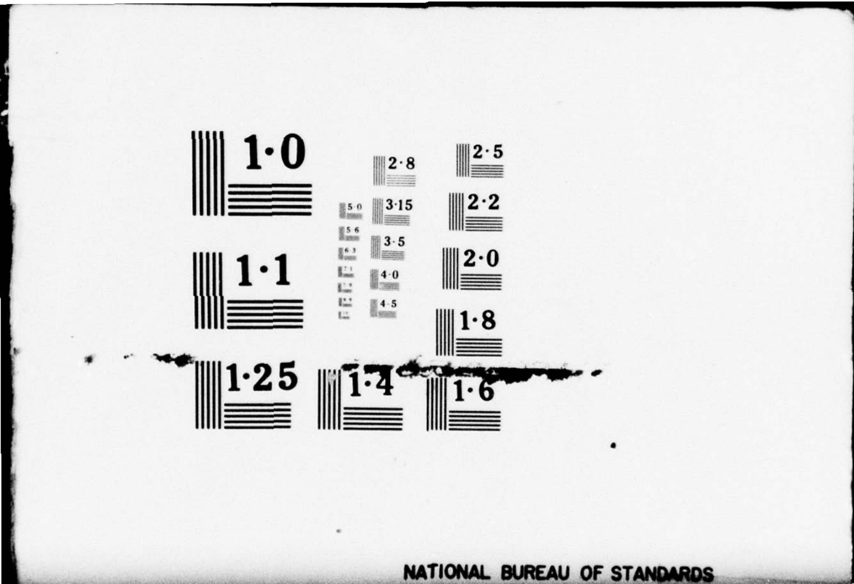
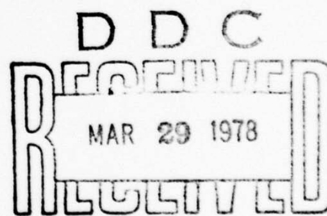
AD-A052 082 IBM UNITED KINGDOM LABS LTD WINCHESTER (ENGLAND) F/G 9/2
TOWARDS A PL/I-BASED 'IRONMAN' LANGUAGE.(U)
DEC 77 R F MADDOCK, B L MARKS
UNCLASSIFIED TR-12-168 NL

1 OF 2
ADA
052082


1·0
1·1
1·25
1·4
1·6
1·8
2·0
2·2
2·5
2·8
3·15
3·5
4·0
4·5
NATIONAL BUREAU OF STANDARDS

Towards a PL/I-Based 'Ironman' Language

Robert F Maddock
Brian L Marks



December 1977

Technical Report TR.12.168

Technical Report TR.12.168

Towards a PL/I-Based 'Ironman' Language

Robert F. Maddock
Brian L. Marks

Unrestricted

December 1977

DDC
RECEIVED
MAR 29 1978
D

IBM United Kingdom Laboratories Limited
Hursley Park
Winchester Hampshire

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

PREFACE

The Department of Defense in the United States is running a project, known as IRONMAN, to establish a new programming language for computer applications embedded in other hardware. Their evaluations of existing languages determined that only PASCAL, ALGOL68 and PL/I were suitable as bases for the new language. This report describes work that was begun on such a PL/I-based language. The work was terminated before completion since the Department of Defense have recently chosen to pursue only PASCAL-based solutions.

This report contains a technical discussion leading to a PL/I-based language to meet the 'Ironman' requirements, and a user manual for this language. Although it is incomplete we believe it will be of some interest since there is enough to establish the potential of the approach. The work did not reach the stage of being amended by the input of IBM specialists in such fields as proof of correctness, and no implementation has been attempted.

ACCESSION for
NTIS White Section ☒
DDC Buff Section ☐
UNANNOUNCED ☐
JUSTIFICATION
Per: H.R. Proctor DDC/TCA-2
BY
DISTRIBUTION/AVAILABILITY CODES
Dist. AVAIL. and/or SPECIAL
A

PRECEDING PAGE BLANK-NOT FILMED

TABLE OF CONTENTS

1 TECHNICAL DISCUSSION

1.1 Introduction
1.2 Relation to PL/I
1.3 Dynamism
1.4 Character Sets and Reserved Words
1.5 Data Types
1.6 Aggregates
1.7 References
1.8 Allocation and Assignment
1.9 Operations
1.10 Program Structure
1.11 Modules
1.12 Exception Handling
1.13 Input/Output
1.14 Dynamic Storage
1.15 Parallel Processes
1.16 Machine Dependent Language
1.17 Syntax

2 USERS MANUAL

2.1 Introduction
2.2 The Physical Components of a Program
2.3 Delimiters and Non-delimiters
2.4 Identifiers and Constants
2.5 Objects, Including those that contain others.
2.5.1 Structure and Member
2.5.2 Arrays
2.5.3 Unions
2.5.4 SHARED
2.5.5 Augmented Arrays
2.6 Arrays of Structures Etc.
2.7 Attributes Acquired Indirectly
2.8 Attributes
2.9 The Builtin Attributes
2.10 Resolution and Modules
2.10.1 Generic Selection
2.10.2 Indirect Resolution
2.11 Referencing
2.12 Function Operations
2.13 Builtin Operations
2.14 Ordering Within Expressions
2.15 Assignment of Objects With Components
2.16 Assignment of Single Objects
2.17 Order of Execution of Statements
2.18 Conditions
2.19 Subroutine Operations

2.20 Allocation and Freeing
2.21 Parameters
2.22 Input/Output
2.23 Starting and Synchronizing
2.24 Invalid Programs
2.25 System Modules
2.26 Shorthands
2.27 Editing, Translating, and Executing programs
2.28 Messages
2.29 Machine Description
2.30 Program Interchange
2.31 Parameters to Builtin Programs
2.32 Builtin Functions
2.33 Syntax

EXAMPLES

3.1 Simple PL/I
3.2 Complex Arithmetic
3.3 Producer/Consumer
3.4 LTPL-E/297

Appendix - Intersection with PL/I
References

CHAPTER 1. TECHNICAL DISCUSSION

1.1 INTRODUCTION

The US Department of Defense are running a High Order Language project. The aim is to develop a common language for use in embedded computers, i.e. computers supplied as part of a larger piece of equipment such as a radar system. Starting in 1975, they have issued a series of requirements documents codenamed "Strawman", "Woodenman", "Tinman" and "Ironman" (ref 1). Because of the intended application of the language, the requirements point to a language which could be called a systems programming language or a real-time programming language. After evaluating many languages, both existing and under development, the DOD decided that the planned language should be based on PASCAL, ALGOL68 or PL/I. However when the four initial contracts were awarded in July 1977, they were all to bidders proposing a PASCAL-based language.

The authors started work on a PL/I-based solution during the evaluation and since the DOD have chosen not to pursue a PL/I-based language, we do not plan to continue our work. But we decided to publish it as it stands today in the hope that it may be of use to others working in this field. This report is therefore of unfinished work, and undoubtedly contains many mistakes, inconsistencies and omissions. To rectify these would be to complete the language design work. We have no plans currently to continue this work: either to complete the language design or to develop a compiler.

The report describes in outline form a design for a programming language to meet the requirements in "Ironman" (ref 1) using ANSI Standard PL/I (ref 2) as a base. We first took only those parts of PL/I which met requirements in "Ironman", and then extended this language to meet the remaining requirements. An important part of the subsetting process was to increase the opportunities for compile-time checking of the program by excluding, for instance, implicit conversions. This was also helped by some of the extensions; constrained pointers for example. But the two main areas of extension were (i) to allow the programmer to define new data types and operations, and (ii) extensions to parallel programming.

Most of the ideas we have used are not new; we have tried to learn from previous work as much as possible. Of particular influence we could mention CLU (ref 3), MODULA (ref 7), ILIAD (ref 4) and ALGOL68. We have also benefited from discussions with colleagues both inside IBM and outside, although the formal reviews by IBM specialists in particular fields that were originally planned have not taken place.

1.2 RELATION TO PL/I

It is unlikely that any existing PL/I programs will fortuitously be PL/I Based Ironman (PBI) programs. The discipline we have proposed for the positioning of declare statements etc. will prevent this (see appendix). One could write PBI programs that were acceptable to a PL/I compiler but major parts of PBI could not be used and the compiler would not make some checks. Hence we are not talking about a zero-cost interchange of PL/I and PBI programs. Mechanical transport of PL/I to PBI would be fairly simple by adapting a PL/I compiler to emit source rather than object code. A translator for the other direction would be somewhat harder.

A principle reason for compatibility with PL/I is to benefit from the base of experience that the ANSI Standard represents. This definition has been rigorously examined from a number of perspectives over recent years. As far as possible we have departed from it only by subsetting.

Programmers moving from PL/I to PBI would not have major hurdles to cross. They would:

- need to adopt the disciplines of explicit declaration and conversion.
- need to work within the subset,e.g. BASED only in AREAs
- eventually learn to benefit from MODULES etc.

1.3 DYNAMISM

There is a general problem with languages in deciding how dynamic things should be. Should the size of an array be changeable at every reference (as with APL), at procedure entry (as with PL/I), or should it be fixed at compile time (as with PASCAL)?

Our approach has been to provide the most dynamic features that are realistically consistent with making the best object code. For example we would expect the compiler to map storage completely at compile time (and tell the user the requirements) if the objects of a particular program were all of fixed size (and there was no recursion etc.) Otherwise the compiler would provide basic information (such as space required by particular procedures) and would make efficient code for dynamic allocation but would not be able to guarantee a maximum storage requirement.

PRIORITY is a difficulty. If the priorities are compile time constants then the implementation can be very efficient when the hardware has priority levels. Dynamic priorities mean significant overheads irrespective of the hardware. It seems better to restrict than to support multiple mechanisms in this case.

1.4 CHARACTER SETS AND RESERVED WORDS

The syntax of the language is based on that of PL/I. The character set used is that common to EBCDIC and the 64-character subset of ASCII, i.e.

AtoZ 0to9 "&'()*+,-./:;<>=?_¬|
(note ¬ and | are alternatives for ^ and ! in ASCII)

%$@# are not used; the latter three are used as alphabetic extenders in some countries.

If [] are available they can be used as in mathematics, i.e. they mean the same as () but like must match like.

Identifiers must be strings of letters, numbers and underscore (_) beginning with a letter. Some keywords in the language are reserved. These are the ones which determine the structure of the program, or ones where the syntax scan (both for a compiler or a human reader) is made more difficult by use of the identifier for other things. The reserved identifiers are: BASED, CASE, DECLARE, DO, END, IF, INITIAL, MODULE, PRIORITY, PROCEDURE, RETURNS, SELECT, SYSTEM, THEN, UNSPEC, VALUES, WHEN, WHILE.

Comments may appear anywhere a blank may appear, and are written thus: /* comment */

1.5 DATA TYPES

Declare statement

The declare statement is used to introduce the names of variables and constants.

```
    DCL { [level-no] { identifier | (identifier-commalist) }
          attributes } ,...;
```

The attributes must include one datatype attribute (except for structures).

Datatype attributes

```
BIT
CHARACTER
FIXED DECIMAL (p [,q])
FLOAT DECIMAL (p)
POINTER TO (type) IN (area-ref)
AREA (area-description)
FILE [RECORD [KEYED] | PRINT]
CONDITION
VALUES (value-constant-commalist)
declared-type [(extent-expr-commalist)]
```

The DECIMAL precision(p) and scale(q) specify the allowed range and interval of the values which an arithmetic data element may be given.

[A possible alternative to DECIMAL(p) would be a RANGE attribute which would be able to specify the allowed values more exactly e.g. with maximum other than 10**n-1, or always non-negative. It avoids the problem of choosing BINARY or DECIMAL too.]

VALUES allows the programmer to specify a list of names to represent the possible values e.g.

VALUES ("MONDAY", "TUESDAY", "WEDNESDAY" etc.)

[Another use of a RANGE attribute could be to specify a subrange of a previously defined VALUES type.]

The attributes BIT and CHARACTER may be regarded as predefined VALUES types with values of:

"0" and "1", and
"A", "B", "C" etc respectively.

Strings of bits or characters are introduced under the heading of arrays. The attributes FILE, CONDITION, POINTER and AREA are discussed later in separate sections.

Literal forms exist for the values of the computational types i.e. FIXED FLOAT and VALUES (including BIT and CHARACTER).

Examples

```
FIXED      :   23   0.625
FLOAT      :   9.62E5 and FIXED literals used in a FLOAT context,
                          e.g. FLT=FLT+1;
VALUES     )   the values specified in the VALUES attribute
BIT        )   for this type.
CHARACTER  )   e.g. "MONDAY", "0" (for BIT), "A" (for CHARACTER)
```

If the same literal value is specified for several VALUES types it may be used where only one of the types is valid. If more than one is valid, the ambiguity should be resolved by a conversion operation to the type intended, e.g. BIT ("1")

[If we did not want the literal values to be syntactically distinguishable,the double quotes could be omitted, and the values specified as normal identifiers. For single (non-letter) characters the single quote could perhaps be used as for strings, although this would not distinguish between a scalar and a string of length one. Later versions of the Ironman specification do not ask for syntactic distinction and this is reflected in some examples in this report.]

Additional attributes

The following attributes may also be specified:

INITIAL (expression)
: means that the object is set to the value of the expression at the beginning of its lifetime.

CONSTANT
: means that the value of the object is constant throughout its lifetime. It must therefore not be assigned to.

CONSTANT (expression)
: is a shorthand for CONSTANT INITIAL (expression).

Declared types

These are specified by a declaration which includes the keyword BASED in addition to a normal attribute specification for an object, e.g.

```
DCL COUNTER BASED FIXED DECIMAL(4);
DCL FIXCON  BASED FIXED DECIMAL(1) CONSTANT;
DCL WEEKDAY BASED VALUES ("MONDAY", "TUESDAY" etc)
```

The attribute specification must include a datatype attribute (except for structures) and may also include CONSTANT and INITIAL. The declared user types can then be used in further declarations of constants, variables or new types, e.g.

```
DCL ONE FIXCON INITIAL (1);
DCL (X,Y,Z) COUNTER;
DCL SPECIAL-COUNTER BASED COUNTER;
```

[Another possible syntax for the declaration of types would be a new statement, e.g.

TYPE MYTYPE BIT;

But using BASED preserves the part of PL/I which provides some type declaration function. The type can be considered "based" on the attributes used to define it.]

1.6 AGGREGATES

In addition to the datatypes already described, objects may be declared to be aggregates, either arrays or structures. Both are made up of components which can be selected and operated upon individually: for arrays the selector is the value of an expression or expressions (the subscripts) which are of a declared type, for structures the selector is one of a set of declared names.

Arrays

Arrays of one or more dimensions can be declared; the range of each subscript can be specified either by a VALUES type or as an integer range. For integer values, the lower bound is always 1 and is not specified. Arrays are declared thus:

```
(dimension-commalist) datatype

dimension::= extent-expression | usertype | BIT | CHARACTER
```

Examples

```
        DCL X (10,I+2) BIT;
        DCL Y (WEEKDAY) FIXED DECIMAL (2);
        DCL POINT BASED (3) FLOAT DECIMAL(6);
    elements can then be referenced thus:
        X(J,4)  Y("MONDAY")
```

Any integer in an attribute, such as the extent-expressions specifying the dimensions of an array are evaluated when the array is allocated at the beginning of its lifetime. For a declared variable or constant this is on entry to the block in which it is declared.

An extent-expression can involve components of the aggregate being declared, in which case these components must be constants initialised either in the declaration of the type, or in the declaration (or allocation) of the variable.

```
DECLARE 1 STRUCT BASED,
          2 N FIXED DECIMAL(2) CONSTANT,
          2 ARRAY(N) BIT;

DECLARE X STRUCT INITIAL(N:50);
```

A declared type or parameter may be specified with an asterisk in place of an extent-expression, e.g.

```
DCL VECTOR BASED (*) FLOAT;
DCL PARAM CHARACTER (*);
```

This means that the object contains a hidden field which will contain at run-time the value of this extent. This value is set at allocation time and is then constant for the lifetime of the object.

For declared types the value is specified in the declaration. The values are associated with the asterisks in order.

DCL V VECTOR(10)allocates a (10)FLOAT plus the hidden field with value 10.

For parameters it is taken from the argument.

[A note on precision

The precision and scale specifications, "p" and "q" in the datatype attribute list are integer values.

So they could be allowed to be expressions calculated at procedure entry. But this would in practice be of limited use and certainly make separate compilation of subroutines more difficult , and should perhaps be disallowed. Expressions which can be evaluated at compile time are of course relatively straightforward.

They could also be asterisks. This is useful for type specification, e.g.

```
DCL INTEGER BASED FIXED DECIMAL(*,0)
DCL (A,B,C) INTEGER(5);
```

But it is probably sufficient, and much easier to implement, if these asterisks can be resolved at compile time. In which case they are really a form of compile time parameter. We could use ? rather than * for this form:

```
DCL INTEGER BASED FIXED DECIMAL(?,0);
```

Note however that since assignment is defined between different precisions, a routine can be written to handle arguments of various precisions by making the parameter precision the maximum required.

On the other hand this means a compiler can compile code to handle ? precision parameters by using the maximum precision of the target machine, or the maximum used by calls to the procedure.

[It might be useful to allow repeated use in a type definition of a single element, e.g.

```
DCL MATRIX BASED (?1,?1) FLOAT;
DCL M MATRIX(3);
```

It might also be useful for function definitions:

```
SIN:PROCEDURE(X:FLOAT DECIMAL(?1))RETURNS(FLOAT DECIMAL(?1));]
```

[These numbered question marks are really nothing but identifiers, and the next stage would be to use identifiers, and allow them to take values other than integers, for example types (ref 5).

It would be a useful extension to allow types as parameters to other types, e.g.

```
DCL PAIR(T) BASED (2) T;
DCL X PAIR(FLOAT DECIMAL(6));
```

This would allow the definition of new aggregates such as STACK or QUEUE. However the implications of this for the definition of operations is complex, and needs further study.]

Array constructor

Array values can be constructed from element-values; the notation is a parenthesised list, each element specifying a subscript and the value which the corresponding array element is to have, e.g.

```
(1:X+Y, 3:A, 2:93)
("A":193, "B":194, "C":195)
```

Consecutive integer subscripts may be omitted:

(10,20,30,50)

and repetition may be specified:

((2)(2,3)) is equivalent to (2,3,2,3).

Multiple dimensions are specified by nesting the parenthesised lists:

((2,3,4),(3,4,5))

These forms may be used wherever a corresponding array reference can be used, e.g. in INITIAL, assignment, or argument. But they cannot be subscripted.

This constructor notation is used for array literals also. When the literal is a commalist of single character literals of a VALUES type (e.g. for (*)CHARACTER) the following shorthand is available:

"ABC' for ("A", "B", "C") etc.

Type names may be used on constructed values to specify their type if required:

BIT('1001") POINT(0,1,1)

<u>Strings</u>

The datatype attributes:
BIT (extent-expression)
CHARACTER (extent-expression)

are builtin with a definition as one-dimensional arrays:
(extent-expression) BIT
(extent-expression) CHARACTER

and certain operations not available for all arrays are defined for these types.

Additional string types can be defined, e.g.

```
DCL BCDCHAR VAIUES ("A", etc ...);
DCL BCDSTRING BASED (*) BCDCHAR;
DCL STRINGVBL BCDSTRING(12);
```

<u>Structures</u>

Structures can be declared using level numbers, e.g.

```
    DCL 1 S,
          2 X,
            3 A FIXED DECIMAL(5)
            3 B CHARACTER(3),
          2 C MYTYPE
```

Structure values can be constructed; the notation is similar to that for arrays: a parenthesised list, each element specifying which component and the value it is to have, e.g.

```
(X:(A:99,B:'ABC'),C:XYZ)
```

[One can debate whether this is too elaborate. There are merits in schemes which allow assignment of a list of values to a structure, e.g. MARY='FEMALE',27;]

CONSTANT/INITIAL may be specified on structures or their elements (but not both). When specified on a structure, the constructor notation is used for the value.

```
DCL 1 COMPLEX BASED, 2(REAL,IMAG) FLOAT DECIMAL(6);
DCL X COMPLEX INITIAL (REAL:1, IMAG:-2)
```

For types declared based on structures, the typename may be written before a constructor where necessary to specify the type intended, e.g.

```
COMPLEX (REAL:1, IMAG:-2)
```

Variants

In some structures, a field may appear or not depending on some other information. This can be declared, but the discriminating information must be part of the structure, so that the structure is self-describing. This makes it possible for the compiler to check for correct usage - for a further discussion see the section on "DYNAMIC STORAGE".

