mid_pt, region)
        THEN
            CALL one_pt_fix_UP(temp, 2, 1);
            l(1) := temp(1);
            CALL one_pt_fix_UP(l, 1, 2);
            EXIT loop;
        ELSEIF quick_elim(temp)
        THEN
            IF quick_elim(line(l(2), mid_pt))
            THEN valid := FALSE; EXIT loop;
        END IF;
        l(2) := mid_pt;
    ELSE
        l(1) := mid_pt;
    END IF;
    END WHILE loop;
END IF;
ASSERT (NOT valid) XOR is_pt_in(l(1), region) AND
    is_pt_in(l(2), region);
END PROCEDURE clip;
END PROGRAM clippings_algorithm;

```

A.4.2 Communications Software Example.

Illustrates proper use of EXTERNAL procedures to take advantage of special hardware.

The following program demonstrates an embedded application coded in REIDL. Some of the features utilized are:

```
CONFIGURATION
PERVERSIVE
PROMOTE
ACCESS
MRECORD
ALIGN
TYPE complex
OVERLOADed OPERATORs
EXTERNAL Procedures
BITS
%CONST (compile time constants)

<* Link 11 is a double sideband suppressed carrier HF radio
<* transmission system. Phase shift modulation is applied to
<* a series of audio tones, which in turn is used to amplitude
<* modulate the radio carrier. *>

<* This program generates a sequence of samples, which when used
<* as input to a digital/analos converter will create the audio
<* tones for this system, ready to be applied to the carrier. *>

<* The program is intended to be run on the AN/UY5-1(PROTEUS)
<* computer, which has a special high-speed pipelined arithmetic
<* unit(AU). The AU is used for arithmetic-intensive calculations
<* such as the FFT, and thus programs normally deal with data in a
<* format suitable for the AU. This program is machine dependent,
<* as it uses the AU for the inverse FFT and bandshifting operations.*>

PROGRAM link_11_transmit;
!ACCESS hamming(hamming_encode);
<* Performs hamming encoding on a BITS(30) *>
!PROMOTE link_11_xmt, frame;
!CONFIGURATION MACHINE('AN/UY5-1'), OS('CROS_EXEC');
!PERVERSIVE frame, compl, compl_array, phase, phase_array,
  fs, fr, compl_zero;
TYPE frame:UNORDERED ENUM(no_signal, preamble, phase_ref,
  data, control);
VAR prev_frame_type:frame INIT no_signal;
```

```

XCONST fs:FIXED(1B~8) INIT 1B~8; /* scale and */
RANGE fr:(-100..100); /* range for fft */
TYPE compl:MRECORD
    real:FIXED(fs,fr);
    !OFFSET WORD(0), BIT(0);
    !SIZE BITS(16);
    imag:FIXED(fs,fr);
    !OFFSET WORD(1), BIT(0);
    !SIZE BITS(16);
END MRECORD compl;
!ALIGN DOUBLEWORD(compl);

TYPE compl_array(i,j):ARRAY (i..j) OF compl;
CONST compl_zero:compl INIT compl(real:0, imag:0);

OVERLOAD OPERATOR * (a:FIXED(fs, fr), c:compl)
    RESULT r:compl INIT
        compl(real:a * c.real, imag:a * c.imag);
    !INLINE;
END OPERATOR *;

TYPE phase:FIXED(0..7);
/* Phase is a multiple of 45 degrees */
TYPE phase_array(i, j):ARRAY (i..j) OF phase;

OVERLOAD OPERATOR + (p1, p2:phase) RESULT p3:phase
    INIT phase((FIXED(p1) + FIXED(p2)) MOD 8);
    !INLINE;
END OPERATOR +;

/* Procedures performed by Proteus hardware */
PROCEDURE fft_inverse(VAR fft_buf:compl_array(0,63));
    !EXTERNAL;
    !LINKAGE PROTEUS_AP;
END PROCEDURE fft_inverse;

PROCEDURE bandshift(VAR fft_buf:compl_array(0,63),
    VAR bs_buf:compl_array(0,127),
    CONST a,b:FIXED(fs,fr));
    !EXTERNAL;
    !LINKAGE PROTEUS_AP;
END PROCEDURE bandshift;

VAR tone_phase:phase_array(2,16) INIT
    phase_array(15 OF phase(0));

VAR out_buf:compl_array(0, 8191) INIT ? ;
VAR out_ptr:FIXED(0..8191) INIT 0;

PROCEDURE link_11_xmt(CONST frame_type:frame,
    CONST data_word:BITS(30));
    !IMPORT tone_phase, out_buf, out_ptr;
    !IMPORT hamming_encode, fft_inverse, bandshift;

    XCONST sart_2_div_2:FIXED(fs) INIT ROUND(fs, 0.707);
    XCONST sart_5:FIXED(fs) INIT ROUND(fs, 2.236);

```

```

FUNCTION di_bit_phase(bi:BITS(2)) RESULT  p:phase INIT ?;
  IF bi = B'00' THEN p := phase(3);
  ELSEIF bi = B'01' THEN p := phase(1);
  ELSEIF bi = B'10' THEN p := phase(5);
  ELSE p := phase(7);
  END IF;
END FUNCTION di_bit_phase;
CONST compl_phase:compl_array(0,7) INIT
  compl_array(compl(real:1, imag:0),
    compl(real, imag:sqrt_2_div_2),
    compl(real:0, imag:1),
    compl(real:-sqrt_2_div_2, imag:sqrt_2_div_2),
    compl(real: -1, imag:0),
    compl(real, imag: -sqrt_2_div_2),
    compl(real:0, imag: -1),
    compl(real:sqrt_2_div_2, imag: -sqrt_2_div_2));
VAR fft_buf:compl_array(0,63) INIT
  compl_array(64 OF compl_zero);
VAR bs_buf:compl_array(0, 127) INIT ? ;
%CONST tone_1:FIXED INIT 5;
%CONST tone_2:FIXED INIT 8;
%CONST tone_3:FIXED INIT 9;
%CONST tone_4:FIXED INIT 10;
%CONST tone_5:FIXED INIT 11;
%CONST tone_6:FIXED INIT 12;
%CONST tone_7:FIXED INIT 13;
%CONST tone_8:FIXED INIT 14;
%CONST tone_9:FIXED INIT 15;
%CONST tone_10:FIXED INIT 16;
%CONST tone_11:FIXED INIT 17;
%CONST tone_12:FIXED INIT 18;
%CONST tone_13:FIXED INIT 19;
%CONST tone_14:FIXED INIT 20;
%CONST tone_15:FIXED INIT 21;
%CONST tone_16:FIXED INIT 26;

```