The alternative fields are declared at the same level and each bear the attribute CASE, e.g.

```
DCL 1 PERSON BASED,
      2 SEX VALUES ("M","F") CONSTANT,
      2 MAIDEN_NAME CHAR(12) CASE (SEX="F"),
      2 MILITARY_SERVICE BIT CASE (SEX="M");
```

The CASE expressions must be of constant value through the lifetime of the structure. Structure elements later in the structure may not be involved. The expressions are evaluated when the structure is allocated and storage is provided for the element for which the expression is true. There should be only one. [This might be a bit strong. We could say the first applies, as with SELECT. We could even allow none or several of the CASEs to be present. In this case IF might be a better word than CASE.]

Unions

The discriminating field will be provided automatically by the compiler if no expressions are specified on the CASE attributes, e.g.

```
DCL 1 NUMBER,
      2 X FIXED DECIMAL(6) CASE,
      2 Y FLOAT  DECIMAL(6) CASE;
```

In this case the discriminating field is not constant, but will reflect the current value of the structure. Sufficient storage will be allocated for all possible cases.

All forms of aggregation may be combined without restriction. Arrays of arrays, arrays of structures, structures of arrays are allowed. Variants and unions may be specified for minor structures and may be nested.

1.7 REFERENCES

Reference to a declared variable or constant is by its name. Aggregates may be referenced in toto by this means. An element of an array is specified by subscript(s). Trailing subscripts may be omitted, thus referencing a connected array of lower rank. Components of structures are referenced with dot-qualified names. For arrays of structures and vice versa, any subscripts in a reference must be written on the name which is declared dimensioned. Subscripts may not be omitted on qualifying names, to avoid disconnected references.

Variants and unions may be referenced in toto like other structures. The components may also be referenced: in this case the compiler will check that the discriminating field(s) have values consistent with the CASE of the component referenced. (This check may be done at run time, or flow analysis at compile time may make it unnecessary.) The exception is assignment to a CASE component of a union, which is always allowed. The (hidden) discriminant is set to describe that CASE.

Examples

```
DCL X BIT;                  X
DCL A(10) CHARACTER;        A(3), A(I+2)
DCL B(20,20) COUNTER;       B(5,2),B(7), B, not B(,7) or B(*,7)
DCL 1 S(10),                S, S(2)
    2 C (5) CHARACTER,      S(5).C(2), S(5).C, not S.C(5,2) or S.C
    2 D BIT                 S(2).D, not S.D(2) or S.D
```

For a one dimensional array where the dimension is of integer type, as well as referencing a single element, a subarray can be referenced, e.g. (using the declarations above)

A(2:5)means a 4-element array consisting of
A(2) A(3) A(4) and A(5).
This array is of type (4) CHARACTER.
B(20,X:Y)sub-array can only be specified for the rightmost dimension, to avoid disconnected references.

[It might be useful to include the notation

A(2::4) also, meaning the same as A(2:5). This can often lead to a constant length being apparent to the compiler and resulting in better code. A useful mnemonic for these operators is to consider the number of dots: 2 "two" 5 and 2 "four" 4.]

1.8 ALLOCATION AND ASSIGNMENT

Allocation, freeing and assignment are defined for all datatypes except CONDITION, AREA and FILE, for elements, aggregates and declared types.

A declared variable or constant is allocated on entry to the procedure in which it is declared. Any initial value declared for it is assigned to it at that stage. During the lifetime of the procedure, i.e. until control returns from it to the caller, the variable or constant may be referenced from any point in the program where it is known (see under "scope of names"). When control leaves the procedure the variable or constant is freed. On a subsequent call to the procedure, the variable or constant is allocated again and re-initialised if INITIAL was specified.

For dynamic or explicit allocation see under "DYNAMIC STORAGE".

Assignment

The assignment statement is written thus:

reference = expression

meaning that the expression is evaluated to give a value (the "source") and this value is given to the variable identified by the reference (the "target"). Assignment is also performed in other constructs of the language, for example a DO-loop with a control variable, or argument passing.

Assignment is defined for all types when the source and target are of identical type. This means for elements:

1. both BIT or both CHARACTER

2. both FIXED with same precision and scale

3. both FLOAT with same precision

4. both POINTER with same TO and IN

5. both BIT(n) or CHARACTER(n) where n is the same value for both

6. both the same declared type, with the same values for any * specification

7. For arrays:the same number of dimensions, type and values of each dimension, and same type of element.

8. For structures:same names and types for all components, including CASE attributes where present. (Major structure names may differ).

Assignment is also defined for some cases where the source and target are not of identical type.

FIXED=FIXED
: of different precisions. If the value of the source exceeds the range of the target, the condition SIZE is raised.

FIXED=FIXED
: where target scale is not less than than source, but not vice versa.

FLOAT=FLOAT
: where target precision is greater than source.

FLOAT=FIXED
where target precision is not less than source.

Any other conversions must be done using explicit conversion functions.

1.9 OPERATIONS

There are four syntactic forms for operations, three return a value and thus appear in expressions and the fourth does not and thus appears as a statement.

i) prefix operations

```
expression::= prefix-operator expression

prefix-operator::= + | - | ¬
```

ii) infix operations

```
expression::= expression infix-operator expression

infix-operator::= + | - | * | / | & | && | ⊥ | ⊥⊥
                  | = | ¬= | >= | <= | > | < | identifier
```

The priority of these operators is:

```
highest   * /
          + - ||
          = > < >= <= ¬=
lowest    & | &&
```

If two operators of the same priority are adjacent, parentheses must be used to specify the exact meaning. The exception is +, - and || where they are performed from left to right, in recognition of the nature of the standard definition of these operations. Identifier operators have undefined priority and parentheses must always be used if they appear with another operator.

```
examples:     A+B-C
              (A/B)*C
              A MOD B
              (A MOD B)LOG 10
```

iii) function operations

```
expression::= identifier (expression-commalist)
```

iv) subroutine operations

```
statement::= [CALL] identifier1 [(expression-commalist)]
                    [identifier2  (expression) ]
                          :
                          :
                    [identifiern  (expression) ];
```

The reason for allowing only a standard list of operator symbols, and for not allowing identifier prefix operators is that with the above rules an expression can be parsed without knowledge of what operators have been defined.

To compile the program, a definition must be available for each operation in the program, i.e. a definition of that operator on the appropriate operand data types.

These definitions may be user-written or they may be built in to the language.

[It would be good documentation if the invocation of an operation indicated whether the the arguments were to be altered or not. Since = is not otherwise used as a prefix or postfix operator, a reasonable syntax would be, for example:

```
CALL X( =A+1, B=, =C= );/* A used,B set, C both */
```

The same syntax would have a use with parameter and export lists.

The DoD has said that functions on the left of assignment are not required, although without them there is no reasonable way to make sparse arrays (say) a user extension. They could also be used to implement sensor output with the syntax of variable reference.]

Built-in operations

prefix "+"
defined on all FIXED and FLOAT, yielding the same.

prefix "-"
as prefix "+"

prefix "¬"
defined on BIT, yielding BIT ;defined on BIT(*), yielding the same

infix "+"

defined on FIXED DECIMAL(p1,q1) and FIXED DECIMAL(p2,q2) yielding FIXED DECIMAL (1+max (p1-q1,p2-q2) + max (q1,q2), max (q1,q2) ;defined on FLOAT DECIMAL(p1) and FLOAT DECIMAL(p2) yielding FLOAT DECIMAL(max(p1,p2))

infix "-"

as infix "+"

infix "*"

as infix "+" except FIXED result is (p1+p2+1, q1+q2)

infix "/"

as infix "+" for FLOAT ;only allowed for FIXED in positions where there is an explicit target, i.e. as the source of an assignment, since the precision required to hold the result exactly is indefinitely large. With an explicit target, the precision of the target can be used. Thus A=B/C; can be written, but not A=(B/C)*D for FIXED data.

infix "&"

defined on BIT or BIT(*) and the same, yielding the same.

infix "|"

as infix "&"

infix "&&"

as infix "&" (exclusive-or)

infix "="

Defined for FIXED and FIXED of any precision/scale, FLOAT and FLOAT of any precision, and all pairs of identical types, including aggregates, yielding BIT.

infix "¬="

for all types, a¬=b defined as ¬(a=b).

infix ">"

defined for FIXED and FIXED of any precision/scale, FLOAT and FLOAT of any precision, pairs of the same VALUES type, including CHARACTER, but not BIT. Yields BIT.

Also defined for CHARACTER(*) and CHARACTER(*) including different lengths, yielding BIT. Normal "dictionary ordering" is used.

infix "<"

for all types, a<b defined as ¬((a>b)|(a=b))

infix "<="

for all types, a<=b defined as (a<b)|(a=b)

infix ">="

for all types, a>=b defined as(a>b)|(a=b)

infix "||"

defined on one-dimensional arrays, with integer dimension and elements of identical type.

Yields a one-dimensional array of the same type with dimension equal to the sum of the dimensions of the operands.

Only allowed where there is an explicit target, i.e. as source of assignment. A=B||C||D etc. is allowed.

Builtin function operations

The builtin functions EMPTY, NULL, ONCODE and DIMENSION are provided as in PL/I. (Because results are system-dependent, ONCODE is allowed only in system modules.) Most other functions can be written in the language and INCLUDEd from libraries.

Builtin subroutine operations

The statements defined in the language include READ, REWRITE, WRITE, OPEN, CLOSE, GET and PUT. These are described in the section "INPUT/OUTPUT".

User-written operations

These are defined by procedures (see below).

1.10 PROGRAM STRUCTURE

Flow of control in programs is specified using the following constructs:

```
IF BIT-expression THEN clause; [ELSE clause;]
```

Each clause may be an executable statement, including IF or a DO-group. They may not be labelled.

```
    [label:] DO;
             executable-statement-list
             END [label];
```

Is a way of combining statements into a clause for use in IF or SELECT constructs. If the label appears on END, the same label must appear on the DO.

DO WHILE (BIT-expression);

The body of the DO-group is executed zero or more times; the WHILE expression is evaluated and tested before each iteration.

[Indefinite loops can be written as DO WHILE(TRUE). It might be better however to have a more elegant form such as LOOP or DO WHILE without operand.]

DO ctlvar = expression [BY expr2] [DOWN] TO expr3;

The control variable must be an unsubscripted unqualified variable. It may be FIXED or a VALUES type; in the latter case the BY option may not appear. It would be possible to extend this to allow any datatype for which the necessary operations + (for BY) and > were defined.

For FIXED the meaning is defined as:

```
    t2 = expr2; t3 = expr3;
    ctlvar = expression;
 y: if ctlvar > expr3 then go to x; [< if DOWN appears]
    do-body
    ctlvar = ctlvar + t2;
    go to y;
 x:
```

If the BY-option does not appear in the FIXED case, t2 is set to one. DOWN and the lack of it assert that t2 is less or greater than zero.

For VALUES, the meaning is similar - the body is executed with the control variable set to each of the specified values, in order. The VALUES type must be ordered, i.e. the operation '>' must be defined for it.

DO ctlvar;

Similar to above, the same rules apply for the control variable, except that the operations + and > need not be defined, and the order in whch in the control variable takes its values could be undefined. The body of the loop is executed repeatedly with the control variable set to all the possible values for its type, in an undefined order.

Multiple case construct

It would be possible to extend the IF statement to multiple cases, e.g.

```
IF A = |                       IF |
       | 1 THEN clause;           | A>0 THEN clause;
       | 2 THEN clause;           | A=0 THEN clause;
         .                        | A<0 THEN clause;
         .
         .
         [ELSE clause;]
```

This is very like the notation of "guarded commands" introduced by Dijkstra (ref 6). But we have a PL/I construct already implemented by IBM and proposed to ANSI, so we had better stick to it.

```
SELECT (expr);                     SELECT;
WHEN (expr-commalist1) clause;     WHEN (BIT-expr1) clause;
   .                                  .
   .                                  .
   .                                  .
[OTHERWISE clause;]                [OTHERWISE clause;]
END;                               END;
```

The WHEN expressions are evaluated in order until either expr=expr-n or BIT-expr-n is true for the two versions respectively. Then the corresponding clause is executed and control passes to after the END. If no WHEN expression is satisfied the OTHERWISE clause is executed, and if there is none the ERROR condition is raised.

GO TO label;

This statement transfers control to the statment with the corresponding label, which must be in the same procedure as the GO TO statement. Transfer into (or out of?) a THEN or WHEN clause, or a DO group is forbidden.

LEAVE [label];

This statement transfers control to the statement following the END of the DO or SELECT group with the corresponding label, or if no label is specified, the immediately containing group. The LEAVE statement must be in the group which is left.

;

A lone semicolon is a null statement which does nothing. It can be used to carry a label, or to express a null clause, after WHEN( ) for example.

Procedures

PROCEDURE blocks specify the code for a function (i.e. a prefix, infix or function operation) or a subroutine. Functions return a value, whereas subroutines do not. Subroutines may have keyword arguments, whereas functions do not. The name of a procedure is an identifier or, for prefix and infix operators, it may be an operator between double quotes.

```
{label|"operator"}:
[(condition-commalist):]
PROCEDURE [(parameter-identifier-commalist)]
          [RETURNS(datatype) |
          {identifier (parameter-identifier)}... ];

declaration-list
procedure-list
executable-statement-list
END [label];
```

Examples

```
SIN : PROCEDURE (X) RETURNS (FLOAT DECIMAL(6));
      DECLARE X FLOAT DECIMAL(6) CONSTANT;
      .
      .
      RETURN(expression);
      .
      .
      END SIN;

MOD : PROCEDURE (X,Y) RETURNS (FIXED DECIMAL(6));
      DECLARE (X,Y) FIXED DECIMAL(6) CONSTANT;
      .
      .
      RETURN(expression);
      .
      .
      END;

"+" : PROCEDURE (X,Y) RETURNS (FIXED DECIMAL(6));
      DECLARE (X,Y) FIXED DECIMAL(6) CONSTANT;
      .
      .
      RETURN(expression);
      .
      .
      END;
```

```
SEARCH : PROCEDURE (A) KEY (B) SET (C);
         DECLARE A (*) CHARACTER(8) CONSTANT,
             B CHARACTER(8) CONSTANT,
             C FIXED DECIMAL(3);
         .
         .
         END;
```

Now these definitions can be used thus:

```
SIN(P)                    )
MOD(P,Q) or P MOD Q       )in expressions
P+Q                       )

SEARCH(P) KEY(Q) SET(R);  as a statement
```

[We have introduced keyword arguments, in a minimal way, for readability. The further steps of allowing testing for the presence of arguments (and hence defaulting) and allowing keywords without arguments, would allow the user to adopt command-language-like syntax.]

[It might make programs clearer if the types of the parameters, like the type of the returned value, were specified in the parameter list rather than in separate a DECLARE statement:

```
SIN : PROCEDURE (X:FLOAT) RETURNS (FLOAT);
```

But perhaps it isn't worth changing.]

Parameters

The parameters of a procedure are allocated on entry to the procedure. Any extents specified by * in the parameter declaration are taken from the type of the corresponding argument. Then the argument(s) are assigned to the parameters, following the normal rules of assignment. If the parameter is CONSTANT, it may not be subsequently assigned to (directly or indirectly). CONSTANT and expression arguments may only be passed to CONSTANT parameters. The parameters of functions must be CONSTANT.

If the parameter is not CONSTANT, it may be assigned to during execution of the procedure, and will be assigned back to the argument variable when the procedure returns control to the caller.

[The CONSTANT attribute is really a declaration of something the compiler could deduce from the program. We could introduce a similar RESULT attribute, declaring that a parameter was never read before being set.]

A compiler can compile code to pass parameters by reference, rather than by value, if it can determine at compile time that the results will be the same. For synchronous calls this will be the case when argument and parameter are of identical type, unless a subroutine assigns to a parameter under some alias. This could be done, for example, by assigning to an argument variable, or to another parameter when the same argument variable was passed to both. The language could forbid all these cases anyway (see more below). Particularly for large aggregates, passing by reference is a useful optimisation.

Inherited names

Functions and subroutines may reference names declared in containing procedures, but not names declared in contained procedures. (The name of a contained procedure is considered declared in its container.)

However a function may not (directly or indirectly) assign to any inherited variable. (This means that functions "have no side-effects".)

A subroutine may not reference an inherited variable which has been passed to it as an argument to a non-CONSTANT parameter, or assign to one which has been passed to a CONSTANT parameter. Neither may the same argument variable be passed to two non-CONSTANT parameters.

[This last paragraph is the "Ironman" requirement (paragraphs 7E and 7I). But it is difficult for a compiler to recognise these situations with certainty because of aliases. There arise in two cases - array subscripting and pointer qualification. Are A(I) and A(J) the same variable? Are PTR-> and PTR2-> the same variable?

If we have the restrictions, the compiler must in these cases compile the program (assuming they are not aliases) and give a warning message. This message should be suppressed by an assertion e.g.

```
ASSERT I¬=J;
```

The alternative would be to allow the alias situations, with the meaning defined by the description of the parameter mechanism in terms of assignment. Then a verifier would have more trouble with the program - trouble which would again be dispelled by the assertion.

The two approaches are not so very different in practice.]

Scope of names

The names under discussion are the declared names of data, including elements, aggregates and their components, variables and constants, and value-constants, the declared names of types (i.e. BASED) including the names of their components if based on a structure, and the declared names of operations (i.e. PROCEDUREs).

Procedures affect lifetime and scope. Declared objects are allocated on entry to the procedure in which they are declared and live until exit from the procedure.

The scope rules for procedures are the same as in PL/I and other block-structured languages. Names are known in the procedure in which they are declared and any contained procedures. If a name is already known in the containing procedure (and thus would be known in the contained procedure) a declaration of the same name in the contained procedure overrides the inherited definition.

[The compiler knows which names are imported into a procedure. The programmer can know from a cross-reference listing, but this is rather detailed. If the programmer is transporting the procedure he needs to know what other procedures should be transported at the same time. The CONTEXT(identifier-list) statement would allow the programmer to document names that the procedure imports. By naming an enclosing procedure (or a module) a single name in the list could ensure the names of that block were included. The compiler would check that the CONTEXT assertion implied all the names actually imported into the procedure.]

For operations, overriding definitions are not allowed. An inner declaration does not override unless the declarations of the operand types are identical. Otherwise the two definitions are both available and resolution is determined by the operand types. Builtin operations on builtin datatypes may not be overridden.

The algorithm for resolving an operation use is as follows. For the scope in which the use appears, then for the containing scopes in order, and finally for the builtin operation definitions, do the

following. If there is a procedure with parameters which exactly match, i.e. same number of operands with same keywords if applicable, and the types of the parameters and the operands of the operation being resolved are identical, except for a * in a parameter and a corresponding integer in the operand, then use that procedure. If not, if there is a procedure with the same number of parameters, same keywords if applicable, and the parameter types are valid assignment targets for the operands of the operation being resolved, and valid assignment sources in the case of non-CONSTANT parameters, then use that procedure. If not, try the next scope.

1.11 MODULES

Modules affect scope but not lifetime. A MODULE is written thus:

```
module-label:  MODULE [export-list]
               [declaration-list]
               [procedure-list]
               [executable-statement-list]
               END [label];
```

If a module contains executable statements, it may appear anywhere an executable statement may appear. If it does not, it may only appear in the declaration or procedure list of a procedure or module.

The scope rules for modules are as for procedures, except that names may be explicitly exported from a module.

Names known in the containing procedure or module are known in a contained module. Names declared in a module are not known in the containing procedure or module unless they are explicitly exported. The syntax of the export list is:

```
EXPORTS( export-item-commalist )
export-item ::= identifier [EXPOSED]
```

The identifier may be the name of a declared constant, variable, type or procedure. The option EXPOSED is used only with types. If a type is exported from a module, the base attributes are not known outside the module and so if the base was an array or structure, objects of the new type cannot be subscripted and components cannot be selected. The EXPOSED option specifies that the base attributes are known outside the module and references to elements or components are permitted. Values of the type can also be constructed by writing the typename as a function with a constructed value of the base type as argument.

```
M : MODULE
    EXPORTS (MYTYPE EXPOSED);
    DCL 1 MYTYPE BASED(100),
        2 A ...;
        2 B ...;
    END M;

    DCL MYVBL MYTYPE;             /* allowed without EXPOSED */
    MYVBL(J).A                    /* references only allowed */
    MYTYPE((100)(A:1,B:2))        /* with exposed            */
```

To expose some components and not others, the components to be hidden should be made into a separate, unexposed type, e.g.

```
DCL 1 HIDDEN BASED,
      2 H1,
      2 H2;

DCL 1 NEWTYPE BASED,
      2 H HIDDEN,
      2 E;

EXPORTS (NEWTYPE EXPOSED, HIDDEN)
```

In the scope in which a type is declared, assignment is defined between the declared type and its base type, and therefore the operation resolution rules allow the operations of the base type to be used on the new type, and data of the new type may be used anywhere data of the base type could be used.

```
e.g. DCL MYTYPE BASED BIT;
     DCL MYVBL MYTYPE;
     IF ¬MYVBL THEN ...        /* is OK */
```

If a type is exported from a module, whether or not the type is exported EXPOSED, no operations are available outside the module unless they too are exported (or defined in the outer scope, but in practice some operations would need to be exported to form a basis). The exceptions to this are the operations which are defined on all datatypes, viz allocation, freeing and assignment. These are always inherited without explicit export, and cannot be redefined.

[It might be useful to have a way of EXPOSing components only for read and not for write - all setting of the components would still have to be done in the module.]

Both operations inherited from the base type and operations defined in the MODULE may be exported, and the process can be nested: the available operations on a type can be used on any new type defined using it as a base.

```
M : MODULE
    EXPORTS (MYTYPE,INCREMENT);
    DCL MYTYPE BASED FIXED;
INCREMENT : PROCEDURE(P);
            DCL P MYTYPE;
            P=P+1;
            END;
END;

DCL MYVBL MYTYPE;     /* user type exported from module */
MYVBL=MYVBL+1;        /* error: no + operation defined
                         on MYTYPE                    */
INCREMENT(MYVBL);     /* ok */
```

A module might define a new type and the operations on it. It can also define operations between types: exported operations act on the types of their parameters.

The lifetime of data declared in a module is determined by the containing procedure. These data may therefore live from one execution of code in the module to the next, providing the function of "OWN" variables in ALGOL60 and replacing STATIC in PL/I, e.g.

```
M : MODULE
    DCL Z FIXED INITIAL(0);
    X : PROCEDURE;
        Z = Z+1;
        END;
    Y : PROCEDURE RETURNS(FIXED);
        RETURN(Z);
        END;
    END;
```

A program which includes the module M can call X any number of times, and call Y to discover how many times X has been called. The value will be reset on each entry to the procedure containing M.

["Ironman" asks for explicit loading of portions of programs. This could be perhaps be done using modules and/or procedures as the unit. They would have to be marked in the program as not loaded automatically (which is the default) and then an explicit statement could load and unload them. They could be referenced only when loaded.

```
M:MODULE LOADED;

     FETCH M;
     RELEASE M;  ]
```

Program libraries

Portions of a program may be kept as separate units in a program library. If the statement:

INCLUDE label-commalist;

appears in a program, the result is as if the program portion with the corresponding label had been taken from the library and lexically included at the point in the program.

The portions of program on a library may be:

i)labelled modules
ii)labelled procedures
iii)labelled declare statements

Modules can be used to include the definition of new types and operations on them, with some hiding of things not exported. Modules can also be used to include in-line portion of executable code, possibly with its own declarations.

Procedures can be used to include the definition of a single operation. Declare statements can be used to include a list of declarations - the "commalist" syntax of the declare statement allows an indefinite number of declarations in one statement. Note that this would not be done to include declarations of shared data in several procedures; in this case the declarations would be in a containing procedure which INCLUDEd the several procedures. But it would be useful for writing several versions of a procedure working with the same data.

[It would be possible to allow labelled DO and SELECT groups to be INCLUDEd from libraries also, although this would provide no function not already available by INCLUDing a module. In the same way, INCLUDing procedures provides no function over including a module containing and exporting that procedure, but it does avoid some extra statements and another name.]

Although the definition of the effect of INCLUDE is in terms of lexical inclusion, it is expected that compilers would be able to process library portions separately and combine the processed form into programs when INCLUDEd. This is especially true for procedures and modules containing type and operation definitions.

1.12 EXCEPTION HANDLING

During execution of a program, an exceptional condition may arise. We consider here conditions which arise synchronously as a direct consequence of the execution of the program, such as dividing by zero, not an interrupt from outside the program even if it was indirectly prompted by previous actions of the program. (LTPL-E call these synchronous conditions "loose happenings".)

There are no execution time error messages, in principle, because errors give rise to conditions and these conditions may be handled by ON-actions. However, an installation may provide a facility to 'dump' the status of an execution if a condition occurs for which you have not specified an ON-action. This 'dump' can be subsequently analysed (in conjunction with the source program) to provide a symbolic picture of the failing execution and the values of its objects.

Some conditions of this kind are builtin to the language and are raised by various language constructs. These are listed below. Others may be declared thus:

```
DECLARE condition CONDITION;
```

All conditions, builtin or declared, may be raised explicitly by the SIGNAL statement:

```
SIGNAL condition;
```

ON statements

What happens when a condition is raised is specified in ON statements:

```
ON condition GO TO label;
```

These may be written in any procedure, after the declaration-list and before any executable code. They may be regarded as a form of initialisation, although they generate no executable code. If a condition is raised, then if an ON-statement for that condition appears in the procedure, control is transferred to the label specified, which must of course be in the same procedure. If no ON-statement appears for the condition but one does appear for the ERROR condition, then that is obeyed. If no ON-statement appears

for ERROR either, the procedure is terminated and the condition is raised in the calling procedure, where the same definition applies.

1.12.1 SETS OF CONDITIONS.

There is a case for generalising the mechanism that allows ON ERROR to be the action for many conditions. If names could be declared for sets of conditions then ERROR would be a special case of this.

[It would be possible to allow as the ON-statement action not only a GO TO statement, but other executable statements or groups. But control must not fall through the ON-units, since there is nowhere to go.

For example:

ON condition SIGNAL another-condition;

might be useful in definition modules. Such an extension might improve readability, but it would provide no function not available through GO TO.]

Condition prefixes

It is possible to assert that a condition will not occur in a portion of program by writing a condition prefix on a PROCEDURE, MODULE, group or executable statement. This means that an optimising compiler need not generate code to deal with cases which would raise the condition.

label: [(NOcondition):] PROCEDURE ...

Such a prefix may be overridden on an internal scope by a (condition): prefix. And conditions and NOconditions may be combined in a commalist as a prefix.

Implementation note

The restrictions on placement of ON-statements mean that the enabled conditions and ON-actions can be bound at compile time to a particular portion of the program. This specification can thus be

kept statically with the object code of the program, and require no executable code to establish at entry to procedures. The terminate and raise-in-caller action can be done at run time using the already established call chain.

Builtin conditions

AREA	see under "DYNAMIC STORAGE"
ENDFILE	)
KEY	)
RECORD	)see under "INPUT/OUTPUT"
TRANSMIT	)
UNDEFINEDFILE	)
CONVERSION	raised by conversion functions BIT, FIXED, FLOAT
ERROR	described earlier
OVERFLOW	raised if the value of a FLOAT variable exceeds the capacity of the machine
UNDERFLOW	raised if the value of a FLOAT variable is too small for the machine to handle
SIZE	raised if the value of a FIXED variable is out of its declared range
SUBSCRIPTRANGE	raised if a subscript on an array element reference is out of the dimension range
ZERODIVIDE	raised if a division of FIXED or FLOAT by zero is attempted

[When a string is assigned a value of too great a length, PL/I raises STRINGSIZE. For a short value it pads. For arrays, different dimension is an error. In this language strings and arrays are the same, references to subarrays are possible, and we do not allow different length assignment. We could raise STRINGSIZE for assignment of both strings and arrays of differing lengths.]

1.13 INPUT/OUTPUT

There are two kinds of input-output required in the language, low-level and high-level. Low-level I/O exposes and exploits the characteristics of particular devices attached to a specific machine. High-level I/O has a meaning defined in the language, and is implemented on every machine, hiding the different characteristics of the machines.

High-level I/O

The main choice with I/O is whether to incorporate traditional operating system aspects into the language. IRONMAN implicitly asks for this. It is reflected by (1) using the name resolution mechanism for resolving TITLE arguments, and (2) using the language allocation mechanism for datasets.

A subset of PL/I I/O will serve for this without extension. High-level I/O is done with files. There are three kinds of files: stream files where character data (i.e. human readable) is transferred sequentially,character by character, RECORD files where data in internal format is transferred sequentially a record at a time, and KEYED RECORD where data in internal format is transferred a record at a time but not necessarily sequentially - the record number can be specified by a FIXED integer called the KEY.

File are declared in a DECLARE statement; the syntax is given in the section on "Data types".

[This subset of RECORD I/O is what in IBM PL/I is called REGIONAL(1). It provides the basic facilities provided by sequential and direct access devices. Content addressing by keys is device-dependent and complex indexes are implemented in software and could be written in this language as library routines; they have therefore been left out.]

Files must be explicitly opened before use and closed after use:

```
    OPEN FILE(filename) [LINESIZE(constant-expression)]
        {INPUT|OUTPUT|UPDATE};
    CLOSE FILE(filename);
```

[LINESIZE would be better as an additive attribute to PRINT in the file declaration, but PL/I does not allow that.]

The allowed input-output statements for each file type are given in the table below:

	Stream	RECORD	KEYED RECORD
INPUT	GET[SKIP] EDIT	READ INTO	READ[KEY] INTO
OUTPUT	PUT[SKIP] EDIT	WRITE FROM	WRITE[KEYFROM] FROM
	PUT[SKIP] LIST		
UPDATE		READ INTO	READ[KEY] INTO
		REWRITE INTO	REWRITE[KEY] FROM

In each statement a FILE option must appear after the statement keyword; the options must appear in the order listed.

The options are:

```
FILE (filename)
INTO (reference)    )these references will normally
FROM (reference)    )be structures
KEY  (integer-expression)
KEYFROM (integer-expression)
SKIP
LIST (expression-commalist)
EDIT (expression-commalist)(format-commalist)
```

LIST output uses a standard format for each data type and successive items are printed with an intervening blank.

In EDIT output each item is printed in the format specified by the corresponding format item. EDIT input is the reverse.

```
format::= (expression) (format-commalist) |
          COL (expression) |
          SKIP [(expression)] |
          PAGE |
          X (expression) |
          A [expression)]   |  for CHARACTER items only
          B [(expression)] |  for BIT items only
          E (expr1[,expr2]) | for FLOAT items only
          F (expr1[,expr2]) | for FIXED items only
          P 'picture'         for FIXED items only
```

Pictures are provided only for FIXED items, the allowed picture characters are 9VZ*$-.,B. All expressions in a format list must be constant integer expressions.

The restrictions on data-lists and format-lists allow compile-time pairing of data items and format items. This and the restricted combinations make it possible to compile in-line code for much formatting.

The following conditions may be raised by file I/O statements:

UNDEFINEDFILE[(filename)]
raised by OPEN if the filename is invalid.

RECORD [(filename)]
raised by READ/WRITE/REWRITE if the FROM/INTO operand does not match the file record.

TRANSMIT [(filename)]
raised by READ/WRITE/REWRITE/GET/PUT if there is an error in transmission.

KEY [(filename)]
raised by READ KEY/REWRITE KEY/WRITE KEYFROM if the KEY value is invalid.

ENDFILE [(filename)]
raised by READ and GET if an attempt is made to read sequentially beyond the end of the file.

These conditions may not appear in a condition prefix.

[These conditions would be better as options on the appropriate statement specifying action to be taken. The mechanism of block termination is not needed for I/O conditions and is more difficult to implement because of the filename. But that would mean a change from PL/I.]

["Ironman" also requires statements to create and destroy files. These could be options on OPEN and CLOSE or they could be separate statements.ALLOCATE and FREE might be used. The capacity of the file would have to be specified somehow, perhaps by a content-list similar to that on AREA declarations. Optionally a device could be specified; this would have to be machine dependent.]

Low-level I/O

The way in which attached devices appear in different machine differs greatly. For example, in System/360 one codes a program in the machine language of a different machine - the channel, whereas in the PDP-11 one references special addresses in the storage of the machine. And the nature of the attached devices may vary greatly too, especially for embedded computers controlling equipment.

This makes it impossible to provide either standard statements in the language with a defined meaning for doing control functions, or to provide standard statements to write the machine dependent code needed to implement such function. Exactly because one wants to access and exploit the special characteristics of the equipment attached to the computer, these operations have to be defined by the user.

These defined operations would typically be subroutine operations which look like statements, e.g.

```
READ DEVICE ('00C') INTO (reference);
OPEN VALVE (15,2);
```

The definitions can be written in system modules where access is provided to machine instructions, storage locations, interrupts etc and the device handling can be written in the way needed for the particular machine.

1.14 DYNAMIC STORAGE

Some problems are best programmed using dynamic allocation of variables, and the manipulation of pointers to them, but the danger with such techniques is that the type checking normally enforced by the compiler will be by-passed. It is of course possible to make the necessary checks at run time, but this is expensive in both processing time and storage. The language proposed here is designed to be "safe" and to permit the majority of checking to be done at compile time.

What do we mean by "safe"? The definition adopted here is that an erroneous program can never cause effects which no correct program could produce. If we accept the frailty of programmers, errors produced by programs which are correct (i.e. ones which obey the rules of the language), but which do not have the required effect, are inevitable. So this definition of safety reduces the set of possible errors to a minimum.

One of the implications of such a definition is that an access reference should always find a storage pattern which is a valid value for the referenced type. This means that instances should be initialised to a valid value on allocation (unless the compiler can be sure that they are set before any use).

For data types such as character or arithmetic, often all storage patterns are valid values, in which case no action need be taken to initialise them. For self-defining structures, discriminants should be initalised to the value (or one of the values) consistent with the storage allocated. Pointers should be initialised to a valid value, the most appropriate default being a recognisable null value. Similar action should be taken for other program control types such as files.

In the absence of pointer-qualified references, these actions would be sufficient. But with pointers the compiler must also ensure that when a pointer is used to refer to a type, it is the same type as that of the instance pointed to. Otherwise the program can read or

set the instance to invalid storage patterns. And if the asserted type is longer than the actual type, the "overhang" may result in the program setting unreferenced, and possibly unrelated, instances.

To solve these problems we must constrain pointers and make all variants self-defining.

There remains another problem which arises when an instance is freed. If subsequently a pointer to the instance is used, it will touch the freed storage, which may well have been allocated to another instance, possibly of a different type. It is possible, but expensive at run-time, to keep track of all pointers and nullify them when the instance is freed, or to free only when no more pointers exist (garbage collection). Another approach is to prohibit a pointer variable from containing values which point to instances with a possibly shorter lifetime than the pointer, since rules of this kind can be enforced at compile-time.

For pointers to implicitly allocated variables, this rule could be enforced at compile time but there is no need to allow pointer reference to such variables anyway, and excluding it removes the need for the compiler to worry about aliases for these variables.

For dynamically allocated storage, we must ensure that any freed storage is not re-used for a different type until all pointers to the previous use have gone away. We do this by allocating dynamic storage in AREAs which have a lifetime known at compile time.

These areas are declared like variables:

```
DCL areaname AREA (content-commalist);
```

The content list specifies what type(s) may be allocated in the AREA, and how many of them must be accommodated:

```
content::= [(constant-integer-expression) type]
```

For types with * in their definition, integers must be provided to complete the type definition and allow space to be reserved, but variables may be allocated in the AREA with any values for these asterisks. The number which can be accommodated will differ from the number of variables with the specified dimension.

Variables (and constants) are explicitly allocated by the ALLOCATE statement and freed by the FREE statement.

```
ALLOCATE [type] SET(pointer reference) [IN(area-reference)]
                                       [INITIAL(expression)];

FREE Pointer-reference -> [type] [IN (area-reference)];
```

Pointers are declared in a declare statement, and must be constrained to a particular type in a particular AREA. Since this AREA is known in the declaration, its lifetime must be at least that of the pointer being declared.

```
DCL P POINTER [CONSTANT] TO (type[CONSTANT]) IN(area-reference);
```

Pointers may be variable or constant, they may themselves be implicitly or dynamically allocated, and they may be components of aggregates. Pointers are of the same type for assignment only if they point to the same type in the same AREA. They must point to identical types, not just types between which assignment is possible.

References to dynamically allocated objects are made by dereferencing pointer values with the "->" symbol, and they may be dot-qualified and subscripted like other references:

```
Pointer-reference -> [type]
[dot qualification | (subscript-commalist)]...
```

for example:

```
P->BIT   P->INTEGER    P->
P(5)->   S.P->T.C(J)   P->.C
```

The optional parts of ALLOCATE and FREE statements, and of pointer references, must be the same as the pointer declaration if they do appear and are inferred from the pointer declaration if they do not appear. They are permitted to improve the readability of programs, and to increase the compatibility with PL/I.

Note that we do not have any referencing operation, such as ADDR in PL/I. We do not want pointers to implicitly allocated objects. Pointers to explicitly allocated objects are provided by ALLOCATE SET. And we do not want pointers to components of explicitly allocated objects, since the compiler would then have to worry about diagnosing attempts to free such pointers.

If an attempt is made to ALLOCATE a variable in an AREA in which there is insufficient space, the AREA[(area-reference)] condition is raised. [This, like the file conditions, might be better as an option on the ALLOCATE statement.]

Implementation note

Dynamically allocated objects which are FREEd are available for re-use, but only for another object of the same type. This cannot be done by a free chain through the spaces, since they may still be referenced (erroneously) by a surviving pointer. Instead a list of free instances of each type in the area must be kept. (This might be more efficient in a virtual store anyway.) This free list can be found easily because the AREA in which the free space is can be determined at compile time from the pointer declaration.

For types with * extents, spaces can only be re-used for another instance with the same values for the asterisks. But we do not need to keep a separate list for each set of values since the asterisk values in the space itself will remain. The same applies to variant structures with explicit discriminants.

For variant structures with implicit discriminant, i.e. CASE without expression, the space can be reused for another instance of the same type in the normal way.

Subroutines and Functions

Pointers can be passed as parameters and as return values from functions. The parameters must be declared to be the same type as the argument, therefore the AREA must be known in the called procedure as well as the calling procedure.

The restrictions on functions, that they must not set inherited variables, could be bypassed by setting a variable referenced by a pointer parameter. Therefore a pointer parameter must not only be a CONSTANT but must be TO (CONSTANT non-pointer). Note that a function may not allocate or free in an inherited AREA, since this is changing the value of the AREA.

There is no reason why AREAs should not be passed as parameters. But there would be little point: the parameters would be (conceptually at least) a copy, and pointers into the argument could not be used on the parameter and vice versa. So it is best to disallow this. AREA assignment is similarly useless.

[As well as allocation in AREAs, we could allow allocation without freeing of objects in a global heap referenced by pointers not constrained to an AREA. But it is not clear that this provides useful additional function. If the objects are never freed, they might as well have been declared, rather than allocated. And an AREA declared in an outer scope is much like a global heap anyway.]

1.15 PARALLEL PROCESSES

We are not concerned here with portions of an algorithm which could be done in parallel because they are disjoint: a clever compiler could work these out from a sequential program anyway. We are concerned with computer processes which must run in parallel because they interact with each other and with real-world processes, such as parts of the equipment which an embedded computer is controlling.

This is the part of the language for which it is most difficult to decide on the design. "Ironman" is at its most vague in this section, no doubt because existing languages use several different approaches and there is no consensus on which is best. The area is one in which there is a lot of current research. And ANSI PL/I has no parallel constructs at all either to guide or constrain us.

The main choices are firstly how to introduce the code of the parallel activities (processes) and how to start their parallel execution, and secondly how co-operating processes, once started, synchronise their activities.

Processes

There is clearly a need for the static programming of a fixed number of processes; the number perhaps depending on the physical equipment being controlled. This is the "independent tasking" model which the industry control community (e.g. the Purdue Workshop) seem to be asking for. "Ironman" in section 9A seems to ask for a number of processes which varies at run-time, but which is predictable at compile-time. Others believe that it is desirable to create processes dynamically in response to happenings, for example to create a process to track each target identified by a radar system.