```

/* statement list for link_11_xmt begins here
SELECT frame_type FROM
CASE no_signal => /* clear buffer to zero */
    bs_buf := compl_array(128 OF compl_zero);
OTHERWISE => /* put signal samples in buffer */
    SELECT frame_type FROM
    CASE preamble =>
        IF prev_frame_type = no_signal THEN
            tone_phase := phase_array(15 OF phase(0));
            /* initialize tone phases to zero */
        END IF;
        fft_buf(tone_1) := compl(real:4, imag:0);
        /* doppler tone has amplitude 4 */
        fft_buf(14 AT tone_2) := compl_array(14 OF
            compl_zero);
        tone_phase(16) := tone_phase(16) + phase(4);
        /* advance frame sync tone by 180 degrees */
        fft_buf(tone_16) := 2 * compl_phase(
            FIXED(tone_phase(16)));
        /* frame sync tone has amplitude 2 */
        fft_buf(tone_1) := compl(real:4, imag:0);
        /* doppler tone has amplitude 4 */
        fft_buf(14 AT tone_2) := compl_array(14 OF
            compl_zero);
        tone_phase(16) := tone_phase(16) + phase(4);
        /* advance frame sync tone by 180 degrees */
        fft_buf(tone_16) := 2 * compl_phase(
            FIXED(tone_phase(16)));
        /* frame sync tone has amplitude 2 */
    CASE phase_ref =>
        fft_buf(tone_1) := compl(real:sqrt_5, imag:0);
        fft_buf(14 AT tone_2) := compl_array(14 OF
            compl(real:1, imag:0));
        tone_phase(16) := tone_phase(16) + phase(4);
        fft_buf(tone_16) := compl_phase(FIXED(
            tone_phase(16)));
    CASE data, control =>
        VAR w:BITS(30) INIT data_word;
        VAR b2:BITS(2) INIT ?;
        IF frame_type = data THEN
            w := hammins_encode(w);
        END IF;
        fft_buf(tone_1) := compl(real:sqrt_5, imag:0);

        FOR i FROM 2 UPTO 15 DO
            b2 := w(2 AT 33 - 2*i);
            tone_phase(i) := tone_phase(i) +
                di_bit_phase(b2);
            fft_buf(tone_2 - 2 + i) :=
                compl_phase(FIXED(tone_phase(i)));
        END FOR;
        b2 := w(2 AT 1);
        tone_phase(16) := tone_phase(16) +
            di_bit_phase(b2);
        fft_buf(tone_16) := compl_phase(FIXED(
            tone_phase(16)));^L

```

```

END SELECT;
CALL bandshift(fft_buf, bs_buf, 0, 1B-7);
  /* shift tones by +55 hz, 128 samples */
CALL fft_inverse(fft_buf);
  /* generate time domain samples of tones */
END SELECT;
BEGIN
  /* put samples in the output buffer, using both out_buf
   * and bs_buf circularly */
CONST n_samples:FIXED INIT 94;
CONST bs_ptr:FIXED(0..127) INIT out_ptr MOD 128;
IF bs_ptr + n_samples <= 128
  THEN
    out_buf(n_samples AT out_ptr) :=
      bs_buf(n_samples AT bs_ptr);
  ELSE
    CONST last_part:FIXED(0..127) INIT bs_ptr +
      n_samples - 128;
    CONST first_part:FIXED(0..127) INIT
      n_samples - last_part;
    out_buf(first_part AT out_ptr) :=
      bs_buf(first_part AT bs_ptr);
    out_ptr := (out_ptr + first_part) MOD 8192;
    out_buf(last_part AT out_ptr) :=
      bs_buf(last_part AT 0);
    out_ptr := out_ptr + last_part;
  END IF;
END BEGIN;
prev_frame_type := frame_type;
END PROCEDURE link_11_xmt;
END PROGRAM link_11_transmit;

```

APPENDIX B: ASCII CHARACTERS

B.1 Character Classes

CHARACTER	CHARACTER CLASS							
	BLANK	LETTER	DECIMAL DIGIT	BINARY DIGIT	OCTAL DIGIT	HEXADECIMAL DIGIT	PRINTING GRAPHIC EXCEPT <	PRINTING GRAPHIC EXCEPT ~ AND ^
SP	X						X	X
HT	X							X
LF VT FF CR	X							X X
A - F		X			X	X	X	X
G - Z	X						X	X
0 - 1		X	X	X	X	X	X	X
2 - 7		X	X	X	X	X	X	X
8 - 9		X			X	X	X	X
~								X
*						X		X
*						X	X	X
>						X	X	X
!"#\$%&()*,./;; <=?@(!)~`()-						X	X	X
a-z	X					X	X	X

- NOTES: (1) The symbols SP, HT, LF, VT, FF, and CR represent the Space, Horizontal Tab, Line Feed, Vertical Tab, Form Feed, and Carriage Return characters, respectively.
- (2) The characters outside the set displayed (NUL, SOH, etc.) may not appear in the program text.

B.2 CHAR Literals.

^NUL^	^ETB^	^.^
^SOH^	^CAN^	^/^
^STX^	^EM^	^0^ through ^9^
^ETX^	^SUB^	^:^
^EOT^	^ESC^	^;^
^ENQ^	^FS^	^<^
^ACK^	^GS^	^=^
^BEL^	^RS^	^>^
^BS^	^US^	^?^
^HT^	^ ^	^@^
^LF^	^!^	^A^ through ^Z^
^VT^	^"^	^[^
^FF^	^#^	^\"^
^CR^	^\$^	^]^
^SO^	^%^	^^^
^SI^	^&^	^ ^
^DLE^	^'^	^-
^DC1^	^(^	^a^ through ^z^
^DC2^	^)^	^{^}
^DC3^	^*^	^ ^
^DC4^	^+^	.^}^
^NAK^	^,^	^~^
^SYN^	^-^	^DEL^

APPENDIX C: DIAGRAM CROSS-REFERENCE

Sections C.1 and C.2 list the lexical and syntax diagrams alphabetically and display the diagram index for each entry.

Sections C.3 and C.4 contain a cross reference for the lexical and syntax diagrams. Next to each entry is the list of diagram indices in which the given diagram is referenced. (The diagrams for lexical unit and <program> are the only ones which are not referenced.)

C.1 Lexical Diagrams (alphabetical)

	Diagram Identifier	Page Number
bits literal	I	3-9
comment	J	3-11
identifier	D	3-4
lexical unit	A	3-2
literal	F	3-6
name	C	3-3
numeric literal	G	3-7
special enumeration symbol	E	3-6
string literal	H	3-8
token	B	3-3
token separator	J	3-11

C.2 Syntax Diagrams (alphabetical)

	Diagram Identifier	Page Number
access directive	86	14-2
align directive	81	13-9
array component	15	5-32
array constructor	17	5-35
array constructor term	17	5-35
array slice	16	5-33
array type	14	5-30
assert statement	63	9-23
assignment statement	50	9-3
associative directive	42	7-22
basic statement	48	9-1
begin statement	51	9-5
body	36	7-11
call statement	60	9-20
caller-assert directive	90	14-18
capsule declaration	28	6-1
case alternative	54	9-9
case label	54	9-9
case label range	54	9-9
commutative directive	41	7-21
compile-time actual parameter	89	14-11
compile-time call command	89	14-11
compile-time command	4	4-7
compile-time constant declaration	87	14-6
compile-time else clause	88	14-9
compile-time elseif-then clause	88	14-9
compile-time if command	88	14-9
compile-time if-then clause	88	14-9
compile-time procedure declaration	89	14-11

	Diagram Identifier	Page Number
conditional statement	52	9-6
configuration directive	78	13-3
connect directive	82	13-10
control transfer statement	59	9-19
const-or-var declaration	10	5-8
declaration	3	4-5
directive	2	4-3
else clause	53	9-7
elseif-then clause	53	9-7
empty statement	49	9-3
enumeration type	13	5-27
event declaration	73	12-11
exception declaration	65	11-2
exception statement	66	11-3
exit statement	61	9-21
export directive	29	6-3
expression	43	8-6
external directive	84	13-14
factor	43	8-8
file property	64	10-2
file type	64	10-2
fork statement	71	12-3
formal parameter list	31	7-4
formal parameter pack	31	7-4
for-range phrase	58	9-16
for statement	58	9-16
function declaration	39	7-17
function invocation	47	8-13
generated type	12	5-27
goto statement	62	9-22
if statement	53	9-7
if-then clause	53	9-7

	Diagram Identifier	Page Number
import directive	7	4-15
inline directive	33	7-7
linkage directive	83	13-11
logical factor	43	8-6
logical term	43	8-6
loop statement	55	9-13
machine-dependent directive	77	13-1
machine-record type	79	13-4
machine-routine body	83	13-11
main directive	38	7-16
multi-path statement	70	12-1
object component	45	8-11
object constructor	46	8-12
off directive	69	11-7
offset directive	79	13-4
on statement	68	11-4
operator declaration	40	7-19
optimize directive	9	4-23
otherwise alternative	54	9-9
parameterized type declaration	26	5-50
path clause	71	12-3
pause statement	76	12-15
pervasive directive	6	4-13
pointer dereference	25	5-46
pointer type	24	5-45
primary	44	8-9
priority directive	72	12-7
procedure declaration	36	7-11
program	1	4-1
promote directive	85	14-1

	Diagram Identifier	Page Number
raise statement	67	11-3
range declaration	27	5-58
readonly directive	8	4-19
record component	19	5-39
record constructor	20	5-39
record type	18	5-38
recursive directive	34	7-9
redeclare directive	91	14-20
reentrant directive	35	7-10
relational expression	43	8-7
relational operator	43	8-7
repeat statement	57	9-15
routine declaration	30	7-1
routine directive	32	7-6
safe directive	37	7-12
send statement	74	12-12
select statement	54	9-9
simple type declaration	11	5-10
size directive	79	13-4
simple expression	43	8-8
statement	5	4-8
store directive	80	13-8
term	43	8-8
type declaration	11	5-10
type reference	11	5-10
union alternative	22	5-43
union constructor	23	5-43
union type	21	5-41
using directive	83	13-11
wait statement	75	12-13
while statement	56	9-14

C.3 Lexical Diagrams (Cross-Reference)

A	lexical unit	
B	token	A
C	name	B 6 29 40 61 85 86
D	identifier	C I 5 7 8 10 11 13 14 17 18 19 20 21 22 23 26 27 28 31 36 39 40 44 47 51 53 54 56 57 58 60 62 64 65 67 68 69 71 73 74 75 78 79 80 81 82 83 86 87 89 91
E	special enumeration symbol	B H 13 44
F	literal	B 44
G	numeric literal	F
H	string literal	F 78 82 83 86
I	bits literal	F
J	token separator	A
	comment	J

C.4 Syntax Diagrams (Cross-Reference)

1	program	
2	directive	1 28 36 71 83
3	declaration	1 28 36 71
4	compile-time command	1 28 36 71
5	statement	36 89
6	pervasive directive	2
7	import directive	2
8	readonly directive	2
9	optimize directive	2
10	const-or-var declaration	3
11	type declaration	3
	simple type declaration	11
	type reference	10 11 14 18 21 23 24
		31 39 40 64 79 87 89 91
12	generated type	11
13	enumeration type	12
14	array type	12 26
15	array component	45
16	array slice	45
17	array constructor	46
	array constructor term	17
18	record type	12 26
19	record component	45

20	record constructor	46
21	union type	12 26
22	union alternative	45
23	union constructor	46
24	pointer type	12
25	pointer dereference	44
26	parameterized type declaration	11
27	range declaration	3
28	capsule declaration	3
29	export directive	2
30	routine declaration	3
31	formal parameter list	36 39 40
	formal parameter pack	31
32	routine directive	2
33	inline directive	32
34	recursive directive	32
35	reentrant directive	32
36	procedure declaration	30
	body	36 40 51 53 54 56 57 58 68 71 88 89
37	safe directive	32
38	main directive	32
39	function declaration	30
40	operator declaration	30

41	commutative directive	32
42	associative directive	32
43	expression	10 11 14 15 16 17 20 23 27 31 39 40 44 47 50 53 54 56 57 58 60 63 72 73 76 79 80 87 88 89 90
	logical term	43
	logical factor	43
	relational expression	43
	relational operator	43
	simple expression	43
	term	43
	factor	43
44	primary	15 16 19 22 25 43 50
45	object component	44
46	object constructor	44
47	function invocation	44
48	basic statement	5
49	empty statement	48
50	assignment statement	48
51	begin statement	48
52	conditional statement	48
53	if statement	52
	if-then clause	53
	elseif-then clause	53

	else clause	53
54	select statement	52
	case alternative	54
	otherwise alternative	54
	case label	54
	case label range	54
55	loop statement	48
56	while statement	55
57	repeat statement	55
58	for statement	55
	for-range phrase	58
59	control transfer statement	48
60	call statement	59
61	exit statement	59
62	goto statement	59
63	assert statement	48
64	file type	12
	file property	64
65	exception declaration	3
66	exception statement	5
67	raise statement	66
68	on statement	66
69	off directive	2
70	multipath statement	5
71	fork statement	70
	path clause	71

72	priority directive	2
73	event declaration	3
74	send statement	70
75	wait statement	70
76	pause statement	70
77	machine-dependent directive	2
78	configuration directive	77
79	machine-record type	12
	offset directive	77 79
	size directive	77 79
80	store directive	77
81	align directive	77
82	connect directive	77
83	machine-routine body	36 39 40
	using directive	77
	linkage directive	77
84	external directive	77
85	promote directive	2
86	access directive	2
87	compile-time-constant declaration	3
88	compile-time if command	4
	compile-time if-then clause	88
	compile-time elseif-then clause	88

	compile-time else clause	88
89	compile-time-procedure declaration	3
	compile-time call command	4
	compile-time actual parameter	89
90	caller-assert directive	2
91	redeclare directive	77

APPENDIX D: LALR(1) GRAMMAR FOR REDL