Our approach to dynamism determines the main structure of processes - that processes can be created dynamically but the efficiency of simple cases will not be compromised. This seems to be valuable for systems where large numbers of processors are to be exploited.

Once we have run-time starting of new processes, it is not much more complicated to allow an indefinite number. With adjustable data, storage usage by procedures cannot always be predicted at compile-time and so we need some form of storage management even with a fixed number of processes. (If the number of processes and the data sizes are all known at compile time, obviously a compiler could optimise by doing the storage allocation at compile-time.)

"Ironman" also requires that processes are able to terminate others running concurrently. If we have language to allow signalling between co-operating processes, then one could signal another which on receipt of the signal would terminate. This seems adequate. To allow the destruction of processes would raise complications about graceful termination which are not worth facing unless there is good reason.

[If it were necessary to do this, the way would probably be to have a FINISH statement which raised a FINISH condition in the other process, allowing it to terminate gracefully, as well as some form of DESTROY statement which stopped it immediately.]

If we do not have any language for destroying or suspending other processes, we do not need any name or other "handle" for the execution of a process. They can be anonymous. All we need to specify to start a new process is the portion of code which should run.

A procedure would do, but since there are special rules for what concurrent process executions may and may not do, it is probably clearer for the compiler and the human reader if the code is specially marked. So we use a procedure where the word PROCEDURE is replaced by PROCESS.

```
process-name: PROCESS (parameters) ... etc
```

To start a new execution of a process we write a START statement, which is similar to a subroutine invocation but starts the concurrent execution of a new process.

```
    START process-name (argument-commalist)
         [identifier(argument)]...
         PRIORITY(constant-integer-expression);
```

The parameters of a PROCESS must be CONSTANT, since the STARTer does not wait for the new process and so cannot accept the return assignment of a variable parameter.

<u>Scheduling</u>

Section 8D of "Ironman" requires that the language should not require an operating system to run. This implies that the scheduling of concurrent processes should be simple, just the allocation of the ready processes to the processor(s). It precludes language such as

SCHEDULE process AT 4PM;
SCHEDULE process ON interrupt;

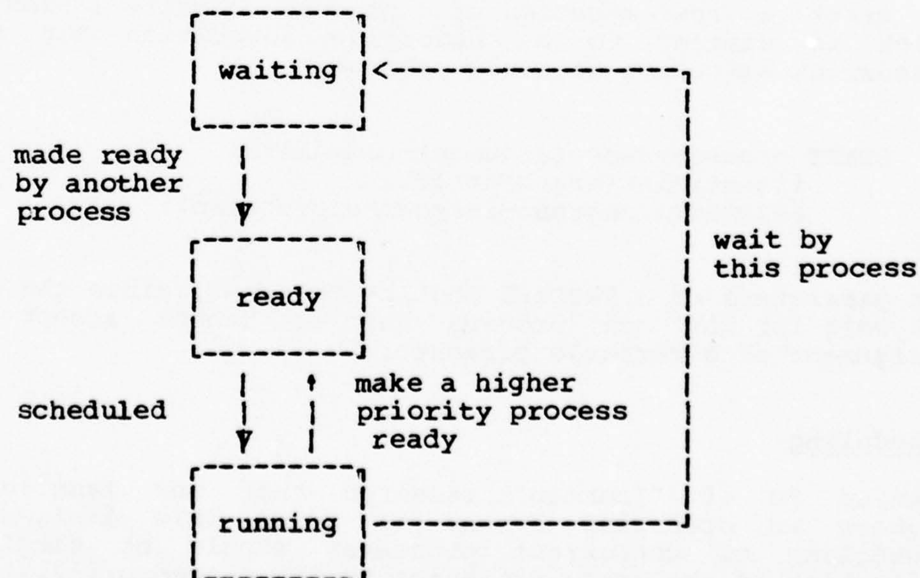
This function can be obtained in another way. A process should be started which waits for 4PM. It will then continue, and might well then load into storage a portion of program to be executed. This approach allows the programmer to decide whether the program should be loaded ready to respond quickly, or not loaded until it can run.

The same lack of operating system precludes the implementation of clock facilities in the language, since the timing hardware attached to different machines varies widely. Rather a machine-dependent module should be written which provides the facilities required (such as a signal at a particular hour) using the hardware. Other processes can then wait for this signal.

The simplest scheduling algorithm would be to run a process until it can run no longer (i.e. it waits for something) and then look for another ready process. Processes made ready by actions of the running process would be run next when it relinquished control. This is not adequate since we do need to be able to specify for an urgent process that it starts as soon as it is ready but is not pre-empted by other processes which it may make ready. So we introduce the PRIORITY operand on START. The higher priority processes are run in preference, i.e. it is always true that (at least) one of the highest priority ready process(es) is running.

(If PRIORITY was never used, the compiler could optimise to a simpler scheduler.)

The state transition diagram for processes is thus as follows:



Synchronisation

There are two main reasons for synchronisation between concurrent processes. Either they both use some resource which can only be used by one process at a time (mutual exclusion) or one process cannot proceed until a certain situation occurs and another process makes it occur, thus allowing the first to continue (signalling).

Mutual exclusion is a special case of the more general signalling, and can be implemented using a signalling primitive such as semaphores. However this approach allows no checking by the compiler of the correct coding of the rather common mutual exclusion. Coding this with signalling is rather error-prone, and the errors can be difficult to detect by testing.

We can introduce separate executable operations for locking and signalling (for example LOCK/UNLOCK and POST/WAIT in Series/1 PL/I). This allows some checking of the locking, but it has to be done at run-time. And there is no declared association of the lock with the resource, and so the checking is somewhat limited.

To allow compile-time checking we must introduce some static program construct such as the monitor in Concurrent PASCAL (ref 8) or the interface module in MODULA (ref 7). These constructs are a portion of the program in which mutual exclusion is enforced - only one process may be executing it a once. Objects shared between processes must be declared in monitors and accessed by calling procedures in the monitor. This works, and it has a certain elegance. To write in a language with monitors is to accept a certain design discipline, which can be useful. The problem is that monitors are rather static. They implement mutual exclusion for (in effect) monadic operations only, since only the shared objects in one monitor can be accessed at once. It is not possible to acquire exclusive access to several objects and then operate between them, unless they are all in the one monitor, and thus in effect sub-components of a single shared object.

Signalling between processes is done using special objects (variously called conditions, signals and queues) which can be declared in a monitor and signalled or waited on by processes executing in the monitor. This method has the drawback that the signal may well represent a condition involving program data, but this association is not declared in the language and therefore not very clear. WAIT and POST in Series/1 PL/I is very similar.

Leaving it to the programmer to invent signals and correlate them with the progress of the execution can be efficient (in the same way that letting the programmer do register allocation can be efficient) but it would be an abdication of the compiler's checking powers. We have chosen shared objects as the synchronization concept instead. If we recognise that the resources for which mutual exclusion is required may well be program data, and will usually be represented by program data if they are not (e.g. the status of real-world objects, outside the computer), we can make it explicit in the program that these data are the resources being contested.

Similarly, conditions which a process waits for (i.e. situations, nothing to do with CONDITIONs and ON statements) are often logical expressions involving program data, and will usually be represented as such if they are not. This logical expression should be specified in the wait operation.

So we introduce a SHARED attribute for data. This can be specified on any variable element or aggregate, or FILE. Only objects with this attribute can be used by more than one process. Also we introduce a static construct to specify the portion of the program where mutual exclusion applies, and allow access to these objects only in such portions. We want to wait for logical expressions involving SHARED data, and we need to enter mutual exclusion from the moment the expression becomes true. Otherwise it might no longer be true by the time the process entered mutual exclusion.

The final consideration is that we sometimes want to wait for one of several conditions - for example for a tank to become full or for a time interval to pass - and to take different actions depending on which becomes true. This leads us to a variation of the SELECT construct.

```
SELECT
WHEN(TANK_FULL) DO; ... END;
WHEN(TIME>LIMIT) DO; ... END;
OTHERWISE WAIT;
END;
```

Each WHEN expression may contain references to SHARED variables. If the process can acquire exclusive use (i.e. a lock) of these variables, and the expression is true, the corresponding WHEN clause is executed holding these locks, then the locks are released and execution continues after the END.If this cannot be done for any WHEN-expression, the process waits until it can, if OTHERWISE WAIT appears. If the OTHERWISE clause is an executable statement this is executed instead.

If we want access to a SHARED variable, but no condition is required we can write WHEN(AVAILABLE(V)). The function AVAILABLE returns true if the SHARED variable argument is not locked.

[The SELECT construct above might be thought too similar to the normal multiple-case SELECT, in which case we could use another keyword in place of SELECT. Alternatively we could use WHEN in the synchronising SELECT and another word, perhaps IF, in the original multiple-case SELECT. OTHERWISE could be changed to ELSE too.]

A process may reference inherited variables only if they are declared SHARED, and SHARED variables may be referenced only in WHEN clauses where the variable is mentioned in the corresponding WHEN expression. Inherited constants may be referenced without restriction, and so constants are never declared SHARED.

Locking SELECT groups may be nested in the usual way (although this should be avoided where possible since it can lead to deadlock), and procedures can be called from WHEN claues which themselves contain locking SELECT groups. A SHARED variable may not be locked twice, even by the same process. Even in the same process this would indicate that an atomic operation was being begun on the variable while another was still in progress.

The compiler will check that the rules in the previous two paragraphs are obeyed, since violations may well be unintentional errors. For aliases the compiler may not be sure. As with the parameter passing rules, the compiler cannot always know whether A(I) and A(J), PTR1->type and PTR2->type are the same objects or not. In these cases the compiler will warn, and an appropriate assertion will dispel the warning.

Note that if a procedure is called from a WHEN clause, it cannot access the SHARED variables which are locked in that clause. For the procedure to operate on them they should be passed as arguments to it.

SHARED objects can only be declared directly in a PROCESS block, not in any contained procedures. If a PROCESS in which SHARED data is declared reaches termination before any contained processes which share the data, it waits for them to terminate so that the SHARED data is not freed.

A process does not (except under the rule above) wait for processes which it started, and this allows "independent tasking" using processes at the same lexical level.

[Read/Write locking

The above is entirely in terms of exclusive locking. For WHEN clauses which only read a SHARED variable it would be possible to use shared locking. This is not always a good idea since it involves more overhead to acquire a lock and more storage in the shared variable. We could introduce two attributes say EXCLUSIVE for what we have already, and SHARED for read/write locking. It is much more complicated though, and read/write locking algorithms are the subject of much discussion.

Note that a compiler could choose to implement the language we have already using read/write locking, as long as the algorithm ensured awarding of locks in the same order i.e. a waiting writer should stop the granting of read locks.]

External interrupts

Signals from real-world processes, i.e. external interrupts, should appear to the program just like signals from another software process. So a "process interrupt" line on a processor should appear as a SHARED BIT which can be referenced in a WHEN expression. Some machine-dependent language would be needed to specify the hardware connection (see section on "machine-dependent language").

Files

SHAREing of files can be used to synchronize dataset access provided one defines that acquiring rights on the file object also acquires rights on the data set OPENed with it. This is a slight distortion of the SHARED concept since two file objects each accessed by a different process (and hence strictly unshared) could be used for shared access to a dataset. It would be more logical to allow the SHARED attribute for datasets (and even objects in them) but this has implementation problems - one does not want the anchor of a chain of processes to be on disk storage.

Conditions

If a condition is raised in a process, the normal rules apply regarding ON statements in that process. But if the condition is not handled in the process, it is not raised either in the containing or the STARTing process.

[The alternative would be to allow the declaration of SHARED CONDITIONs, which are raised in the containing process when it waits for the failing contained process to terminate.]

Implementation Note

The implementation of process scheduling can be done fairly simply, along the lines described in MODULA (ref 7). There is a control block for each process, perhaps at the top of the stack for that process. All the ready processes are in a chain. A SHARED variable has probably one extra word of storage: a bit to show if it is locked and a pointer. Processes waiting for this SHARED variable are chained to this pointer. When a process waits it detaches itself from the scheduling chain and joins itself to the appropriate waiting chain and then hands control to the next process on the scheduling chain. When the variable is unlocked, the unlocking process takes this chain of waiting processes and joins it on the scheduling chain. Any of these processes which is waiting for an expression which is still false, will find its way back into the waiting chain when it is scheduled.

Two things complicate this scheme a little. If processes have priority, the scheduling chain should be ordered, and processes should be put on it in the appropriate place. And when a process waits for one of several SHARED variables it must be attached to all the waiting chains, but removed from all when one variable becomes available. A special wait block must be put in each chain, these being chained in a ring with the process block. Note that this ring can be largely made at compile-time. Now the wait chains have wait blocks as well as process blocks on them and this is another reason why the chain must be followed on unlock. Waiting for an ANDed expression can be implemented by waiting for one then the other, although the code should be such that it does not hold one lock while waiting for the other. Note that the complications here only arise for programs which do such waits, simple waits for one variable will be much simpler.

A process block also needs a containing process pointer and a contained process count to implement the waiting for termination.

1.16 MACHINE DEPENDENT LANGUAGE

There is a need in a real-time programming systems language to be able to write code for functions which must be machine-dependent. (In this context machine means the target for the language compiler, which might be a virtual machine provided by an operating system.) The need arises in at least two ways: to use the special facilities on a particular machine or the equipment attached to it, or to exploit the characteristics of the particular machine to get optimal performance in some crucial area.

If we are not to lose the advantages of a high-level language with a lot of compile-time checking, we must localise the machine dependency as far as possible, allowing the rest of the program to be written with all the usual checking. This also means that if we want to re-implement the program on a different machine, the alterations are limited to these portions. We do this by having a special kind of MODULE in which machine-dependent things can be written. The SYSTEM MODULE is identified as such in its header:

```
label: SYSTEM MODULE;
```

but is otherwise similar to normal modules. Contained modules are not SYSTEM MODULES unless so marked.

Because they are modules, the effect of the machine-dependent coding is limited to whatever is exported. To rewrite the program for a different machine, we need only to rewrite the system modules to have the same exported effect. (The machine-dependent parts of a system module can easily be found by compiling it without the SYSTEM prefix - the diagnostics will then be a list of the machine dependencies.)

Data Representation

Data representation must be a part of its datatype, since it determines the storage requirements for objects of that type, and how operations are performed upon it. An assignment for example between two objects with different representations but otherwise the same type will lead to what is in effect an implicit conversion, which could take a lot of code to achieve. In the case of parameters the code generation will be even more complex. So we will make representation part of type, and all objects of the same type will have the same representation.

There will be an implementation-defined representation for each of the basic types of the language, e.g. FIXED(1) to (4) might be 16 bits. These representations cannot be changed, since if they were, normal statements could generate unexpected code.

There is an implementation-defined representation for aggregates also, but this can be overridden by the attributes

ALIGNED[(integer)]

UNALIGNED

which align the component on a multiple bit boundary. UNALIGNED means ALIGNED(1). To put spacing in an aggregate a dummy component can be declared by using * as its name.

It may be required to implement a datatype with some special representation, e.g. ones complement on a twos complement machine. (The only reason for wanting to do this is to construct data for transmission to another machine or device.) This is done by declaring the type and basing it on a bitstring. The operations on it will then have to be coded in terms of this bitstring. (The implementation-defined representation of BIT and one-dimensional array BIT(*) should always be packed.)

[If an exchange data representation was established the compilers could support an attribute for it. This would save the user having to create a bit equivalent in order to exchange data. Assignment from the installation representation to the exchange representation would be supported, but probably no other operations since it makes little sense to do arithmetic in other than the hardwares natural format, and is expensive to implement.]

[It might be useful to allow the declaration of a VALUES type specifying the representation for each value, although this could be done by basing it on a bit string as described above.]

Another facility available in SYSTEM MODULES to manipulate data representation is UNSPEC. UNSPEC(reference) allows reference to any object as the bitstring by which is is represented. This can be used to inspect or set any part of the representation, and provides a way to assign between types without conversion. This is of course highly machine-dependent.

Pointers

In SYSTEM MODULES the implementation of pointers as an integer address in storage is exposed. Pointers may be declared not constrained to any type or area. Integer operations may be performed on pointers and integers may be used as pointers to address specific storage locations.

[We could allow allocation and freeing not in an AREA. This would be "unsafe" in the terms of the section on "DYNAMIC STORAGE".

Data shared between processes

System modules may reference data not declared SHARED from several processes. The compiler will not generate and checking or locking code. This can be used when the programmer knows that there can be no interference.

Built-in instructions

The executable instructions of the machine are provided in the same way as in PL/S II (ref 9). A built-in instruction, with the syntax of a subroutine operation is defined for each machine instruction. These may be written in the place of an executable statement in SYSTEM MODULES, and to provide connection with the high level language, the operands to the instruction are references or expressions in the usual way. Code to evaluate these references and expressions is generated before the instruction itself, and also any code necessary to, for example, put operands into registers as appropriate.

For example, on the System/360,

```
    DCL  A FIXED DECIMAL(6),
         P POINTER TO (FIXED DECIMAL(6));
    AL (A, P->);
```

might generate

```
    L    R1,P
    L    R2,A
    AL   R2,0(R1)
    ST   R2,A
```

Other machine-dependent facilities

The machine-dependent facilities on each machine will obviously differ, but the ones listed so far: ALIGNED, UNSPEC, pointer arithmetic and an appropriate set of builtin instructions would probably be provided on all. Examples of other things which might be provided are:

i)A REGISTER attribute, as in PL/S II.

ii)An option like PSW(integer) on a SHARED BIT variables to specify a machine interrupt.

iii)An option on a file creation statement to specify the device to be used.

1.17 SYNTAX

The notation is similar to that in the ANSI PL/I standard, but the overstruck characters are not used.

Square brackets mean optional. List means one or more, commalist means one or more, separated by commas. Characters that might otherwise be taken as meta-characters are underlined.