```

1  <PROGRAM> ::= <FROG HDR> <BODY> <FROG END> _I_
2  <FROG HDR> ::= PROGRAM <IDENT> ;
3  <FROG END> ::= END PROGRAM <OPT IDENT> ;
4  <BODY> ::= <COMMAND>
5      | <BODY> <COMMAND>
6  <COMMAND> ::= <DIRECT>
7      | <OCL>
8      | <LAS STM>
9      | <MCODE HCR> <STRING> <MCODE END>
10     | %IF <%IF BODY> <%IF END>
11     | %CALL <IDENT> <OPT %DCL ARGS> <OPT CT ARGS> ;
12     | <IDENT> ;
13  <MCODE HCR> ::= BEGIN MACHINE_CODE
14  <MCODE END> ::= END MACHINE_CODE ;
15  <%IF BODY> ::= <%THEN PART>
16      | <%THEN PART> %ELSE <BODY>
17      | <%THEN PART> %ELSEIF <%IF BODY>
18  <%THEN PART> ::= <EXP> %THEN <BODY>
19  <%IF END> ::= %END %IF ;
20  <OPT %DCL ARGS> ::=
21      | %DECLARE <IDENT LIST> :
22  <OPT CT ARGS> ::=
23      | ( <CT ARG LIST> )
24  <CT ARG LIST> ::= <CT ARG>
25      | <CT ARG LIST> , <CT ARG>
26  <CT ARG> ::= <COMMAND>
27      | <EXP>
28  <DIRECT> ::= ! IMPORT <NAME LIST> ;
29      | ! EXPORT <NAME LIST> ;
30      | ! PROMOTE <NAME LIST> ;
31      | ! ACCESS <ACCESS BODY> ;
32      | ! READONLY <IDENT LIST> ;
33      | ! PRIORITY <EXP> ;
34      | ! COMMUTATIVE ;
35      | ! ASSOCIATIVE ;
36      | ! SAFE ;
37      | ! MAIN ;
38      | ! INLINE ;
39      | ! RECURSIVE ;
40      | ! REENTRANT ;
41      | ! OPTIMIZE SPACE ;
42      | ! OPTIMIZE TIME ;
43      | ! OFF <IDENT LIST> ;
44      | ! CALLER_ASSERT <EXP> ;
45      | ! REDECLARE <REDECLARE BODY> ;
46      | ! CONFIGURATION <CONFIG BODY> ;
47      | ! STORE <STORE BODY> ;
48      | ! ALIGN <ALIGN BODY> ;
49      | ! CONNECT <CONNECT BODY> ;
50      | ! EXTERNAL ;
51      | ! USING <IDENT LIST> ;
52      | ! LINKAGE <LINKAGE BODY> ;

```

```

53  <ACCESS BODY> ::= <ACCESS>
54    | <ACCESS BODY> , <ACCESS>
55  <ACCESS> ::= <OPT STRING> <IDENT>
56    | <OPT STRING> <IDENT> ( <NAME LIST> )
57  <REDECLARE BODY> ::= <IDENT LIST> : <TYPE REF>
58  <CONFIG BODY> ::= <CONFIG>
59    | <CONFIG BODY> , <CONFIG>
60  <CONFIG> ::= <IDENT> ( <IDENT-STRING LIST> )
61  <STORE BODY> ::= <IDENT> AT <EXP>
62  <ALIGN BODY> ::= <ALIGN>
63    | <ALIGN BODY> , <ALIGN>
64  <ALIGN> ::= <IDENT> ( <IDENT LIST> )
65  <CONNECT BODY> ::= <CONNECT>
66    | <CONNECT BODY> , <CONNECT>
67  <CONNECT> ::= <IDENT> TO <STRING>
68  <LINKAGE BODY> ::= <LINKAGE>
69    | <LINKAGE BODY> , <LINKAGE>
70  <LINKAGE> ::= <NAME>
71    | <STRING>
72    | <IDENT> IN <IDENT>
73  <DCL> ::= TYPE <IDENT> <TYPE BODY> ;
74    | <V OR C> <IDENT LIST> : <V OR C BODY> ;
75    | XCONST <IDENT LIST> : <V OR C BODY> ;
76    | RANGE <IDENT> : <RANGE EXP> ;
77    | <CAP HDR> <BODY> <CAP END>
78    | <FROC HDR> <BODY> <FROC END>
79    | <FUNC HDR> <BODY> <FUNC END>
80    | <OPER HDR> <BODY> <OPER END>
81    | <CT PROC HDR> <BODY> <CT PROC END>
82    | EXCEPTION <IDENT LIST> ;
83    | EVENT <IDENT LIST> INIT <EXP> ;
84  <TYPE BODY> ::= : <TYPE REF>
85    | : <GEN TYPE>
86    | ( <IDENT LIST> ) : <COMP TYPE>
87  <V OR C> ::= VAR
88    | CONST
89  <V OR C BODY> ::= <TYPE REF> INIT <?EXP>
90  <CAP HDR> ::= <OPT MONITOR> CAPSULE <IDENT> ;
91  <CAP END> ::= END CAPSULE <OPT IDENT> ;
92  <PROC HDR> ::= <OPT OVERLOAD> PROCEDURE <IDENT> <OPT PARMs> ;
93  <FROC END> ::= END PROCEDURE <OPT IDENT> ;
94  <FUNC HDR> ::= <OPT OVERLOAD> FUNCTION <IDENT> <OPT PARMs> <RESULT SPEC>
95  <FUNC END> ::= END FUNCTION <OPT IDENT> ;
96  <OPER HDR> ::= OVERLOAD OPERATOR <OPER NAME> <PARMS> <RESULT SPEC>
97  <OPER END> ::= END OPERATOR <OPT OPER NAME> ;
98  <OPT OPER NAME> ::=
99    | <OPER NAME>
100 <OPER NAME> ::= <OP1>
101   | <OP2>
102   | <OP3>
103   | <OP4>
104   | <OP5>
105   | <OP6>
106   | <OP7>

```

```

107 <OPT PARMs> ::=*
108   | <PARMS>
109 <PARMS> ::= ( <PARM LIST> )
110 <PARM LIST> ::= <PARM>
111   | <PARM LIST> , <PARM>
112 <PARM> ::= <BIND> <IDENT LIST> : <TYPE REF>
113   | RESULT <IDENT LIST> : <TYPE REF> <PARM INIT>
114 <BIND> ::=*
115   | <V OR C>
116 <PARM INIT> ::= INIT <?EXP>
117 <RESULT SPEC> ::= RESULT <IDENT> : <TYPE REF> ;
118 <OPT MONITOR> ::=*
119   | MONITOR
120 <OPT OVERLOAD> ::=*
121   | OVERLOAD
122 <CT PROC HDR> ::= %PROCEDURE <IDENT> <OPT %DCL PARMs> <OPT CT PARMs> ;
123 <CT PROC END> ::= %END %PROCEDURE <OPT IDENT> ;
124 <OPT %DCL PARMs> ::=*
125   | %DECLARE <IDENT LIST> :
126 <OPT CT PARMs> ::=*
127   .           | ( <IDENT LIST> )
128 <GTYPE REF> ::= <TYPE REF>
129   | GROUPED <TYPE REF>
130 <TYPE REF> ::= <IDENT>
131   | <IDENT> ( <TYPE ARGs> )
132 <TYPE ARGs> ::= <TYPE ARG>
133   | <TYPE ARGs> , <TYPE ARG>
134 <TYPE ARG> ::= *
135   | <EXP>
136   | <RANGE EXP>
137 <GEN TYPE> ::= ENUM ( <ENUM LIST> )
138   | ORDERED ENUM ( <ENUM LIST> )
139   | <CCMP TYPE>
140   | POINTER ( <V OR C> <TYPE REF> )
141   | MRECORD <MFIELDS> END MRECORD <OPT IDENT>
142   | FILE <FILE PROPS> OF <GTYPE REF>
143 <ENUM LIST> ::= <ENUM>
144   | <ENUM LIST> , <ENUM>
145 <ENUM> ::= <IDENT>
146   | <SP ENUM SYMB>
147 <MFIELDS> ::= <MFIELD>
148   | <MFIELDS> <MFIELD>
149 <MFIELD> ::= <IDENT> : <TYPE REF> ; <MD DIRECTS>
150 <MD DIRECTS> ::= <MD DIRECT>
151   | <MD DIRECTS> <MD DIRECT>
152 <MD DIRECT> ::= ! OFFSET <OFFSET BODY> ;
153   | ! SIZE <SIZE BODY> ;
154 <OFFSET BODY> ::= <OFFSET>
155   | <OFFSET BODY> , <OFFSET>
156 <OFFSET> ::= <IDENT> ( <EXP> )
157 <SIZE BODY> ::= BITS ( <EXP> )

```

```

158  <FILE PROPS> ::=*
159    | ( <IDENT LIST> )

160  <COMP TYPE> ::= ARRAY <DIMS> OF <GTYPE REF>
161    | RECORD <RFIELDS> END RECORD <OPT IDENT>
162    | UNION <UFIELDS> END UNION <OPT IDENT>

163  <DIMS> ::= ( <DIM LIST> )

164  <DIM LIST> ::= <DIM>
165    | <DIM LIST> , <DIM>

166  <DIM> ::= <IDENT>
167    | <RANGE EXP>

168  <RFIELDS> ::= <RFIELD>
169    | <RFIELDS> <RFIELD>

170  <RFIELD> ::= <IDENT LIST> : <GTYPE REF> ;

171  <UFIELDS> ::= <UFIELD>
172    | <UFIELDS> <UFIELD>

173  <UFIELD> ::= <IDENT LIST> : <TYPE REF> ;

174  <EXP> ::= <EXP1>
175    | <EXP> <OP1> <EXP1>

176  <OP1> ::= !
177    | ( R
178    | XCR

179  <EXP1> ::= <EXP2>
180    | <EXP1> <OP2> <EXP2>

181  <OP2> ::= &
182    | AND

183  <EXP2> ::= <EXP3>
184    | <OP3> <EXP3>

185  <OP3> ::= NOT

186  <EXP3> ::= <EXP4>
187    | <EXP4> <OP4> <EXP4>

188  <OP4> ::= =
189    | !=
190    | <
191    | <=
192    | >
193    | >=

194  <EXP4> ::= <EXP5>
195    | <OP5> <EXP5>
196    | <EXP4> <OP5> <EXP5>

197  <OP5> ::= +
198    | -

199  <EXP5> ::= <EXP6>
200    | <EXP5> <OP6> <EXP6>

201  <OP6> ::= *
202    | /
203    | CAT
204    | DIV
205    | MOD

206  <EXP6> ::= <PRIMARY>
207    | <EXP6> <OP7> <PRIMARY>

208  <OP7> ::= ++

```

```

209  <PRIMARY> ::= <IDENT>
210  | <FIXED>
211  | <FLOAT>
212  | <STRING>
213  | <BIT>
214  | <SP ENUM SYMB>
215  | <PRIMARY> ( <SLICE> )
216  | <PRIMARY> . <IDENT>
217  | <PRIMARY> & <IDENT>
218  | <PRIMARY> ^
219  | <PRIMARY> ( <FIELD INIT> )
220  | <PRIMARY> ( <GEN EXP LIST> )
221  | ( <EXP> )

222  <SLICE> ::= <EXP> AT <EXP>

223  <FIELD INIT> ::= <FIELD INIT>
224  | <FIELD INIT> , <FIELD INIT>

225  <FIELD INIT> ::= <GEN EXP LIST> : <?EXP>

226  <GEN EXP LIST> ::= <GEN EXP>
227  | <GEN EXP LIST> , <GEN EXP>

228  <GEN EXP> ::= <ACON TERM>
229  | *
230  | <RANGE EXP>

231  <ACON TERMS> ::= <ACON TERM>
232  | <ACON TERMS> , <ACON TERM>

233  <ACON TERM> ::= <?EXP>
234  | <EXP> OF <?EXP>
235  | [ <ACON TERMS> ]
236  | <EXP> OF [ <ACON TERMS> ]

237  <EXP LIST> ::= <EXP>
238  | <EXP LIST> , <EXP>

239  <RANGE EXP> ::= <EXP> .. <EXP>

240  <?EXP> ::= ?
241  | <EXP>

242  <LAB STM> ::= <STM>
243  | <IDENT> : <STM>

244  <STM> ::= ;
245  | <PRIMARY> := <EXP> ;
246  | BEGIN <BODY> END <OPT IDENT> ;
247  | IF <IF BODY> <IF END>
248  | <SELECT HDR> <SELECT BODY> <SELECT END>
249  | WHILE <EXP> DO <ECODY> <WHILE END>
250  | REPEAT <BODY> UNTIL <EXP> <REPEAT END>
251  | <FOR HDR> <ECODY> <FOR END>
252  | CALL <IDENT> <OPT ARGS> ;
253  | EXIT <IDENT> ;
254  | GOTO <IDENT> ;
255  | ASSERT <EXP> ;
256  | RAISE <IDENT> ;
257  | RAISE <IDENT> IN <IDENT> ;
258  | JN <IDENT LIST> <ON BODY> <ON END>
259  | FORK <FORK BODY> <FCRK END>
260  | SEND <IDENT> ;
261  | SEND <IDENT> RELEASE <IDENT> ;
262  | WAIT <IDENT> ;
263  | WAIT <IDENT> RELEASE <IDENT> ;
264  | PAUSE <EXP> SECONDS ;
265  | PAUSE <EXP> MILLISECONDS ;
266  | PAUSE <EXP> TICKS ;