Low-level syntax

```
program-text::=[ delimiter-list] delimiter-pair-list
delimiter-pair::= non-delimiter delimiter-list
delimiter::= +
           |-
           |*
           |/
           |<
           |>
           |=
           |<=
           |>=
           |¬=
           |¬
           |&
           |&&
           |?
           |⊥
           |⊥⊥
           |(
           |)
           |.
           |,
           |;
           |:
           |->
           |blank
           |comment
non-delimiter::= identifier
               | arithmetic-literal
               | string-literal
               | values-literal
identifier::= letter [identifier-character-list]
identifier-character::= letter | digit | _
letter::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z
digit::=0|1|2|3|4|5|6|7|8|9
arithmetic-literal::= decimal-number[ E exponent]
decimal-number::= integer[. digit-list]
integer::= digit-list
exponent::=[+|-] integer
```

```
string-literal::='string-symbol-list'
values-literal::= "literal-symbol-list"
string-symbol::= any-character-except-quote|''
literal-symbol::= any-character-except-double-quote|""
```

<u>High-level syntax</u>

```
procedure::= statement-name [ prefix] procedure-statement
            [ declare-statement-list][ on-statement-list]
            [ executable-statement-list]
            ending
statement-name::= identifier:
prefix::=( condition-prefix-commalist):
on-statement::= ON condition-reference goto-statement
ending::=[ statement-name] end-statement
executable-statement::=
       |  statement-name  [prefix] module
       | [statement-name] [prefix] group
       | [statement-name]          if-statement
       | [statement-name] [prefix] executable-single-statement
module::=module-statement
       [declare-statement-list] unit-list ending
group::= do-statement[ unit-list] ending
       | select-statment choice-list
       [ default-choice] ending
select-statement::= SELECT [(expression)];
choice::= WHEN( expression-commalist) executable-statement
default-choice::= OTHERWISE executable-statement
if-statement::= IF expression THEN executable-statement
              [ ELSE executable-statement]
executable-single-statement::= allocate-statement
                             | assignment-statement
                             | call-statement
                             | close-statement
                             | free-statement
                             | get-statement
                             | goto statement
                             | leave statement
                             | null-statement
                             | open-statement
                             | put-statement
                             | read-statement
                             | return-statment
                             | rewrite-statement
                             | signal-statement
                             | write-statement

condition-prefix::= computational-condition
                 |  disabled-computational-condition
computational-condition::= CONVERSION
                         | OVERFLOW
                         | SIZE
                         | SUBSCRIPTRANGE
                         | UNDERFLOW
                         | ZERODIVIDE
```

```
disabled-computational-condition::= NOCONVERSION
                                   | NOOVERFLOW
                                   | NOSIZE
                                   | NOSUBSCRIPTRANGE
                                   | NOUNDERFLOW
                                   | NOZERODIVIDE

declare-statement::= DECLARE declaration-commalist
declaration::=[ integer] declareds[ dimension-suffix]
              [ attribute-list]
declareds::= identifier
           |( identifier-commalist)
dimension-suffix::= ( attribute-argument-commalist)
attribute-argument::= expression
                    | *
attribute::= data-attribute
           | ALIGNED[(expression)]
           | BASED
           | CASE[( expression)]
           | CONSTANT
           | environment
           | initial
           | KEYED
           | PRINT
           | RECORD
           | SHARED
           | UNALIGNED
data-attribute::= AREA (area-content-commalist)
                | BIT[(attribute-argument)]
                | CHARACTER[(attribute-argument)]
                | DECIMAL( attribute-argument)
                | FILE
                | FIXED
                | FLOAT
                | POINTER
                | VALUES( value-literal-commalist)
                | type-reference
area-content::= (type-reference expression)
environment::= ENVIRONMENT ( environment-specification)
initial::= INITIAL( general-expression)

procedure-statement::= PROCEDURE [( parameter-name-commalist)]
                       [keyword-parameter-list]
                       [RETURNS ( type-reference)]
keyword-parameter::= identifier(parameter-name)
parameter-name::= identifier
module-statement::= [SYSTEM] MODULE
                    [EXPORTS (exported-name-commalist)];
exported-name::= unsubscripted-reference
do-statement::= DO;
              | DO while-option;
              | DO do-spec;
do-spec::= identifier [=spec]
spec::= expression to-by
to-by::=[ by-option] to-option
```

```
to-option::= TO expression
by-option::= BY expression
while-option::= WHILE (expression)
end-statement::= END [ identifier];
call-statement::= [CALL] identifier arguments [keyword-argument-list]
keyword-argument::= identifier(general-expression)
return-statement::= RETURN[(general-expression)];
goto-statement::= GO TO identifier
leave-statement::= LEAVE [identifier];
null-statement::=;
signal-statement::= SIGNAL condition-reference;
condition-reference::= computational-condition
                     | named-io-condition
                     | AREA[(identifier)]
                     | ERROR
                     | identifier
named-io-condition::= io-condition[(identifier)]
io-condition::= ENDFILE
              | KEY
              | RECORD
              | TRANSMIT
              | UNDEFINEDFILE
stop-statement::= STOP
assignment-statement::= reference= general-expression
                                 | concatenation;
allocate-statement::= ALLOCATE allocation;
allocation::= [identifier] set-option [in-option] [initial]
set-option::= SET( reference)
free-statement::= FREE freeing;
freeing::= locator-qualifier [identifier][in-option]
in-option::= IN( identifier)

open-statement::= OPEN single-opening;
single-opening::= file-option[ linesize-option]
                  [INPUT|OUTPUT|UPDATE]
file-option::= FILE( reference)
linesize-option::= LINESIZE( expression)
close-statement::= CLOSE file-option;
read-statement::= READ file-opton into-option[ key-option]
into-option::= INTO( reference)
key-option::= KEY( expression)
rewrite-statement::= REWRITE file-option from-option[ key-option]
write-statement::= WRITE file-option from-option[ keyfrom-option]
from-option::= FROM( reference)
keyfrom-option::= KEYFROM( expression)
get-statement::= GET get-file
get-file::=[ file-option][ skip-option][ input-specification]
skip-option::= SKIP[( expression)]
put-statement::= PUT put-file
put-file::=[ file-option][ skip-option]
           [ output-specification]
input-specification::= edit-directed-input
edit-directed-input::= EDIT edit-input-pair
edit-input-pair::=( input-target-commalist)
                  ( format-specification-commalist)
```

```
input-target::= reference
output-specification::= list-directed-output
                      | edit-directed-output
list-directed-output::= LIST( output-source-commalist)
edit-directed-output::= EDIT edit-output-pair
edit-output-pair::=( output-source-commalist)
                   ( format-specfication-commalist)
output-source::= expression
format-specification::= format-item
                      | format-iteration
format-iteration::= format-iteration-factor format-item
format-iteration-factor::= integer
format-item::= data-format
             | control-format
data-format::= real-format
             | picture-format
             | string-format
real-format::= fixed-point-format
             | floating-point-format
fixed-point-format::= F( expression[, expression])
floating-point-format::= E( expression[, expression])
picture-format::= P picture
string-format::= character-format
               | bit-format
character-format::= A[( expression)]
bit-format::= B[( expression)]
control-format::= space-format
                | skip-format
                | column-format
                | PAGE
line-format::= LINE( expression)
space-format::= X( expression)
skip-format::= SKIP[( expression)]
column-format::= COLUMN( expression)

general-expression::= expression
                    | [type-reference] (component-commalist)
component::= [{identifier|expression}:] general-expression
expression::= expression-four
              [level-four-operator expression-four]
            | expression-one user-operator expression-one
user-operator::= identifier
concatenation::= expression-one [|| concatenation]
level-four-operator::= & | ⊥ | &&
expression-four::= expression-three
                   [comparison-operator expression-three]
comparison-operator::= = | ¬= | ]= | [ | ] | [
expression-three::= expression-two
                    [level-two-operator expression-three]
level-two-operator::= + | -
expression-two::= expression-one
                  [level-one-operator expression-one]
level-one-operator::= * | /
```

```
expression-one::= reference
                | literal
                | prefix-operator expression-one
                | (expression)
prefix-operator::= + | - | ¬
reference::= unargumented-reference [arguments]
           | UNSPEC(reference)
unargumented-reference::= locator-qualifier [basic-reference]
                        | basic-reference
locator-qualifier::= reference ->
arguments::= (general-expression-commalist)
basic-reference::= [structure-qualification] identifier
structure-qualification::= basic reference [arguments]
unsubscripted-reference::= [unsubscripted reference.] identifier
literal::= arithmetic-literal
         | string-literal
         | values-literal
```

2.1 INTRODUCTION

When a program executes, storage is allocated to represent objects. These objects may be modelling the 'real-world', as for example when the object captures the relevant characteristics of a person, or they may be objects that are required solely for the implementation, such as a catalog of procedures.

In either case the program may need to refer to the alternative values that an object can have during execution. These references are called literal constants. For one object they might be the familiar 1 2 3 4 5 etc. For another they might be COLD WARM HOT.

Identifiers (sequences of letters,digits and underscores starting with a letter) are also used to reference the objects themselves, for example to test the current value of object WEIGHT during execution:

```
IF WEIGHT=120 THEN etc.
```

There are usually many more objects during execution of a program than there are unique identifiers in the written form of the program, so they cannot have distinct identifiers. Their are various ways that an identifier may represent several objects.

(1) The identifier may have different meanings at different places in the program. This is useful, for example, when several programmers each write part of a program. The rules for associating an identifier with the declaration of its meaning are described later in this introduction.

(2) An array of similar objects may be referenced by one identifier. The objects within the array are distinquished by implicit names, these names being the values of some other object. Thus two identifiers, referencing an array and a suitable object, can form a reference to a particular object in an array. This is written with the latter in brackets, e.g. A(J) when the execution time value of J selects an object within A.

(3) An ordered collection of objects, called a dataset, may be created. Another object, a file, can reference one object of a dataset at one time. There are explicit operations to associate a file with a dataset and to position the file to access the N'th object of the dataset. Also, accessing the N'th object positions the file ready to access the N+1'th object. c.f. Input/Output.

(4) An object may be given a name dynamically when it is allocated, and this name can become the value of another object. Such a reference is written with an arrow, e.g. P-> is a reference to the object whose name is the current value of P.

(5) An identifier may reference the most recently allocated object in a stack of objects. This only happens within recursive procedures q.v.

(6) A piece of program may be being executed more than once at one time, e.g. because two CPUs are at work.(c.f. tasking) The distinct tasks will each allocate objects from their own storage resources, so a reference may be implicitly qualified by a task.

In order to allocate appropriate storage for a an object, the computing system needs to know the values it may be called upon to hold. These are written in a list.

```
  DECLARE BATH_TEMPERATURE VALUES(COLD,WARM,HOT);
```

If the object is to contain the values of more than one characteristic it can be declared as a structure. This is written with numbers to indicate the containment.

```
  DECLARE 1 PRESIDENT,
              2 SEX VALUES(MALE,FEMALE),
              2 RELIGION VALUES(HINDU,CATHOLIC,OTHER);
```

The object PRESIDENT can take on all combinations of a value from the first list and a value from the second list. The objects within a structure can be referenced by postfixing their name within the structure to a reference to the structure. This is written with a period, e.g.

```
    PRESIDENT.RELIGION
```

If several similar objects are to be allocated it will be appropriate for the program to seperate the description of values from the operation of allocation. The program will first describe a template for the objects and then reference the template when allocating the objects. A template declaration is written like the declaration of an object, with an extra word BASED.

```
   DECLARE 1 BOOK BASED,
               2 AUTHOR VALUES(BLYTON,DICKENS,OTHER),
               2 COVER VALUES(HARDCOPY,PAPERBACK);
   DECLARE HISBOOK BOOK;
   DECLARE MYBOOK  BOOK;
   DECLARE HERBOOK BOOK;
```

This method of declaring objects is more than just a shorthand. The specification of what operations are meaningful in the program makes use of the template name, e.g. the program may define an operation EXCHANGE that operates on two BOOK objects. Thus the template name

controls the behaviour of the object, and it is called the data type of the object.

Some types are so widely useful that their declarations are provided by the language translator, and not written in the program. These 'builtin' types include a FIXED type - FIXED objects take numeric values and are subject to arithmetic operations.

There are three operations that apply to all objects - allocate, free, and assign. Allocate makes an object which can then be the subject of other operations. Free withdraws this availability. Assign sets the object to one of its possible values. The value is specified either as a literal constant or as the value of a referenced object. The written form of assignment has the source value on the right, e.g.

```
LENGTH=120;
A(J)=B(J);
```

For arrays and structures assignment implies assignment to all the contained objects.

For the builtin data types some further operations are builtin, i.e. predefined. Other operations may be defined by a particular program. The use of an operation can be written in various ways e.g.

```
B>C
SQUARE_ROOT(G)
H MODULO 6
CALL ABC;
UPDATE_PAYROLL(A1,A2);
READ FILE(F) INTO(Y);
```

The ingredients of an operation are an operator (like READ, SQUARE_ROOT,>) and usually some operands (references to objects or literal constants, like B, C and 6).

Operations are either function operations or subroutine operations. Function operations allocate a new object and give it a value - the type of this object is defined as part of the definition of the operation. Subroutine operations do not have this result object. Subroutine operations are written with a semicolon after them.

Sequences of function operators may be written, forming an expression. The result from one operation may be the operand of a subsequent one. Such operands are normally written with parentheses around the prior operation, e.g (B>C) but there are rules for particular operators which allow the parentheses to be omitted.

```
J+K-1
SQUARE_ROOT(G MOD 6)
(N+1)*(N+2)
```

Sequences of subroutine operations are written in the sequence they are to be executed, e.g.

```
READ FILE(F) INTO(Y);
YY=Y;
PUT FILE(OUT) LIST(YY);
```

A subroutine operation may be executed conditionally on the value of an expression being TRUE. This is written with an IF clause preceeding the statement, e.g.

```
IF B>C THEN OPEN FILE(F);
```

A sequence of subroutine operations can be grouped to form a single executable statement. This is written with a do-statement and an end-statement enclosing the sequence.

```
IF B>C THEN DO;
             J=2;
             K=2;
            END;
```

There are ways of writing a choice between two alternative actions, or several actions.

```
IF B>C THEN OPEN FILE(F);
       ELSE OPEN FILE(G);

        SELECT(HUE);
WHEN(PINK) OPEN FILE(F);
WHEN(MAUVE,ORANGE) OPEN FILE(G);
WHEN(COBALT);
OTHERWISE J=2;
        END;
```

A group may be executed repeatedly. If the group is to be repeated while some expression remains true a WHILE option is written after the DO.

```
DO WHILE(J>10);
  PUT FILE(OUT) LIST(A(J));
  J=J/2;
END;
```

There are forms for repeating the group with all possible values of some object, and for repeating with an arithmetic sequence of numbers.

```
DO BATH_TEMPERATURE;
  .
  .
END;

DO J=2 TO N BY 3;
  .
  .
END;
```

Execution of a group can end prematurely, by execution of a leave-statement. This references the name of the first statement of the group being terminated.

```
ALPHA:DO;
        .
        .
        IF B>C THEN LEAVE ALPHA;
        .
        .
        END;
```

A particular order of execution of statements is specified with the goto-statement. This references the name of a statement to be executed next.

```
      .
      .
    IF B>C THEN GO TO BETA;
      .
      .
   BETA: J=3;
      .
```

Statements can be formed into a procedure by enclosing them between a procedure-statement and an end-statement. The procedure-statement must be given a name and reference to this name causes the statements of the procedure to be executed.

```
OPENBOTH:PROCEDURE;
            OPEN FILE(F);
            OPEN FILE(G);
         END;
           .
           .
        CALL OPENBOTH;
           .
```

Execution of a procedure can also finish by the execution of a return-statement.

Objects declared in a procedure are implicitly allocated when the procedure is entered and freed when it finishes. The declare-statements are written below the procedure statement.

```
SUMSQ:PROCEDURE;
        DECLARE J FIXED DECIMAL(2);/* Fixed decimal is builtin. */
        X=0; /* X and A are declared elsewhere. */
        DO J=1 TO 50;
          X=X+A(J)*A(J);
        END;
      END SUMSQ;
```

The objects allocated by the procedure may be given values at the same time as they are allocated, in two ways. An expression may be given as part of the declaration, to be evaluated and the result assigned to the new object.

```
DECLARE J FIXED DECIMAL(2) INIITIAL(-3);
```

Alternatively the object can be made a parameter, by writing its name in the parameter list of the procedure statement.

```
SWAP:PROCEDURE(BOOK1,BOOK2);
      DECLARE BOOK1 BOOK;
      DECLARE BOOK2 BOOK;
      DECLARE SPARE BOOK;
      SPARE=BOOK1;
      BOOK1=BOOK2;
      BOOK2=SPARE;
    END;
```

When a procedure with parameters is referenced, initial values for the parameters are given with the reference, usually in the form of a list.

```
SWAP(MYBOOK,HISBOOK);
```

These arguments are paired with the parameters by position in the the lists, and assignments made from the first argument to the first parameter etc. When the procedure finishes assignments are made in the reverse direction, from the first parameter to the first argument etc. Of course in many cases the language translator can tell that these theoretical assignments would have no effect, and hence need not be made.

Procedures are the mechanism for defining operations. The name is the operator and the parameter declarations determine the number of arguments to the operator, and also the allowable types of the arguments because assignment is only allowed between similar data types.

The presence of a returns-option on a procedure statement specifies that it is a function operator rather than a subroutine. It specifies the type of the object that is to be the result of the operation. The procedure must finish by executing a return-statement causing the object to be allocated and assigned the value specified by the return-statement.

```
PROOF_READ:PROCEDURE(BOOKIN) RETURNS(BOOK);
             DECLARE BOOKIN BOOK;
             DECLARE BOOKOUT BOOK;
               .
               .
             RETURN(BOOKOUT);
           END;
```

Below the declare-statements in a procedure on-statements may be written. Each on-statement specifies a condition-name and a statement-name. If a signal-statement naming the same condition is executed in the procedure then execution continues at the named statement.

```
       ON TROUBLE GO TO FIXUP;
         .
         .
       SIGNAL TROUBLE;
         .
         .
   FIXUP: N=N+1;
         .
```

If a signal-statement is executed in a procedure that has no on-statement relating to the specified condition, then the procedure is prematurely finished, and further execution is as if the signal occurred in the procedure that invoked this procedure. This mechanism allows a procedure to specify what is to happen in the event of conditions being signalled by procedures that it invokes.

Some conditions are builtin, for example the ZERODIVIDE condition will be signalled by the builtin divide operation if the divisor has value zero. The SUBSCRIPTRANGE condition will be signalled at a reference to an array element if the value of the object used to specify the element does not name an element in the array.

The programmer may assert that the flow of his program execution is such that certain conditions will not be signalled. Assertions may help the compiler to compile more efficient code. To assert that a condition will not be signalled it is written ahead of the PROCEDURE word, with the letters NO before it.

```
  SUMSQ:(NOZERODIVIDE):PROCEDURE;
```

When large programs, or large collections of programs, have to be written it becomes vital to have rules about how programmers can use the same identifier with different meanings, can reference statements written by another programmer, etc.

The procedure provides some of this mechanism. It allows a sequence of statements to be used indirectly, by referencing the procedure name. Names declared in the procedure can be referenced only in the statements that comprise the procedure. This means that the same identifier can be used in different procedures without ambiquity.

```
SUMA:PROCEDURE;
       DECLARE ACCUMULATOR FIXED DECIMAL(8);
         .
         .
     END;
SUMB:PROCEDURE;
       DECLARE ACCUMULATOR FLOAT DECIMAL(10);
         .
         .
     END;
```

If the procedures are nested the potential ambiquity is resolved in favour of the the most closely containing procedure.

```
SUMA:PROCEDURE;
       DECLARE ACCUMULATOR FIXED DECIMAL(8);
         .
         .
       CALL SUMB;
         .
    SUMB:PROCEDURE;
           DECLARE ACCUMULATOR FLOAT DECIMAL(10);
           ACCUMULATOR=0;/* Refers to declaration in SUMB */
             .
             .
         END SUMB;
      END SUMA;
```

The names of statements and procedures are declared by their appearence at the left of a statement, with a colon following them. This means they cannot be referenced outside the procedure that contains them. (A procedure does not contain its own procedure name - the containing procedure does.)

A procedure may reference a declaration in a surrounding procedure. (There are restrictions on what the goto-statement may reference.)

Note that the rules about where in the program a declaration can be referenced are consistent with the execution time effect of allocating the objects for a procedure when it is entered and freeing them when it is left. The references can only be executed when the objects are available.

The procedure is a blunt instrument for controlling the resolution of identifiers. Putting a procedure-statement and an end-statement around a section of code prevents any of the enclosed declarations being referenced from outside. The module is a more selective method - the module-statement lists the identifiers for declarations that are not to be hidden by the enclosure.

```
MODULE;
  EXPORTS(A,SQRT);
  DECLARE A FIXED DECIMAL(3);
  DECLARE B FIXED DECIMAL(3);/* Cannot be referenced from outside */
  SQRT:PROCEDURE;
       .
       .
       END;
END; /* Of the module */
```

Enclosing declarations in a module does not affect the allocation of objects - they are allocated when the containing procedure is entered.

There are some refinements to the rules above for associating a reference with a declaration. In the case of operations, i.e. procedure names, the basic rules above might lead to an identifier being resolved to a procedure that was clearly the wrong one, because it had the wrong number or type of parameters.

```
SWAP:PROCEDURE(BOOK1,BOOK2);
     .
     .
     END;
     .
  CALL SWAP(A,B,C);/* Note three arguments */
```

In this case the putative association is rejected, and there must be another procedure with the same identifier and appropriate parameters that the reference actually resolves to.

Some of the builtin operators are implemented as more than one procedure.

```
DECLARE J FIXED DECIMAL(3);
DECLARE X FLOAT DECIMAL(8);
  J=-J;
  X=-X;/* Same operator,but different type of argument */
```

The combination of these rules allows a programmer to define a module which provides operations accessible over a wide area of a program without making accessible the details of how the operations are implemented. c.f. the examples.

Operations may be combined into a module because they have some common factor, other than all being used in the same program. For example one module might be defined to contain a wide variety of trigonometric operations even though no single program was expected to use them all. Similarly the procedures being maintained by a particular programmer might be grouped together. Such large groupings are not usually executed as a complete unit, instead the operations they make available are referenced in other programs.