267  <IF BODY> ::= <THEN PART>
268  | <THEN PART> ELSE <BODY>
269  | <THEN PART> ELSEIF <IF BODY>

270  <THEN PART> ::= <EXP> THEN <BODY>

271  <IF END> ::= END IF <OPT IDENT> ;

```

```

272  <SELECT HDR> ::= SELECT <EXP> FROM
273  <SELECT BODY> ::= <CASE ALTS>
274    | <CASE ALTS> OTHERWISE => <BODY>
275  <CASE ALTS> ::= <CASE ALT>
276    | <CASE ALTS> <CASE ALT>
277  <CASE ALT> ::= CASE <CASE SPECS> => <BODY>
278  <CASE SPECS> ::= <CASE SPEC>
279    | <CASE SPECS> , <CASE SPEC>
280  <CASE SPEC> ::= <EXP>
281    | <RANGE EXP>
282  <SELECT END> ::= END SELECT <OPT IDENT> ;
283  <WHILE END> ::= END WHILE <OPT IDENT> ;
284  <REPEAT END> ::= END REPEAT <OPT IDENT> ;
285  <FOR HDR> ::= FOR <FOR RANGE> DO
286    | FOR <IDENT> FROM <FOR RANGE> DO
287  <FOR RANGE> ::= <IDENT>
288    | <RANGE EXP>
289    | <EXP> DOWNTO <EXP>
290    | <EXP> UPTO <EXP>
291  <FOR END> ::= END FOR <OPT IDENT> ;
292  <OPT ARGS> ::=
293    | ( <EXP LIST> )
294  <ON BODY> ::= IN <BODY> DO <BODY>
295    | IN <BODY> DO SYSTEM
296  <ON END> ::= END ON <OPT IDENT> ;
297  <FORK BODY> ::= <BODY> <PATHS>
298  <PATHS> ::= <PATH>
299    | <PATHS> <PATH>
300  <PATH> ::= PATH <OPT IDENT> ; <BODY> <PATH END>
301  <PATH END> ::= END PATH <OPT IDENT> ;
302  <FORK END> ::= END FORK <OPT IDENT> ;
303  <OPT IDENT> ::=
304    | <IDENT>
305  <IDENT LIST> ::= <IDENT>
306    | <IDENT LIST> , <IDENT>
307  <OPT STRING> ::=
308    | <STRING>
309  <IDENT-STRING LIST> ::= <IDENT-STRING>
310    | <IDENT-STRING LIST> , <IDENT-STRING>
311  <IDENT-STRING> ::= <IDENT>
312    | <STRING>
313  <NAME> ::= <IDENT>
314    | <OPER NAME>
315  <NAME LIST> ::= <NAME>
316    | <NAME LIST> , <NAME>

```

INDEX

* operator 3-4, 5-20, 5-25, 8-3, 8-4
** operator 3-4, 5-25, 8-3
*> symbol 3-11
! operator 3-4, 5-14, 8-3, 8-4, 8-5
operator 3-4, 5-14, 5-16, 5-21, 5-26, 5-29, 5-48, 5-55, 5-56,
 8-3, 8-4
%CALL 3-10
%CONST 3-10
%DECLARE 3-10
%DECLARE parameters 14-13
%ELSE 3-10
%ELSEIF 3-10
%END 3-10
%IF 3-10
%PROCEDURE 3-10
%THEN 3-10
& operator 3-4, 5-14, 8-3, 8-4, 8-5
+ operator 3-4, 5-20, 5-25, 8-3, 8-4
- operator 3-4, 5-20, 5-25, 8-3, 8-4
/ operator 3-4, 5-20, 5-25, 8-3
< operator 3-4, 5-16, 5-21, 5-26, 5-29, 8-3
<* symbol 3-11
<= operator 3-4, 5-16, 5-21, 5-26, 5-29, 8-3
= operator 3-4, 5-14, 5-16, 5-21, 5-26, 5-29, 5-48, 5-55, 5-56,
 8-3, 8-4
> operator 3-4, 5-16, 5-21, 5-26, 5-29, 8-3
>= operator 3-4, 5-16, 5-21, 5-26, 5-29, 8-3
absolute address 13-7, 13-8
ACCESS directive 2-5, 14-1, 14-2
actual parameter 7-4
addition 8-3
ADDRESS type 13-7
aggregate type 8-11
ALGOL 60 4-20, 8-1

aliasing 2-3, 4-13, 7-2, 7-14
dangerous 7-14

ALIGN directive 4-22, 13-9

alignment 13-7, 13-9

AND operator 3-4, 5-13, 5-56, 8-3, 8-4, 8-5

ANY formal parameter 10-10, 14-17

arguments 7-5

arithmetic types 5-6

ARRAY 3-5, 5-6, 5-30, 5-33, 10-3
catenation 5-33
construction 5-35
parameters 5-50
slicing 5-33
type generator 5-30, 10-3

ASCII 3-1, Appendix B

assembly language 2-4, 7-12, 13-12

ASSERT 3-5
statement 9-2, 9-23, 14-18

assignment 5-3, 5-7
statement 9-3

ASSOCIATIVE directive 7-6, 7-22

associativity 8-2, 8-4

AT 3-5

attribute 2-2, 5-1, 5-51, 5-59

BEGIN 3-5
statement 4-17, 9-2, 9-5

binary literal 5-56

binding
class 2-2, 7-4, 7-18, 13-15
strength 8-2

BITS 3-9, 5-6, 5-50, 5-56
literal 3-9, 5-56

block-structure 2-1

BNF syntax 1-6

BOOLEAN 5-13, 5-56
literal 5-13
operator 8-1

bounds ranges 5-31

CALL 3-5
call by reference 7-5
call by value 7-5
statement 7-3, 9-20
see compile-time call command

CALLER_ASSERT directive 10-10, 14-17, 14-18

CAPSULE 2-2, 3-5, 4-12, 6-1, 6-2, 12-11
declaration 4-17, 6-1
element 6-1
see MONITOR

CASE 3-5
alternative 4-17, 9-10, 9-11, 9-12
label 9-10, 9-11, 9-12

CAT operator 3-4, 5-31, 5-33, 8-3, 8-4

catenation 5-31, 5-33, 8-1, 8-3
see CAT operator

CHANGE_PRIORITY procedure 12-8

CHAR 5-15, 5-55
literal 5-15

character set 3-1, 3-6, 5-28

CLOSE procedure 10-4

closed scope 2-2, 4-11, 4-15, 4-17, 6-1

comment 3-1, 3-11
contents 3-11

COMMUTATIVE directive 7-6, 7-21
commutativity 8-2, 8-4

compile-time actual parameter 4-9, 14-15

compile-time call command 14-12

compile-time command 4-7, 12-3

compile-time constant 2-5, 14-6

compile-time expression 14-6, 14-8

compile-time facilities 2-3, 2-5, 10-1, 14-1, 14-10

compile-time identifier 8-10

compile-time if command 14-9, 14-15

compile-time procedure 4-7, 4-9, 4-17, 14-11

compile-time symbol 3-10

compile-time type interrogation 10-10

complement 8-3
component
 selection 5-32
 type 5-31, 5-38, 10-3
compools 14-1
conditional compilation 2-5, 4-7, 14-9
conditional statement 9-6
configuration constant 14-7
CONFIGURATION directive 2-4, 13-3, 14-20
conjunction 8-1, 8-3
CONNECT directive 13-10
CONST 3-5, 7-4
constant 5-8, 14-6
 value 3-6
 see compile-time constant
control transfer statements 9-19
conversion 10-9
copy in binding 7-5
copy out binding 7-5
CT 14-12
cumulative processing time 12-15
CUM_MILLISECONDS 12-15
CUM_SECONDS 12-15
CUM_TICKS 12-15
dangerous aliasing 7-14
data abstraction 6-1
 see CAPSULE
data element 4-9
data object 4-9
data types 2-1, 5-1
deadlock 12-9, 12-10
declaration 2-1, 4-5, 4-6