The dot notation that is used for referencing objects in structures, e.g. PRESIDENT.RELIGION, can also be used with a procedure reference

to the left of the dot. This says that the name to the right is to be resolved as if it appeared in the referenced procedure. This sort of reference is used when editing the written form of a program.

```
A:PROCEDURE;
   .
   .
   B:PROCEDURE;
     .
     .
     END;
   .
  END A;
DELETE A.B;/* Deletes the procedure from the written program */
```

The language system editor is used by the programmer to develop and modify sections of program. This editing references sections of program by their names, and may also use numbering of the individual lines in a section.

The editor is a builtin operation, and there are other builtin operations for translating and executing sections of program. The translator checks for errors that can be found without execution and creates a version of the section in an internal form. The existence of this internal form will economise on the translating necessary before the section can be executed on a subsequent occasion. The system keeps track of whether an internal form is obscleted by editing of the written section that it corresponds to.

The arguments to the system services like editing allow the programmer to control some details of their operation, e.g. whether the translator is to produce a cross-reference of where identifiers are declared and used.

There are two circumstances where the language does not fully define what happens during execution. The first is if some assertion in the program is incorrect. The second is if SYSTEM modules are used. Inside SYSTEM modules code can be written that is peculiar to the hardware that the program is going to execute on, and certain constructions can be used that might lead to errors that the language system cannot detect.

2.2 THE PHYSICAL COMPONENTS OF A PROGRAM

A standard representation for programs is defined so that the programs can be processed by different machines. This representation uses the 64 character subset of ASCII. The characters are formed into lines. Each line consists of 72 characters of user program and an 8 character line identification. The latter consists of a 5 digit decimal line number and a 3 character user field. (Which might be the authors initials or a modification level for example.) The formation of lines into datasets is described in an appendix.

2.3 DELIMITERS AND NON-DELIMITERS.

Here we describe the most basic constructions of the language.

The method of describing syntax is BNF-like, and follows the method used in the definition of Standard PL/I, except that overstruck characters are not used, angle brackets replace square brackets and curly brackets are avoided by the use of extra productions. Where there might be ambiquity quotes are used, e.g. '|' does not mean syntactic alternative, while | does.

```
text::=<delimiter-list><delimiter-pair-list>
delimiter-pair::=non-delimiter delimiter-list
delimiter::=operator-symbol|punctuation-symbol
operator-symbol::=+|-|*|/|**|>|<|=|>=|<=|¬=|¬|&|&&|'|'|'||'
punctuation-symbol::=(|)|.|,|;|:|->|?|blank|comment
non-delimiter::=identifier|arithmetic-constant|string-constant
identifier::=letter|identifier identifier-character
identifier-character::=letter|digit|_
arithmetic-constant::=decimal-number<E<sign>integer>
decimal-number::=integer<.digit-list>|.digit-list
sign::=+|-
string-constant::='<string-symbol-list>'
```

A comment begins with /* and ends with */ and any characters may appear between these except the consecutive pair */

A string-symbol may be two consecutive characters '' or any character other than quote.

Extra rules are imposed to make programs easier to read. The end of a line of program is treated as a blank character. This prevents most delimiters and non-delimiters from continuing across lines. Also string constants must not continue across lines.

There are some punctuations of the language which look like identifiers but are not the names of anything:

BASED
CASE
DECLARE
DO
END
IF
INITIAL
MODULE
PROCEDURE
PRIORITY
RETURNS
SELECT
SUBSTR
SYSTEM
THEN
UNSPEC
VALUES
WHEN
WHILE

2.4 IDENTIFIERS AND CONSTANTS

```
values::=VALUES(constant-commalist)
constant::=arithmetic-constant|string-constant|identifier
statement-name::=identifier:
```

In addition to the arithmetic and string constants, some identifiers represent constants. These identifiers are defined in a list of values, or they appear on the left of statements.

The builtin constants are:

NULL
TRUE
FALSE

Where the language specifies an opening round bracket and matching left bracket, square brackets may be used instead. When there are multiple brackets this will make programs clearer.

2.5 OBJECTS, INCLUDING THOSE THAT CONTAIN OTHERS.

```
declare-statement::=DECLARE declaration-commalist;
declaration::=<integer> identifier <BASED> <dimension-suffix>
              <attribute-list><SHARED><CASE<(expression)>>
```

The variations in the form of a declaration can specify attributes about the containment of objects. These attributes are known as Structure, Member, Array, Union and Augment. DATA sets and AREAs can also contain objects, and what they contain is determined by the execution of the program.

2.5.1 STRUCTURE AND MEMBER.

An integer at the start of the declaration specifies that the identifier is a structure or a member. The structure and its members are written as a sequence of declarations. The integer on the structure is one less than the integer on its immediate members. This notation can be nested, so that an identifier is both member and structure.

e.g.

```
1 journey,
  2 start,
    3 latitude degrees,
    3 longitude degrees,
  2 finish,
    3 latitude degrees,
    3 longitude degrees;
```

The advantage of forming a structure is that the complete structure can be operated on, with the detailed effect on the members only specified where necessary.

2.5.2 ARRAYS

An array is composed of a number of identical objects. The names of these identical objects are the values of some other data type. This other data type is specified in the dimension suffix.

```
DECLARE HOURS(WEEKDAY) INTEGER;/* HOURS(MONDAY),HOURS(TUESDAY) etc.*/
```

In the case where the names are the integers 1 to N, an expression for the value of N is given in the dimension suffix. The value may be zero, indicating there are no objects in the array.

```
DECLARE TRANSITION(6) FLOAT;/* TRANSITION(1) ... TRANSITION(6) */
```

One dimensional arrays are also known as strings.

2.5.3 UNIONS

The keyword CASE indicates that a union structure is being defined. There are two possibilities. 1. All the members of the structure have CASE without a following argument. This is called an augmented union. 2. All the members of the structure have CASE with a following argument, except the first, which does not have CASE. In the former construction, the members of the structure do not combine to comprise the structure but are alternatives, only one of which is held in the structure at any one time. Assignment to a member of a union makes that member the current member, destroying any object previously in the structure. The structure also contains an un-named object with a value that indicates which member is current in the structure. This augment value is used when the program specifies the structure as a source of a value, to check that the current member is the one specified.

```
DECLARE 1 NUMBER,
          2 X FIXED DECIMAL(6) CASE,
          2 Y FLOAT DECIMAL(8) CASE;
    NUMBER.X=27;/* Makes X the current member */
    YY=NUMBER.Y;/* An error, since NUMBER does not currently
                                           contain Y. */
```

In the second form above the determination of the current member is contained within the structure. The members, other than the first, are alternatives. When the structure is used, the CASE arguments are evaluated in order of appearence and the first TRUE one indicates that the corresponding member is currently the only member of the structure. To ensure that this indication is consistent with the asignments to the structures members, the following restriction is made, (outside of SYSTEM modules). The CASE expressions must be constant over the lifetime of an instance of the structure. This means that the only member that can become current in an instance is determined when the instance is allocated.

```
DECLARE 1 PERSON BASED,
          2 SEX VALUES(MALE,FEMALE) CONSTANT,
          2 MAIDEN_NAME CHARACTER(12) CASE(SEX=FEMALE),
          2 MILITARY_SERVICE BIT(1)   CASE(SEX=MALE);
DECLARE MARY PERSON CONSTANT INITIAL(FEMALE,'SMITH');
DECLARE JACK PERSON CONSTANT INITIAL(MALE,FALSE);
```

2.5.4 SHARED

The SHARE keyword describes another situation where the written declaration is augmented by an un-named augment object. Even if there is no level number, a structure is implied.

```
DECLARE B BUFFER SHARE;/* Declares B and an augment. */
```

The augment is used to perform synchronisation when the shared object is accessed from different processes.

2.5.5 AUGMENTED ARRAYS

An array with an asterisk in the dimension suffix is augmented with an object, (of type FIXED) containing a count of the objects within the array.

2.6 ARRAYS OF STRUCTURES ETC.

In general, a composed object can itself be part of a bigger object.

```
 DECLARE 1 A(10),/* An array of structures. */
           2 A1 FLOAT,
           2 A2 FLOAT;
```

The restrictions are:

1. A dataset can only contain, it cannot be contained.
2. An area can only contain, it cannot be contained.
3. A SHAREd object must not be contained in a SHAREd object.

An array of arrays may be written with the sizes listed in the dimension-suffix e.g. DECLARE RECTILINEAR(8,9) INTEGER;

```
 dimension-suffix::=( attribute-argument-commalist )
```

2.7 ATTRIBUTES ACQUIRED INDIRECTLY

```
 attribute::=identifier<(attribute-argument-commalist)>
 attribute-argument::=expression|*|?|attribute-list
```

The identifier of an attribute may be declared. If it is not declared then it will be a builtin data type. If it is declared then the BASED keyword will be used in its declaration.

```
DECLARE X COUNTER;
DECLARE COUNTER BASED FIXED DECIMAL(4);
```

Attributes which are referenced by declared name in this way are expanded out, i.e. the item being declared acquires the named attribute and all the attributes declared for the named attribute. (In the example X has attributes FIXED DECIMAL(4) and COUNTER.) All attributes can be acquired in this way. When structuring is acquired the member names are also acquired.

The attributes being added may have asterisk or question-mark in their arguments. If so, the reference to the declared attribute must have just enough arguments to match one-to-one with these asterisks or question-marks.(The association is by order of appearence.) In the case of a question-mark the argument on the reference replaces the question-mark in the attribute being added. In the case of an asterisk the argument becomes the value of the un-named object defined by the asterisk.

```
DECLARE VECTOR BASED (*) FLOAT DECIMAL(?);
DECLARE V VECTOR(10,6);/* Attributes (*) FLOAT DECIMAL(6) with *=10 */
```

The attribute 'parameter' is acquired by the appearence of the identifier in the parameter-list of a procedure statement.

2.8 ATTRIBUTES

The full set of attributes for a name that is not declared with BASED must meet certain requirements.

1. If the attributes include neither array nor structure, they must include just one set of values.(This may be written explicitly as the attribute VALUES(value-commalist) or it may be implied by a builtin attribute.)

2. There must be no question-marks remaining.

3. There must be no asterisks, unless there is also 'parameter'.

2.9 THE BUILTIN ATTRIBUTES.

The attribute INITIAL(expression) means that the object is set to the value of the expression at the beginning of its lifetime. It can appear at most once in a full set of attributes. If it appears on a structure declaration it must not also appear on anything contained in the structure.

The CONSTANT attribute means that the only assignment to the object occurs at the beginning of its lifetime.

The values of the FIXED data type are an arithmetic sequence of numbers. The values of FLOAT are approximate numbers.

The DECIMAL(precision-expression,scale-expression) attribute gives further specification of a FIXED or FLOAT object. The precision P and scale Q give the values for a FIXED type. The values are strictly between plus or minus 10**(P-Q) with an interval of 10**(-Q). A fixed value is written with P-Q digits before the decimal point and Q digits after.

```
00.625  /* FIXED DECIMAL(5,3) */
```

For FLOAT data the scale is omitted. The precision indicates that the ratio of the largest to smallest of the values is less than 10**P. A FLOAT value is written with P digits in the mantissa and E before the exponent.

```
9.84E1  /* FLOAT DECIMAL(3), an approximation to 98.4 */
```

BIT(attribute-argument) is an array of Boolean values, i.e. there is an implicit declaration BIT BASED (*) Boolean; DECLARE Boolean VALUES(FALSE,TRUE); The attribute Boolean is also builtin but is written as BIT, (without an argument). Thus BIT(10) and (10) BIT are the same.

CHARACTER(attribute-argument) is an array of character codes. The implicit declaration is CHARACTER BASED (*) Codes; DECLARE Codes VALUES(SP, HT,... '(', ')', '*',... 'A', 'B',... '0', '1',... DEL); (These are the ASCII codes. See 'Machine Specification' for how to specify that the hardware works in another set of codes.)

The AREA(attribute-list) attribute specifies a storage area from which objects can be allocated.

The DATA(attribute-list) attribute specifies a special storage area in which objects can be allocated, a dataset. Objects in a data set cannot be referenced directly, but can be copied. See Input/Output.

The values of POINTER objects are names of objects allocated in an AREA.

The TO(identifier) attribute specifies that a pointer object has values that name objects of the specified type.

The IN(identifier) attribute specifies that a pointer object has values that name objects allocated in the specified area.

A FILE object has two states, open and closed. When open its values are the names of objects within a dataset.

The RECORD, KEYED, PRINT, INPUT, OUTPUT, and UPDATE attributes specify which forms of Input/Output statements may be used with the file object as an argument.

A CONDITION object is used only as the subject of the SIGNAL statement and in condition prefixes.

2.10 RESOLUTION AND MODULES.

```
procedure::=statement-name <prefix> procedure-statement
            < declare-statement-list > < on-statement-list >
            < executable-list > < procedure-list >
            ending
executable::=< statement-name > < prefix > executable-statement
executable-statement::=module|group|if-statement|
                       executable-single-statement
module::=module-statement < declare-statement-list >
         < executable-list > < procedure-list > ending
module-statement::=< SYSTEM > MODULE
                   < EXPORT(resolution-item-commalist) >;
resolution-item::=name|operator-symbol
name::=identifier|identifier.name
```

The most significant constructs in associating references with declarations are procedures and modules. All the rules for finding the declaration corresponding to a name are applied first in the procedure or module that most closely contains the use of the name. Only if there is no matching declaration is the next most closely containing procedure or module considered, and so on. It is an error if none of the containing procedures and modules has a matching declaration. The builtin identifiers are declared in a notional outermost procedure, so that they can be used in all procedures and modules. The rules for resolution within an individual procedure or module are as follows.

A list is made of the declarations in the procedure or module, indicating the identifier declared and its attributes. This is called the catalog. Declarations stem either from declare-statements or from statement-names. The attributes of a statement name on a procedure include the number and attributes of the parameters to the procedure. The catalog includes the names in any module statements immediately contained in the procedure or module.

There are constraints on the identifiers in the catalog. Identifiers which are not declared as procedure names or members of a structure must be unique within the catalog. The identifiers of members of a structure must be unique within the structure that immediately contains them. Two procedure names that are the same must have attributes that differ in the number or type of parameters. These rules ensure that the process described below leads to a single declaration.

When looking up a name, items from the catalogue are selected which have the same name. In the case when the name is a sequence of identifiers with dots between, each identifier must correspond to a member such that all the identifiers to the left of it name structures that contain that member, directly or indirectly. This subset of the catalog may be empty, may be one declaration, or may be several declarations.

If it contains one declaration then this is the declaration being sought. If it contains more than one declaration then all the declarations must be for procedure names, and generic selection, q.v., is used in an attempt to select one of them. If it is empty then an attempt is made to resolve the name by 'indirect resolution' q.v. If these attempts fail the name cannot be associated with a declaration in this catalog.

2.10.1 GENERIC SELECTION

Generic selection depends on the notion of type. When gathering the attributes for an object a distinction is made on how a referenced attribute is known to the declaration making use of it. It may be known because the reference resolves directly to a declaration, as in the earlier examples. Or it may be known only because the name is in the list of a module-statement:

```
   DECLARE F SMALL_FIXED INITIAL(0);
   MODULE(SMALL_FIXED,ALPHA);
   DECLARE SMALL_FIXED BASED FIXED DECIMAL(2);
   .
   .
   END;
```

The attributes acquired in the former way, with INITIAL and CONSTANT discounted, are called the type of the declared item.

```
 /* F has type SMALL_FIXED and attributes SMALL_FIXED, INITIAL(0),
    FIXED, DECIMAL(2) */
```

Generic selection starts with a list of procedure-names, as described above, and selects one or none of them. The selection is made on the basis of the arguments on the reference to the procedure name. (The reference will usually be written with the arguments in a list, like MAX(A,B), but can be any form of operation like A+B or A MOD B.)

A first selection is done by matching the type and number of the arguments with the type and number of the parameters for each of the procedures. The argument and parameter types match if the former has all the attributes comprising the latter. This includes whether they are both structures with the same member names, whether they are both arrays covering the same names, and whether both have asterisk arguments. INITIAL and CONSTANT are not included in the test for equality. If there is a procedure with appropriate arguments this completes the selection, otherwise a second selection is made.

In the second selection a weaker test of matching is made. An argument matches a parameter if assignment between of an object of the argument type to an object of the parameter type is defined in the language. c.f. assignment. If more than one procedure matches under this weaker test it is an error. If no procedure matches the selection fails. Otherwise a procedure has been selected.

2.10.2 INDIRECT RESOLUTION

Indirect resolution operates only on a sequence of identifiers with dots between them. The leftmost must resolve to a procedure or module. The catalog of this procedure or module is then used to resolve the remainder of the sequence. This latter resolution may again be indirect. Indirect resolution is only permitted in certain circumstances, e.g. to reference a piece of program to be edited.

2.11 REFERENCING

```
reference::= unargumented-reference < arguments >
unargumented-reference::= locator-qualifier < basic-reference >
                        | basic-reference
locator-qualifier::= reference ->
arguments::=(<expression-commalist>)
basic-reference::=< structure-qualification > identifier
structure-qualification::= basic-reference <arguments>.
```

The resolution of names to declarations occurs before program execution - it does not depend on the values of any objects. (There is an exception in the area of Input/Output) Here we describe how different objects can be referenced during execution.

A declaration that is not BASED relates to an object allocated automatically when the procedure that contains the declaration is entered. The identifier from the declaration is used to reference the object.

```
BATH_TEMPERATURE
```

If instead an object has been allocated explicitly,c.f. 'Allocation and freeing', then there will be a POINTER object whose value selects the object. Referencing the object is written with an arrow.

```
P->CELL
```

When an array or structure object is referenced, the reference may be augmented to indicate that only a component of the object is being referenced. For an array the object whose value selects the component, the subscript, is written after the array reference. In the case of a structure the member name is used.

```
TAX_RATE(N)    PRESIDENT.RELIGION
```

These selections can be applied successively, in left to right order.

```
Q->ALPHA(J).BETA
```

Successive selection on arrays is customarily written with an argument list rather than many brackets.

```
PLOT(R,S) /* Rather than PLOT(R)(S) */
```

References to the results of operations, which are un-named, are achieved by writing the operation and any operands,q.v.

Reference to an alternative in a structure will be an error, if the current member is not that alternative, and is not being changed to that alternative. This error raises the TYPE condition when the reference is executed. (Reference as the argument to the IS(argument) builtin function is an exception - the result of the function indicates whether the alternative is current.)

A section of an array can be viewed as another array. The form SUBSTR(array-reference,N,M) refers to the objects array-reference((N-1)+1), array-reference((N-1)+2), ... array-reference((N-1)+M)

The objects are renumbered, so that the section selected is an array with components named 1, 2, ... M.

2.12 FUNCTION OPERATIONS

Function operations are written in three ways. Where there is an operator-symbol that takes one argument the symbol is written ahead of the argument, e.g. -X For operator symbols that take two arguments and for procedure identifiers that take two arguments, the operator may be written between the arguments. e.g. B>C H MODULO 6 For procedure identifiers the arguments may also be listed after the identifier. Superficially this looks like subscripting, but the type of the identifier makes the distinction.

2.13 BUILTIN OPERATIONS

VALUEs are ordered, so that comparison of them is meaningful. Thus if the values are VALUES(COLD,WARM,HOT) it is true that HOT>COLD and that WARM=WARM. When objects are referenced it is their values that are compared. Thus immediately after an assignment T=HOT it is true that T>COLD and that T=HOT.

The meaning of comparison for builtin data types stems from their underlying VALUEs. Thus 5>3 and 'B'>'A'. Comparison is not defined where there are no values, as with AREA,FILE, and CONDITION.

Comparison may also be applied to strings, which means the objects of the strings are compared in pairs, left to right. If each object in one string is equal to the object in the corresponding position in the other string then the strings are equal. If they are not equal the leftmost unequal position determines which is greater. If one string is shorter than the other and hence does not provide enough values for comparison, then the smallest possible value is used as a substitute in these comparisons.

The result of a comparison is BIT(1), i.e. TRUE or FALSE.

The operations written '¬', '&', '|', and '&&', (without quotes) operate on Boolean values with the meanings NOT, AND, OR, and EXCLUSIVE-OR. When applied to BIT strings where the Boolean values represent inclusion/exclusion from a set, these give a result representing the inverse, intersection, union, or symmetric difference of the set(s). The operands should be of the same length and this determines the length of the result.

The operators written '-', '+', '*', '/', provide the arithmetic operations of subtraction (and negation when used with one operand), addition, multiplication and division. They yield the normal arithmetic results, e.g. 5+5=10, with the caveat that FLOAT values are approximations so that 0.4E0+0.6E0 may not equal 1.0E0.

The operands must be of the same numeric type, although they may differ in precision and scale. The precision and scale of the result are defined so that no accuracy is lost. For operations on float values this means the precision of the result is equal to the maximum of the operand precisions. For FIXED data it means there are enough digit positions for all digits of the result, after lining up the decimal points. More formally, if the operand types are FIXED DECIMAL(P1,Q1) and FIXED DECIMAL(P2,Q2) then the result of addition and subtraction is FIXED DECIMAL(1+MAX(P1-Q1,P2-Q2)+MAX(Q1,Q2), MAX(Q1,Q2)). The result of multiplication is FIXED DECIMAL(1+P1+P2,Q1+Q2). Division strictly needs indefinitely large scale in order to retain accuracy. Hence division is only permitted in contexts where these extra scale positions, if produced, would immediately be discarded. These contexts are ones where the result is about to be assigned to an object with known scale. For example FIXED division is allowed in the form A=B/C but not in the form

A=(B/C)*D. Note that this situation is an exception to the general rule that digits are not lost on assignment.

2.14 ORDERING WITHIN EXPRESSIONS

```
expression::= expression-four < level-four-operator expression-four >
            | expression-one  user-operator expression-one
level-four-operator::=&|'|"|&&
user-operator::= identifier
expression-four::=expression-three
                   < comparison-operator expression-three >
expression-three::= expression-two
                    < level-two-operator expression-three >
level-two-operator::= + | -
expression-two::=expression-one
                   < level-one-operator expression-one >
level-one-operator::= * | /
expression-one::= reference | literal
                    | prefix-operator expression-one
                    | (general-expression)
prefix-operator::= + | - | ¬
```

Sequences of function operators may be written, forming an expression. The result from one operation may be the operand of another operation. Such operands are normally written with parentheses around the earlier operation, e.g. (B+C).
There are rules for particular operators which allow the brackets to be omitted. These rules are chosen to simplify writing some common cases.
No brackets are necessary around prefix operators since they always apply to the expression on their right, e.g. A*-2 is the same as A*(-2). No brackets are necessary to ensure that Boolean operations are performed before comparisons, e.g. C|D=TRUE is the same as (C|D)=TRUE, not C|(D=TRUE). No brackets are necessary to ensure that arithmetic precedes comparison, e.g. A+B>10 is the same as (A+B)>10. (The alternative of A+(B>10) would not be meaningful anyway.) No brackets are necessary if multiplication/division precedes addition/subtraction, e.g. A+B*10 is the same as A+(B*10).

A sequence of additions and subtractions, or a sequence of concatenations, does not need brackets if the operations are to be evaluated from left to right. e.g. I+J-2 is the same as (I+J)-2

The order of evaluation of operators is obvious when the result of one operation is required as an operand of another. In very rare cases the order of evalution of the operands for a single operation will be significant, since these evaluations may be function operations with 'side-effects' additional to their main effect of returning a result. The order is left to right.

2.15 ASSIGNMENT OF OBJECTS WITH COMPONENTS.

```
assignment-statement::=reference = general-expression;
general-expression::=expression-item-commalist
                    | concatenation
expression-item::= expression | (expression-item-commalist)
concatenation::=expression < '||' concatenation >
```

Assignment uses the value of an expression, the source, to set the value of an object, the target. The details of this operation depend on the data types of the source and target. Note that it is the type and not the complete set of attributes which is relevant. If a module defines a type using an array then a single object of that type is not an array.

If the target is a structure then the assignment is expanded to be an assignment to each of its members. The source must be a structure with the same number of members, or a list with the required number of members. The component targets and the sources are paired by order of appearence.

```
DECLARE 1 PRESIDENT,
              2 SEX VALUES(MALE,FEMALE),
              2 RELIGION VALUES(HINDU,CATHOLIC,OTHER);

  PRESIDENT=FEMALE,HINDU;/* Has effect of PRESIDENT.SEX=FEMALE
                            PRESIDENT.RELIGION=HINDU */
```

Union structures may be assigned only if they are identical in type.

If the target is an array, then the source must be an array or a list of values. As for a structure, the assignment is expanded. However if there are more components in the target than the source the residual target components are set to their lowest possible value. (i.e. the leftmost in their VALUES list.)

By applying these expansion rules repeatedly if necessary all assignments are reduced to ones where there are no components to the source or target.

2.16 ASSIGNMENT OF SINGLE OBJECTS.

If the source and target are of identical type then the value of the source becomes the value of the target. Some other assignments between builtin types are meaningful.

Assignment of a FIXED value to a FLOAT object is permitted, since any loss of accuracy is not significant because the result is only used as an approximation. Assignment of FLOAT to FIXED is not permitted because there is no obvious interpretation of constructing an exact value from an approximation.

Assignment of FLOAT to FLOAT is permitted provided the precision of the target is at least that of the source, so that no accuracy is lost. Assignment of FIXED to FIXED requires the scale of the target to be at least that of the source, for the same reason.(If it is larger the extra digit positions are set to zero.)

In all assignments there is a condition that occurs if the value being assigned to the target is too large for the target to hold.

2.17 ORDER OF EXECUTION OF STATEMENTS.

```
group::=do-statement < executable-list > ending
       | select-statement < choice-list < default-choice >> ending
choice::= WHEN( expression-commalist ) executable
default-choice::= OTHERWISE executable
if-statement::= IF expression THEN executable < ELSE executable >
do-statement::= DO;
               | DO WHILE < expression >
               | DO do-spec;
do-spec::= identifier< =spec >
spec::= expresssion to-by
to-by::= TO expression < BY expression >
call-statement::= < CALL > reference-list;
return-statement::= RETURN < (general-expression) >;
goto-statement::=GO TO identifier;
leave-statement::= LEAVE <identifier>;
null-statement::= ;
stop-statement::=STOP;
```

<< In the Users Manual this would be an elaboration of the sequencing constructs IF, SELECT, etc described in the introduction. This should present no difficulties. Only iteration over a domain will be new to most programmers.>>

2.18 CONDITIONS

```
signal-statement::= SIGNAL condition-reference;
prefix::=( condition-reference-commalist ):
on-statement::= ON condition-reference goto-statement;
condition-reference::=identifier<(identifier)>
```

<< This User Manual requires an elaboration of what is in the Introduction. The only tricky bit is compile time distinction between ERROR and other conditions: >>

The condition ERROR is special. It represents the condition ERROR, and also all builtin conditions other than those with ON statements in the same procedure. Thus if the procedure has just ON ERROR GO TO LA; then SIGNAL SIZE or SIGNAL ERROR will transfer control to LA. If the procedure has ON SIZE GO TO LS; ON ERROR GO TO LA; then SIGNAL SIZE will transfer to LS, SIGNAL ERROR will transfer to LA, and SIGNAL OVERFLOW will transfer to LA.

<< The following list of builtin conditions is incomplete. The intention is to extend the list until there is no need for an error code. >>

AREA(reference)
: Signalled by ALLOCATE if there is insufficient free storage in the area.

CONVERSION
: Signalled by certain builtin functions, q.v.

ENDFILE(reference)
: Signalled by an input operation if the current value of the file has been so far advanced that there is no corresponding object in the dataset.

OVERFLOW
: Signalled by FLOAT arithmetic when a result exceeds the maximum value an implementation can support. This value can be determined by examining the machine description. See also builtin functions, which may signal this condition.

RECORD(reference)
: Signalled when the object in a dataset referenced by the specified file, does not match the type which the I/O statement specifies it should have.

SIZE
: Signalled when the value being assigned to a FIXED object is not a value that the object can hold.

STORAGE
: Signalled when there is insufficient storage remaining in the pool used for allocating objects automatically when a procedure starts.

STRINGSIZE
: Signalled when the source string of an assignment is longer than the target.

SUBSCRIPTRANGE
: Signalled if the value of a subscript is not the name of an element of the array.

UNDEFINEDFILE(reference)
: Signalled by OPEN if the TITLE argument does not name a dataset.

UNDERFLOW
: Signalled when the result of FLOAT arithmetic is too small for the machine to handle.

ZERODIVIDE
: Signalled when division by zero is attempted.

2.19 SUBROUTINE OPERATIONS

```
procedure-statement::=PROCEDURE<(parameter-name-commalist)>
                      < keyword-parameter-list >
                      < RETURNS( attribute-list )>;
parameter-name::=identifier
keyword-parameter::=identifier(parameter-name)
```

The arguments to subroutine operations are written in one or more argument lists. The procedure statement for the operation determines how the arguments should be written. The arguments written after the operator (the procedure name) are matched with the parameters listed after the word PROCEDURE. The arguments after keywords are matched with the parameters after the same keyword in the procedure statement. The order of appearence of keywords is not significant. e.g.

```
SORT:PROCEDURE(P1) USING(P2) LIKE(P3);...;END;
 SORT(A) LIKE(F) USING(Y);/* Matches A to P1, F to P3, Y to P2 */
```

Many of the builtin subroutine operations (executable single statements) do not follow this format, because they have an argument that is not an object (e.g. ALLOCATE) or because they are shorthands for several operations (e.g EDIT(A,B)(F(3))).

```
executable-single-statement::=allocate-statement
                            | assignment-statement
                            | call-statement
                            | close-statement
                            | free-statement
                            | get-statement
                            | goto-statement
                            | leave-statement
                            | null-statement
                            | open-statement
                            | put-statement
                            | read-statement
                            | return-statement
                            | rewrite-statement
                            | start-statement
                            | signal-statement
                            | write-statement
```

2.20 ALLOCATION AND FREEING

```
allocate-statement::= ALLOCATE identifier
                  < IN(identifier) > < SET(reference) > ;
free-statement::= FREE reference < IN(identifier) > ;
```

Allocation makes a new object available for use. Freeing asserts that the object is no longer in use, and the language system may reuse its storage.

Any expressions that determine array sizes or initial values are evaluated at the same time as the allocation is done, and the initial values are also set at this time.

Declarations without BASED are declarations of objects that are to be automatically allocated each time the containing procedure is entered and freed each time it is left. Notice that if the procedure is recursive, i.e. entered more than once without an intervening exit, there can be more than one object associated with the declaration. There is no way of referencing other than the latest object to be allocated.

These automatic allocations are made out of a storage area associated with the current process.

Explicit allocation and freeing are performed by the corresponding statements.

2.21 PARAMETERS

The procedure-statement defines which arguments of an operation correspond to which parameters. The parameter objects are declared like other objects and are automatically allocated and freed in the same way. (They cannot be allocated explicitly in areas.)

Initialization of parameters is not the same as for other objects. A parameter cannot have the INITIAL attribute and it is initialized by assigning the argument to it. Also the parameter's value is assigned back to the argument when the procedure terminates. These assignments will not occur if they are redundant, i.e if the parameter is known to have the same value as the argument at the finish of the procedure, or if the parameter is only used as a target in the procedure.

Parameters may have attributes with unmatched asterisk or query extents in them. These values are taken from the corresponding attribute arguments of the argument.

Objects returned by function operations are not declared or allocated like other objects. Their type is defined in the RETURNS clause of the procedure-statement. They are allocated and assigned to by the RETURN statement.

2.22 INPUT/OUTPUT

<< In this report we have not reproduced all the relevant details of PL/I I/O, simply enough to to show what subset is used.>>

Data that is typed by humans, or destined to be read by humans, will be held in character form. The values are written in the same way as values are written in a program.

```
276 HOT 9.84E1 '1'
```

There are additional ways in which data can be written. The format of any particular constant can be specified by a format item.

```
data-format::=fixed-point-format
             |floating-point-format
             |picture-format
             |character-format
             |bit-format
fixed-point-format::=F(expression < ,expression >)
floating-point-format::=E(expression < ,expression >)
picture-format::= P string-constant
character-format::=A<(expression)>
bit-format::=B<(expression)>
```

A format is written in conjunction with a reference to the object which is to provide, or receive, the value.e.g EDIT(ALPHA)(F(5)).

```
edit-directed-input::=EDIT edit-input-pair
edit-directed-output::= EDIT edit-output-pair
```

The data-item may be a structure, in which case it is expanded into its components. A list of data-items may be given, which are paired with the format items in order of appearance. If there are fewer format items then 'wrap-around' of the format list occurs. e.g. EDIT(I,J,K)(F(5)).

The word LIST may be used instead of EDIT, in which case appropriate formats will be deduced from the types of the data items.

```
edit-input-pair::=(reference-commalist)
                  (format-specification-commalist)
edit-output-pair::=(expression-commalist)
                   (format-specification-commalist)
list-directed-output::=LIST(expression-commalist)
output-specification::=list-directed-output
                      |edit-directed-output
```

There are also control format items. These have an effect without being paired to a data item.

```
control-format::= LINE(expression)
                | X(expression)
                | SKIP<(expression)>
                | COLUMN(expression)
```

When a PUT statement is executed the output-specification produces a sequence of characters, the human-readable form of the data values. When a GET statement is executed the input-specification consumes a sequence of characters.

```
put-statement::=PUT file-option
                < SKIP<(expression)> | PAGE< LINE(expression) >>
                <output-specification>;
get-statement::=GET file-option input-specification;
```

Each of these characters is transmitted by an implicit WRITE or READ operation to or from the dataset associated with the specified file.

The relation between a file object and a dataset is determined when the file is opened.

```
open-statement::= OPEN file-option <TITLE(expression)>
                  <LINESIZE(expression)>;
close-statement::=CLOSE file-option;
file-option::=FILE(reference)
```

The value of the argument to TITLE is the name of the dataset. The resolution of the name to the declaration is made in the same manner as resolution of other names, but is made when the open-statement is

executed. The significance of this is that different executions of the same statement can resolve to different datasets. The file object remains associated with the dataset until it is CLOSEd. A file object can only be associated with one dataset at a particular time. Any number of files may be associated with one dataset.

As well as determining a dataset, an opened file object names an object in that dataset. The objects are numbered 1 to N, and when initially opened the file names the first. Statements with a KEY or KEYFROM option set the name to the given value at the start of execution of the statement. The READ, WRITE, and REWRITE statements increment the number by one on their completion.

The READ statement copys a value from the object currently named by the file into the object referenced in the INTO option. A REWRITE statement copys a value from the object referenced in the FROM option into the object currently named by the file. The WRITE statement creates an object in the dataset with the name given by the file value, and then acts like REWRITE. These operations of copying values are not exactly the same as assignment - the data types of source and target must be identical.

```
read-statement::= READ file-option key-option
                   INTO(reference);
 key-option::=KEY(expression)
 rewrite-statement::=REWRITE file-option <key-option> from-option;
 from-option::=FROM(reference)
 write-statement::=WRITE file-option < KEYFROM(expression) >
                   from-option;
```

2.23 STARTING AND SYNCHRONIZING

For any procedure to execute there has to be a source of storage for the objects automatically allocated when the procedure starts, and there has to be an engine to execute the statements of the procedure. This combination of storage and engine is called a processor, and ultimately is implemented by hardware - the memory and Central Processing Unit.

Most programs are executed on the basis that there is only one processor, so that it does not need to be referenced explicitly. However for some problems it is valuable to be able to write as if there were several processors. This may be because the execution is on hardware with several processors, and this formulation of the program helps the implementation to exploit them. Or it may be the program is modelling, controlling, or reacting to several real-world processes that are not (entirely) synchronised.

The unit of allocation of work to a processor is the execution of a procedure (and the procedures invoked normally from it). A process begins when the procedure is STARTed, and terminates when the procedure terminates.

```
start-statement::= START call-body < PRIORITY(expression) >;
```

A number of factors contribute to how fast a process executes. The actual source statements in the procedure are relevant. The quality of the compiler that prepares them for execution is relevant. The power of the hardware processor(s) is relevant. If the program interacts with real-world activities, e.g. closing a valve, then the speeds of these activities are relevant. The language system cannot guarantee the speed of a process, but it will provide information to allow the speed to be worked out for a particular implementation, e.g. timings of generated code sequences will be listed.

Some control over process speed is given by the PRIORITY option. The argument is an integer between 1 and 1000 which represents a ranking of the desired speed of the new process. (i.e. Where the implementation has a choice the process with the larger ranking will execute faster.)

Often processes will need to cooperate. This may arise because one computes information used by the other, or may arise because they control or monitor pieces of real-world equipment that are not independent. Since the progress of real-world activity is discernable in the computer only by its reflection in the value (state) of some internal object, these two cases are essentially the same. Hence the general situation is one process assigning to an object and another using the value. Access to the object is shared (although only one of them allocates and frees the object).

Access to a shared object is not allowed to be simultaneous access by the two processes. (Assigning to the object may be a multi-step operation and it would not be meaningful to try to access the value

in the middle of these steps.) A process which is going to access a shared object must, during execution, acquire and release the right to do so. While it has the right no other process has the right. The construction for controlling this is the SELECT group.

When a SELECT group references shared objects in a WHEN expression the WHEN expression is deemed to evaluate to FALSE if a shared object cannot be accessed. If this causes all the WHEN expressions to be FALSE (and there is no OTHERWISE clause) then evaluation of the WHEN expressions is repeated. (The language system will detect if the program reaches a state where there is no prospect of the shared objects becoming accessible.)

When a WHEN expression evaluates to TRUE, the shared objects it references are available to this process (only) until the select group is left. To acquire this exclusive use, extra references may be added to the expression - the READY builtin function returns TRUE only when its argument is accessible.

```
WHEN(A>B & READY(C)):DO;/* A,B,C rights now held by this process */
                     C.PART1=A;
                     C.PART2=99;
                    END;/* Rights released */
```

2.24 INVALID PROGRAMS

The language specifies certain things that should not be written, e.g. unmatched parentheses. When these rules are broken the program is invalid. This is somewhat different from being in error, since a program can be in error solely in the sense that it fails to implement the programmers intentions. Mistakes that make a program invalid fall into different categories, in practice, because of the trade-offs that users want to make between the cost of checking them and the cost of not checking them.

Many mistakes are always checked for. This includes all those that can be checked at compile time.(q.v.)

There is a category of mistakes where the checking is under the users control. This control is exercised by assertions, in particular the NO-condition prefixes which assert that certain conditions will not occur. If the user elects to have assertions checked then a wrong assertion will cause ERROR to be signalled in execution. If the user elects not to have assertions checked and an assertion is wrong then the program is invalid. This last case is not checked for by the language system since to do so would negate the users control of checking.

The effect of such incorrect unchecked assertions is in general undefined. It may be that the effect is the same as that produced by some valid program, e.g. if SIZE occurs the value assigned to the target will be a possible value for the target, although not the value the programmer tried to give it. On the other hand the effect may be one that no valid program could produce. It will depend on the implementation whether a particular error is a 'safe' or 'dangerous' one.

Finally there are a class of mistakes which are never checked because a mechanism to check them would be prohibitively expensive to implement or use. These too may be 'safe' or 'dangerous' errors.

There are two noteworthy special cases. (1) The only dangerous error which can be written outside of a SYSTEM module is a subscript with the wrong value when this is not being checked. (2) The 'safe error' of unchecked UNDERFLOW has a natural interpretation as giving the result zero. This is sufficiently valuable that it is part of the language and not regarded as an error.

2.25 SYSTEM MODULES

Your installation may have chosen to make available some subroutine operations that only work on your system. These 'builtin instructions' are written like any other subroutine operations. Your installation will maintain a list of these builtin instructions, what type of arguments they need, and what they do.

Programs that use builtin instructions are not immediately suitable for use on other installations. To avoid inadvertent use, builtin instructions are only permitted in modules that are written with SYSTEM on the module statement.

Other special facilities are available in system modules, to allow programs to be dependent on a specific representation of data values. Every installation will represesent a Boolean value as one bit in storage, and represent BIT(n) as n bits. The representation of other data will be defined individually by installation. A program can know these representations by referencing a builtin constant structure (see the appendix). The representation is defined in terms of bit storage, e.g. FIXED DECIMAL(3) and FIXED DECIMAL(4) might be held as 2's-complement numbers in 16 bits.

In order to refer to the bit string underlying a reference one writes UNSPEC(reference). If a program is to run on one installation and create code for another installation it will be necessary to pass the data as entirely BIT data, for which UNSPEC is necessary.