default 4-6, 10-3, 10-8

dereference 5-46
device-dependent 10-1
directive 2-1, 4-3, 7-6
directly reachable 7-8
disjunction 8-1
display 7-9
DIV operator 3-4, 5-20, 5-21, 8-3
division 5-21, 8-3
DO 3-5
DOWNTO 3-5
dynamic 5-5, 5-45
 constant 5-8
 variable 5-8
EBCDIC 3-1
element 4-9
ELSE 3-5
 body 4-17
 clause 9-8
ELSEIF 3-5
 body 4-17
 ELSEIF-THEN clause 9-8
embedded applications 13-2
empty statement 9-3
encapsulation 2-2
 see CAPSULE
END 3-5
ENUM 3-5
 enumeration 5-27
 type 3-6, 5-28
EOF function 10-5
EOLN function 10-8
equality 8-3
EVENT 2-4, 3-5, 12-1, 12-5, 12-7, 12-11, 12-14
EXCEPTION 2-4, 3-5, 9-10, 11-1, 12-4
 declaration 11-2
 statement 11-3
exclusive disjunction 8-3
execution time 12-15

EXIT 3-5
 statement 4-8, 7-3, 9-2, 9-21, 12-4
exponentiation 5-25, 8-1, 8-3
EXPORT 2-2
 directive 4-22, 6-1, 6-3
expression 2-2, 8-1, 8-2, 8-4
external routines 13-14
FDIV operator 5-21
field 5-38
 identifier 5-38, 5-39
FILE 3-5, 10-2, 10-3
 parameter 10-3
 type 10-4
 type descriptor 10-1
 type generator 2-3, 10-3
 FILE-RECORD 10-3, 10-6
FILE_COMPATIBLE 14-19
FILE_REP 10-4
FILE_TYPE 10-10
FIXED 3-7, 5-6, 5-17, 5-18, 5-20, 5-29
 division 5-21
 literal 5-19
FLOAT 3-7, 5-6, 5-24, 5-25
 literal 5-19, 5-25
FOR 3-5
 statement 4-17, 9-16
 for variable 4-6, 9-17
FORK 3-5
 statement 2-4, 4-17, 12-1, 12-2, 12-3, 12-4, 12-5, 12-6
 fork-wait 11-6, 12-4, 12-5
formal parameter 4-6, 5-7, 5-50, 7-4, 7-22, 14-17
 list 7-4
 pack 7-4
formatted I/O 10-8
forward references 4-22
FREE 5-47, 5-48
function 3-5, 7-17, 7-20
 invocation 7-3, 8-13
 result 7-2, 7-17
 result variable 4-6, 7-18, 7-22

generic parameters 14-12
GET_POSITION procedure 10-5
GOTO 3-5
 statement 4-8, 9-2, 9-22
greater-than 8-3
grouping attribute 5-59
guarded body 11-4
handler 11-4
hardware interrupt 13-10
hexidecimal literal 5-56
HIGH function 5-16, 5-22, 5-26, 5-29, 5-31
HIGH_ACTUAL 5-22
I/O 2-3, 11-1, 14-19
 formatted 10-8
 line-oriented 10-7
 sequential input 10-6
identifier 3-4
IF 3-5
 statement 4-17, 9-2, 9-7
 if-then clause 9-8
 see compile-time if command
immediate containment 4-11
implementation-dependent 14-20
IMPORT 2-1
 directive 4-12, 4-15, 4-16, 4-18, 6-1
IN 3-5
inequality 8-3
infix operator 8-1
inheritance 4-12, 4-15, 4-16, 4-18
INIT 3-5
initialization 4-6, 5-7, 5-9, 12-12
INLINE directive 7-6, 7-7, 13-12
Input/Output
 see I/O
input file 10-8
integer 5-17
interrupt 2-4, 13-10

invocation 7-3
see operator invocation

IS_ENUM 14-19

IS_FILE 14-19

IS_MRECORD 14-19

IS_POINTER 14-19

IS_RECORD 14-19

IS_UNION 14-19

iteration 9-17

label 4-6, 4-8, 4-22, 9-1, 9-21, 9-22
label identifier 4-9

LAST_EXCEPTION 9-10, 11-8

less-than 8-3

lexical analysis
lexical diagram 1-2, 1-6
lexical properties 1-6
lexical structure 3-1
lexical units 1-2
see token

libraries 14-1
library definition facilities 10-10

lifting 5-61

line 7-2
line separator 10-7
line terminator 3-11
line-oriented I/O 10-7

LINKAGE 13-10
directive 13-12, 13-15
linking 14-3

literal 3-6, 5-13, 5-15, 5-19, 5-25, 5-29, 5-55, 5-56, 8-10

LOC function 13-7

logical operators 5-56

loop statement 9-2, 9-13

LOW function 5-16, 5-22, 5-26, 5-29, 5-31

lower case 3-4

lowering 5-61

LOW_ACTUAL 5-22, 5-26

machine dependent facilities 2-4, 4-4, 7-12, 10-1, 13-1, 14-20
 machine-code 3-5
 machine dependent directive 7-6
 machine record 2-4, 13-4
 machine routine 7-12, 13-11

machine language 7-12, 13-12

macro facility 2-5, 4-7, 14-12

main directive 7-6, 7-16
 main routines 4-4

MAX_EXPONENT 5-24, 14-7

MAX_INTEGER 5-18, 14-7

MAX_PRECISION 5-24, 14-7

MAX_SENDERS 12-11, 14-7

MILLISECONDS 12-15

MIN_EXPONENT 5-24, 14-7

MIN_INTEGER 5-18, 14-7

MIN_PRIORITY 12-7, 14-7

MOD operator 3-4, 5-20, 5-21, 8-3

MONITOR 2-4, 3-5, 6-2, 12-1, 12-5, 12-6, 12-7, 12-9, 12-10, 12-12
 12-13, 12-14

MRECORD 3-5
 generator 10-3

multi-path facilities 12-1

multiplication 8-3

mutual exclusion 2-4, 12-1, 12-9, 14-13

name 3-3
 nameless objects 4-9
 namescopes 4-11, 4-21, 6-1
 reuse of names 4-20

NEW 5-5, 5-9, 5-47, 5-48

NIL 5-46

NIL_FILE 10-3

NOT operator 3-4, 5-13, 5-56, 8-3

numeric literal 3-7

object 4-9, 5-1
 component 8-10, 8-11
 modules 14-3
 nameless 4-9
 R-valued 5-9
 W-valued 5-9
octal literal 5-56
OF 3-5
OFF directive 11-7
OFFSET directive 13-6
ON 3-5

opaque types 2-1
OPEN procedure 10-1, 10-3, 10-4, 10-8
open scope 2-1, 4-15, 4-17
operand 8-1, 8-2
OPERATOR 3-5, 7-12, 7-19, 7-20, 8-1, 8-3
 infix 8-1
 invocation 7-3
 prefix 8-1
 symbol 2-3, 3-4
optimization 4-4, 7-2, 7-6, 7-10
 OPTIMIZE directive 4-23
OR operator 3-4, 5-13, 5-56, 8-3, 8-4, 8-5
order of evaluation 8-5
ORDERED 3-5, 5-28
 enumeration 5-28, 5-29
OTHERWISE 3-5
 alternative 4-17, 9-10, 9-11
output file 10-8
overload line 7-2
 overloading 2-2, 2-3, 3-5, 4-11, 4-18, 4-20, 7-2, 7-19, 7-20,
 7-21, 7-22, 14-2, 14-17
OVERWRITE procedure 10-6
parallel execution 12-2, 12-4
 parallel paths 2-4, 6-2, 11-6, 12-1

parameters 5-3, 13-12, 13-15
 ~~DECLARE~~ 14-13
 parameterized types 2-1, 5-7, 5-12, 5-30, 5-50
 actual 7-4
 ANY 10-10
 compile-time 4-9, 14-15
 formal 5-50, 7-4
 type-unresolved 14-17

parentheses 8-2

parse 8-2

Pascal 1-1, 2-3, 4-2, 4-20, 5-5, 5-6, 5-42, 7-2, 8-1, 9-2, 10-1,
 10-7

path 3-5, 12-2, 12-3, 12-9, 12-10, 12-12, 12-13, 12-14, 12-15
 clause 4-17, 12-3, 12-4
 identifier 4-6, 11-3, 12-4, 12-8
 states 12-5

PAUSE 3-5
 statement 2-4, 12-1, 12-5, 12-6, 12-15

PERVASIVE 2-1
 directive 4-12, 4-13, 4-16, 4-22, 12-11

PL/I 5-5

POINTER 3-5, 5-45, 5-46
 dereference 8-10
 type 10-4

pre-defined exception 11-1, 11-7

precedence 8-2, 8-3, 8-4

precision 5-25, 5-26

PRECISION_ACTUAL 5-26

PRED function 5-16, 5-29

prefix operator 8-1

primary 8-1, 8-9, 8-10

PRIORITY 12-7, 12-8, 12-12
 directive 12-7
 function 12-8

PROCEDURE 3-5, 7-11
 see compile-time procedure

processing time 12-15

PROGRAM 3-5, 4-1, 4-17, 12-2
element 4-6, 4-9
level 4-12, 13-3, 13-7, 13-8, 14-2
structure 2-1, 4-1, 4-2

programming examples 1-6

PROMOTE directive 2-5, 4-22, 14-1

qualification 5-39, 5-51

R-value 4-10, 5-9, 8-10, 8-13

radix 5-24, 14-7

RAISE 3-5
statement 11-1, 11-3, 11-7

RANGE 3-5, 5-22, 5-31, 5-58
bounds ranges 5-31

reachability 7-8

READ procedure 10-5

READLN 10-7

readonly 12-11
READONLY directive 4-12, 4-19, 7-13

real-time 2-4
facilities 12-1, 12-15

recompilation 14-3

RECORD 3-5, 5-6, 5-38
component selection 5-39
construction 5-39
type generator 5-38, 13-6
variant records 5-42

recursion 4-22, 7-9
RECURSIVE directive 7-6, 7-9

redeclaration 4-11, 4-20
REDECLARE directive 10-10, 14-17, 14-20

REENTRANT directive 7-6, 7-10

reference counts 5-49

relational operators 5-26, 5-29, 5-48, 8-1

RELEASE clause 12-9, 12-10, 12-12, 12-13

REPEAT 3-5
statement 4-17, 9-15

representation 2-2, 5-1

resolved type references 5-53

RESULT 3-5, 7-4
parameter 9-21
variable 7-4

RETURN 7-3

reuse of names 4-20

REWIND procedure 10-5

ROUND 5-19, 5-22

routine declaration 4-17, 7-1, 13-15
routine directives 7-6
routines 2-1, 2-2, 4-22, 7-1

SAFE directive 7-6, 7-12, 7-18
safe procedure 7-13

scalar types 5-13

scale 5-22
conversion 5-22, 5-23

scope 2-1, 4-1, 4-11
see closed scope, open scope

SECONDS 12-15

SELECT 3-5
statement 4-17, 5-42, 9-2, 9-9, 9-11, 11-8

SEND 3-5
statement 12-6, 12-9, 12-10, 12-11, 12-12, 12-13, 12-14

SENDERS function 12-11, 12-14

separate compilation 2-5, 4-1, 14-1, 14-3

sequential input 10-6

set membership 8-1

SET_POSITION procedure 10-6

shared data 2-4, 6-2, 12-1, 12-9

short-circuiting 5-14, 7-5, 7-20, 7-22, 8-1, 8-5
short-circuit connunction 8-3
short-circuit disjunction 8-3

side effects 2-2, 4-13, 7-2, 7-12, 7-17, 7-18

simple type declaration 5-12, 5-30

SIZE function 5-31

SIZE formal parameter 5-55

SIZE directive 13-6

special enumeration symbol 3-6, 5-28
special symbol 3-10
state transitions 12-5
statement 2-1, 2-3, 4-8, 9-1, 9-2
static 5-5
 constant 5-8
 variable 5-8
status parameter 2-3, 10-1
status value 10-4
storage classes 5-5
 storage layout 2-4, 13-5
STORE 13-7
 directive 4-22, 13-7, 13-8
STRING 3-8, 5-50, 5-55, 10-8
 literal 3-1, 3-8, 5-55
structural description 1-2
subscripting 5-32
subtraction 8-3
SUCC function 5-16, 5-29
symbol
 compile-time 3-10
 operator 2-3, 3-4
 special 3-10
 special enumeration 3-6
synchronization 2-4, 12-1, 12-11
syntactic categories 1-2
syntactic trigger 3-4, 3-5
syntax
 diagrams 1-4, 1-6, Appendix C
 BNF 1-6, Appendix D
SYSTEM 3-5, 11-5
tag field 9-10, 9-11, 9-12
templates 14-4
textfile 10-7, 10-8
 textfile I/O 10-7
THEN 3-5
 body 4-17

TICKS 12-15
TICKS_PER_SECOND 12-15
token 1-4, 3-2, 3-3
 token separator 1-5, 3-2, 3-11
 see lexical analysis
top-level procedure 7-16
TRUNCATE 5-19, 5-22
TYPE 3-5, 4-22, 5-1, 5-10
 ADDRESS 13-7
 aggregate 8-11
 arithmetic 5-6
 comparison 14-19
 compile-time interrogation 10-10
 component 5-31, 10-3
 conversions 5-23, 5-26
 declaration 5-12
 generators 5-27
 identity 2-1, 5-12
 naming 5-4, 5-6
 opacity 2-1, 5-61
 reference 5-28
 resolved references 5-53
 type-unresolved parameters 14-17
 unresolved references 5-52
 user-defined 7-2
TYPE_EQ 14-19
underscore 3-4
UNION 3-5, 5-6, 5-41
 component selection 5-43
 construction 5-43
 generator 10-3
 object 2-3
 select 9-11
 type 9-11
unordered enumeration 5-28
unresolved parameters 2-5
 unresolved type references 5-52, 5-53
UNTIL 3-5
UPTO 3-5
user-defined exception 11-1
 user-defined types 7-2

USING directive 13-12, 13-15
value 4-10
 value select 9-10
VAR binding class 3-5, 5-59, 7-4
variable 5-8
variant records 5-42
W-value 4-10, 5-9, 8-10
WAIT 3-5
 statement 12-5, 12-6, 12-9, 12-11, 12-13, 12-14
WAITERS function 12-11, 12-14
WHILE 3-5
 statement 4-17, 9-14
WRITE procedure 10-5
WRITELN procedure 10-7
XOR operator 3-4, 5-13, 5-56, 8-3, 8-4
X_ALIAS exception 7-15, 11-1
X_ANY exception 11-4
X_ASSERT exception 11-1
X_FILE exception 11-1
X_INIT exception 5-9, 11-1
X_MULTIPATH exception 11-1, 11-6
X NIL POINTER 11-1
X_NON_FREEABLE exception 5-49, 11-1
X_OVERFLOW exception 5-21, 5-23, 11-1
X_RANGE exception 5-4, 5-23, 5-29, 5-31, 5-58, 11-1
X_SELECT exception 9-11, 11-1
X_SIZE exception 11-1, 14-20
X_SUBSCRIPT EXCEPTION 5-32, 5-33, 11-1
X_TAG exception 5-43, 9-11, 11-1
X_TERMINATE exception 2-4, 11-1, 11-3, 11-6, 12-1, 12-4, 12-5,
 12-6, 12-14
X_UNDERFLOW exception 5-23, 11-1
X_ZERO_DIVIDE exception 5-21, 11-1