Within an installation the representation of a pointer is the same as the representation of some integer, e.g. POINTER and FIXED DECIMAL(3) may both be 16 bits. In SYSTEM modules pointers and numbers having the same representation are regarded as having the same type. Hence if the installation attaches a special significance to some address, that numeric address can be used as a pointer. e.g. 80->T may be a reference to the hardware timer.

In a system module pointers may be declared that are not constrained to any type or area. The type and area associated with the object the pointer names are determined dynamically by the allocation that SETs the pointer.

The installation will define what is an error in a system module. e.g. 80->T may be OK but 81->T wrong, because of alignment. System modules should be written with extraordinary care - there are many opportunities for 'unsafe' mistakes.

2.26 SHORTHANDS

Several identifiers may be declared to have the same attributes by putting them in a list, e.g. DECLARE (J,K,L) INTEGER;

The form CONSTANT(expression) is a shorthand for CONSTANT INITIAL(expression)

A string constant containing several string symbols is an abbreviation for concatenation. e.g 'ABC' for 'A'||'B'||'C'

An arrow without an identifier following can be used if the pointer to the left of the object has a TO attribute. The argument of the TO is inserted on the right of the arrow.

```
 DECLARE P TO(CELL);X=P->;/* Refers to P->CELL */
```

A subscript consisting of two expressions separated by a colon is a variation of SUBSTR. Reference(J:K) means SUBSTR(Reference,J,K-J+1)

DO WHILE; means DO WHILE(TRUE);

2.27 EDITING, TRANSLATING, AND EXECUTING PROGRAMS.

<< It may not be a good idea to make this part of the language. One can argue that it is not necessary for the embedded programs themselves, only for their production. And there are certainly some difficulties in avoiding implicitly specifying an operating system.>>

Programs are entered and modified under control of an editor. The programs produced are IRONMAN datasets but the user is not concerned with their details because only the editor directly maintains the datasets. The user refers to procedures and modules. Reference at a detail level depends on the style of the editor, e.g. it might be by line number or by cursor position on a screen. Because of the possibility of indirect addressing, any procedure or module can be referenced.

The translator partially prepares programs for execution. The arguments to it determine the procedure or module compiled, the context it is compiled in, and the objectives of the compilation.

Execution occurs when a procedure is STARTed.

Details of parameters to these builtin programs are given in the appendices.

2.28 MESSAGES

The translator makes a variety of consistency checks on a piece of program. The written form must be recognizable so, for example, there will be an error message if opening and closing brackets are not matched. Typical messages of this sort are: +++

Other errors are errors of type, for example if a reference has an argument list the identifier must be either an array being subscripted or an operation. Typical messages of this sort are: +++

Some language constructs are intended as documentation that also causes checking. The CONSTANT attribute documents that the object is not assigned to, and the RESULT attribute that it is only assigned to.

2.29 MACHINE DESCRIPTION

<< A special constant structure can be referenced from any program to obtain details of the hardware being compiled for, such as the details of floating point implementation. >>

2.30 PROGRAM INTERCHANGE

<< It is necessary to say more than just ASCII to promote interchange. Here would be the specifications for the datasets and labelling necessary. >>

2.31 PARAMETERS TO BUILTIN PROGRAMS

<< A specification of the inputs to the translators etc. that make up the language system. For example, of the parameter that determines whether the compiler produces a source listing. >>

2.32 BUILTIN FUNCTIONS.

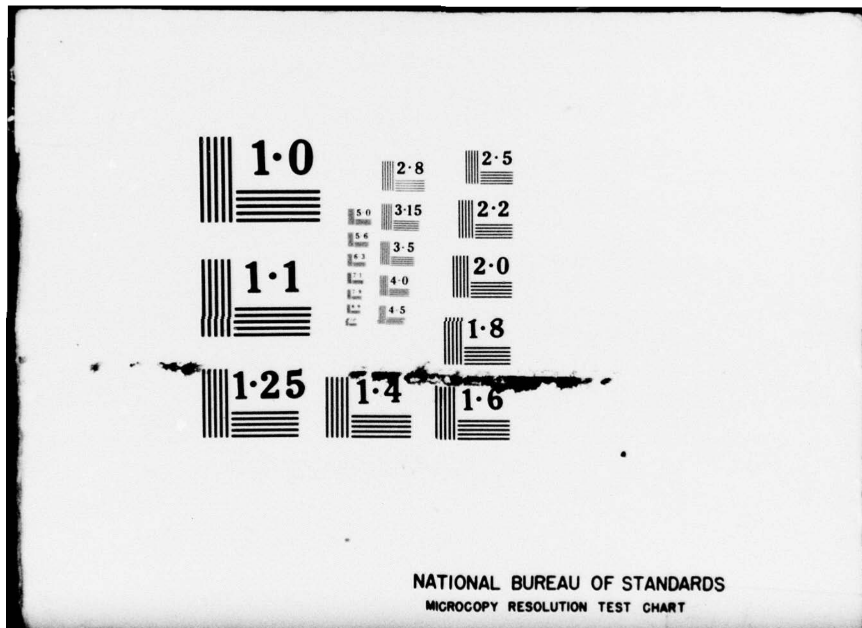
<< names, arguments, conditions raised etc.

AD-A052 082   IBM UNITED KINGDOM LABS LTD WINCHESTER (ENGLAND)   F/G 9/2
TOWARDS A PL/I-BASED 'IRONMAN' LANGUAGE.(U)
DEC 77   R F MADDOCK, B L MARKS

UNCLASSIFIED   TR-12-168   NL

2 OF 2
ADA
052082

END
DATE
FILMED
5 -78
DDC


1·0
1·1
1·25
1·4
1·6
1·8
2·0
2·2
2·5
2·8
3·15
3·5
4·0
4·5
5·0
5·6
NATIONAL BUREAU OF STANDARDS
MICROCOPY RESOLUTION TEST CHART

2.33 SYNTAX

<< The Users manual will need a 'Railway-shunting-yard' style of syntax description. >>

CHAPTER 3. EXAMPLES

3.1 SIMPLE PL/I

This example illustrates simple data processing. It is a subroutine that reads N records from a dataset, normalises them and prints them.

```
SHOW_NORMALIZED:PROCEDURE(N);
 DECLARE N FIXED DECIMAL(4);/* Number of records, for this call */
 DECLARE VALUES(N) FLOAT DECIMAL(8);/* Copies of the records */
 DECLARE J FIXED DECIMAL(4);/* Counts up to N */
 DECLARE X FLOAT DECIMAL(8) INITIAL(0);/* Sum of the values */
   DO J=1 to N;
   READ FILE(IN) INTO(VALUES(J));
   X=X+VALUES(J);
   END;
/* If X is zero the caller will be signalled by the following code */
   DO J=1 TO N;
   PUT FILE(OUT) EDIT(VALUES(J)/X)(SKIP,E(8,6));
   END;
 END SHOW_NORMALIZED;
```

3.2 COMPLEX ARITHMETIC

We define a COMPLEX datatype and the usual arithmetic operations on COMPLEX and between COMPLEX and reals. The definition is encapsulated in a module which could be included in a program by INCLUDE COMPLEX;

```
COMPLEX:MODULE                        /* same name as type is ok */
EXPORTS( COMPLEX,
         "+","-","*","/");            /* this exports all the definitions
                                         of e.g. "+" in this module */
  DECLARE 1 COMPLEX BASED,
          2 REAL FLOAT DECIMAL(6),
          2 IMAG FLOAT DECIMAL(6);

/* Infix operators between COMPLEX */

"+":PROCEDURE(P,Q) RETURNS(COMPLEX);
    DECLARE (P,Q) COMPLEX;
    RETURN [COMPLEX (REAL: P.REAL+Q.REAL,
                     IMAG: P.IMAG+Q.IMAG) ];
    END "+";

"*":PROCEDURE(P,Q) RETURNS(COMPLEX);
    DECLARE (P,Q) COMPLEX;
    RETURN [COMPLEX (REAL: P.REAL*Q.REAL-P.IMAG*Q.IMAG,
                     IMAG: P.IMAG*Q.REAL+P.REAL*Q.IMAG) ];
    END "*";

"-":PROCEDURE(P,Q) RETURNS(COMPLEX);  similar to above
"/":PROCEDURE(P,Q) RETURNS(COMPLEX);  similar to above
/* Prefix operations on COMPLEX */

"+":PROCEDURE(P) RETURNS(COMPLEX);
    DECLARE P COMPLEX;
    RETURN(P);
    END "+";

"-":PROCEDURE(P) RETURNS(COMPLEX);  similar to above

/* Operations between COMPLEX and real */

/* The "+" operation is always commutative, and so this
   definition defines both COMPLEX + real and vice versa
*/

"+":PROCEDURE(P,X) RETURNS(COMPLEX);
    DECLARE P COMPLEX, X FLOAT DECIMAL(6);
    RETURN [COMPLEX (REAL:P.REAL+X,
                     IMAG:P.IMAG)  ];
    END "+";
```

```
"-":PROCEDURE(P,X) RETURNS(COMPLEX);  similar to above
"*":PROCEDURE(P,X) RETURNS(COMPLEX);  similar to above

/* The "/" operation is not commutative, and needs two definitions */

"/":PROCEDURE(P,X) RETURNS(COMPLEX);
    DECLARE P COMPLEX, X FLOAT DECIMAL(6);
    RETURN [ COMPLEX( REAL:P.REAL/X,
                      IMAG:P.IMAG/X) ];
    END "/";

"/":PROCEDURE(X,P) RETURNS(COMPLEX);
    DECLARE P COMPLEX, X FLOAT DECIMAL(6);
    RETURN [ COMPLEX(REAL:X,IMAG:0)/P ];
        /* notice the use of another procedure in this module */
    END "/";

END COMPLEX;
```

To allow precision to be declared for COMPLEX items, rather than be fixed in the definition, we could write:

```
DECLARE 1 COMPLEX BASED,
          2 REAL FLOAT DECIMAL(?1),
          2 IMAG FLOAT DECIMAL(?1);
```

The rest of the example could be the same, except that the real operands X would be declared:

```
DECLARE X FLOAT DECIMAL(?);
```

COMPLEX items would then be declared COMPLEX(n) rather than COMPLEX.

It would also be possible to define a REAL type as FLOAT DECIMAL(6) or FLOAT DECIMAL(?), and this could be used to define the components of COMPLEX, as well as to declare data.

3.3 PRODUCER/CONSUMER

This is the simple example in the Series/1 PL/I manual of one producer, running from 9 to 5, and multiple consumers communicating via a bucket holding one widget. The solution here could easily be extended to multiple producers or a larger bucket.

```
EXAMPLE:PROCESS;
INCLUDE CLOCK;  /* provides timing facilities, including HHMMTIME */
DECLARE BUCKET VALUES("FULL","EMPTY","LAST") SHARED;

PRODUCER:PROCESS;
   DECLARE INFILE RECORD,
           INRECORD ... ;
   ON ENDFILE(INFILE) GO TO FIN;
   OPEN FILE(INFILE);

   SELECT;
   WHEN(HHMMTIME > 0900);
   OTHERWISE WAIT;
   END;

 LOOP:DO WHILE(TRUE);
   SELECT;
   WHEN(HHMMTIME > 1700) RETURN;    /* End of producer process */
   OTHERWISE SELECT;
             WHEN(BUCKET="EMPTY") BUCKET="FULL";
             OTHERWISE WAIT;
             END;
   END; END LOOP;
FIN:SELECT;
   WHEN(BUCKET="EMPTY") BUCKET="LAST";
   OTHERWISE WAIT;
   END;

   CLOSE FILE(INFILE);
   RETURN;
   END PRODUCER;

CONSUMER:PROCESS;

 LOOP:DO WHILE(TRUE);
   SELECT;
   WHEN(BUCKET="FULL") BUCKET="EMPTY";
   WHEN(BUCKET="LAST") RETURN;         /* End of consumer process */
   OTHERWISE WAIT;
   END; END LOOP;
   END CONSUMER;
```

```
/* Body of main process EXAMPLE */

START PRODUCER;
START CONSUMER;          /* Several if desired */

END EXAMPLE;             /* Will wait for producer and consumer(s)
                            because they use the SHARED BUCKET     */
```

The example above includes a module called CLOCK to access the time of day. The following is an example of how this might be implemented using the System/360 interval timer. Presumably we would not want coarse timing like 0900 hrs and 1700 hrs to interfere with fine timing. This clock module implements a coarse timer called HHMMTIME, and could be extended to provide others.

Procedure SET_CLOCK(T) sets the time of day and starts a process to maintain it, and STOP_CLOCK stops this process.

```
CLOCK:SYSTEM MODULE     /* Uses the S/360 interval timer */
EXPORTS (HHMMTIME, SET_CLOCK, STOP_CLOCK);
DECLARE TIME SHARED FIXED DECIMAL(9) /*hhmmssddd*/,
        HHMMTIME SHARED FIXED DECIMAL(4),
        STOP SHARED BIT INITIAL(FALSE);

SET_CLOCK:PROCEDURE(T);
   DECLARE T CHARACTER(4);
   TIME=FIXED(T)*100000;
   START TIMER_WATCH;
   END SET_CLOCK;
STOP_CLOCK:PROCEDURE;
   STOP=TRUE;
   END STOP_CLOCK;

TIMER_WATCH:PROCESS;
   DECLARE TENTH_SECOND FIXED DECIMAL(9) CONSTANT(30),
           TIMER_INTERRUPT SHARED BIT ENVIRONMENT(PSW(88));
           /* S/360 timer interrupt loads PSW from location 88 */

 LOOP:DO WHILE(TRUE);
   SELECT;
   WHEN(STOP) RETURN;
   WHEN(TIMER_INTERRUPT) DO;
                         80->=TENTH_SECOND;  /* location 80 is
                                                the timer */
                         UPDATE_TIMERS;
                         END;
   OTHERWISE WAIT;
   END; END LOOP;

UPDATE_TIMERS:PROCEDURE;
   add 1/10 sec to TIME
   IF (TIME is exact minute) THEN add 1 min to HHMMTIME
   update any other timers
   END UPDATE_TIMERS;

   END TIMER_WATCH;

END CLOCK;
```

3.4 LTPL-E/297

This example was first presented by Peter Elzer in LTPL-E paper number 297.

"Assume three tasks T1, T2 and T3. Let T1 be a data acquisition task taking data from an ADC and filling them into two buffers B1 and B2. It can only write into one buffer when this buffer is empty. It declares a buffer 'full' after it has written into it.

Let T2 be a sorting task which takes the data from B1 or B2 alternatively, does some calculation on these data and puts the results into another data area called S (for spectrum). T2 can only read from the buffers B1 and B2 after they have been filled by T1. It declares them empty after having evaluated their respective contents.

Let T3 be e.g. a display task, which draws a picture of the contents of S on a display screen. It shall run forever, unless stopped by external influence, because it shall show the dynamic behaviour of data accumulated in S.

Up to now this is a classical double-buffer-problem which can be solved by nearly all known synchronization mechanisms. But now let's assume a boundary condition imposed on this problem by the demands of practical use. For some reason the device ADC shall stop or be stopped. Now the task T2 shall be able to evaluate the remaining data in either B1 or B2 whatever buffer is just in use when 'ADC-stop' occurs.

This problem has turned out to be somewhat difficult to solve with e.g. sempahores."

```
MAIN: PROCEDURE;

DECLARE N FIXED DECIMAL(3) CONSTANT(100),
        S SHARED ...;

DECLARE 1 BUFFER BASED SHARED,
          2 STATE VALUES ("EMPTY","FILLED") INITIAL("EMPTY"),
          2 COUNT FIXED DECIMAL(3),
          2 DATA(N) ...;

DECLARE B(2) BUFFER;
```

```
T1:  PROCESS;
     DECLARE X ..., I FIXED DECIMAL(1);
LOOP: DO WHILE(TRUE);
      DO I=1 UP TO 2;
      SELECT;
      WHEN(B(I).STATE="EMPTY") DO;
          DO B(I).COUNT=1 UP TO N;
          R: READ ADC INTO(X);
          IF X=STOPPED THEN IF B(I).COUNT > 1 THEN LEAVE;
                                             ELSE GO TO R;
                        ELSE DATA(B(I).COUNT)=X;
          END;
          B(I).STATE="FILLED";
          END;
      OTHERWISE WAIT;
      END;
      END; END LOOP;
END T1;

T2:  PROCESS;
     DECLARE I FIXED DECIMAL(1);
LOOP: DO WHILE(TRUE);
      DO I=1 UP TO 2;
      SELECT;
      WHEN (B(I).STATE="FILLED") DO;
                     process B(I).DATA(1 TO COUNT);
                     SELECT;
                     WHEN(AVAILABLE(S)) put result in S;
                     OTHERWISE WAIT;
                     END;
                     END;
      OTHERWISE WAIT;
      END;
      END;
      END LOOP;
END T2;

T3: PROCESS;

LOOP: DO WHILE(TRUE);
      SELECT;
      WHEN(AVAILABLE(S)) draw S on screen;
      OTHERWISE WAIT;
      END;
      END LOOP;
END T3;
/* Body of MAIN procedure */

START T1;
START T2;
START T3;
```

/* No way has been included to stop these three processes. The "stopping" of ADC will suspend T2 and T3, but they will resume when

more data is received. */

END MAIN;

Appendix: Intersection with PL/I

Since the proposed language is based on PL/I, there is an imtersection between the two languages: the parts of PL/I which are valid in the new language. The intersection language could perhaps be used to implement the first compiler for the new language, using existing PL/I compilers to bootstrap. The intersection language is described briefly in this appendix.

1. DATA TYPES

All variables must be declared with one of the following data types specified (except for structures).

```
BIT [(extent-expr)]
CHARACTER [(extent-expr)]
FIXED DECIMAL (p[,q])     0≤q≤p
FLOAT DECIMAL (p)
POINTER
AREA (extent-expr)
```

Note: Because the POINTERs are unconstrained and not in a SYSTEM MODULE the new language compiler will issue a warning. The extent-expression form of AREA size would also only be allowed in SYSTEM MODULEs.

Literal forms exist for constants of the four computational types:

```
BIT            '101' or BIT('101')
CHARACTER      'ABC'
FIXED DECIMAL  23   0.625
FLOAT DECIMAL  9.62E5 or any FIXED DECIMAL constant in a FLOAT
               context.
```

File constants (but not variables) can also be declared.

```
FILE [RECORD[KEYED]][PRINT]
```

Procedures are discussed further under "Program Structure", and Files under "Input-Output".

Aggregates

Structures can be declared using level numbers, e.g.

```
DCL 1 STRUCT,
      2 A FIXED DECIMAL(5),
      2 B POINTER;
```

Arrays of one or more dimensions can be declared; the lower bound is always 1 and may not be specified, the upper bounds are specified thus:

```
(extent-expr-commalist) datatype, e.g.
DCL ARRAY (10,I+2) BIT;
```

Arrays of structures, structures of arrays and structures of structures are all allowed. A structure component or array may have any variable type, except AREA.

Storage Classes

BASED, parameter and automatic can be used. BASED is specified explicitly in the declaration. Parameter is implied by the variable appearing in a parameter list. If none of these is specified, the variable is automatic. BASED and automatic may be initialised.

The syntax of initial is:

```
initial-attribute ::= INITIAL (item-commalist)
item ::=  [prefix-op] constant|NULL} |
          (constant-expression |
          (constant-integer-expression) (item-commalist)
```

Note: ALIGNED/UNALIGNED can be specified on elements or structures,although the new language compiler will issue a warning message outside SYSTEM MODULEs. Unaligned BIT items cannot be used as arguments.

Extent-expressions for array bounds, string lengths, and area size (if included) may have the following forms:

```
1) constant-expression or *          for parameter
2) expression                        for automatic
3) expression involving only
   constants and components of
   the BASED structure               for BASED
```

Declare statement

```
DECLARE   {[level-no]
          {identifier|(identifier-commalist)}
          attributes},..;
```

Allocation of BASED storage

```
[label:] ALLOCATE based-variable [IN(area-ref)
                                   SET(pointer-ref)];
[label:] FREE based-reference;
```

Note: The SET and IN options can be omitted if the new language allows this in SYSTEM MODULEs.

2. COMPUTATION

The assignment statement has the form

reference=expression;

References can be pointer qualified, dot qualified and subscripted. They must be connected, so * subscripts and references to components of arrays of structures are not allowed. The expression must have the same data type aggregation and extents as the target reference, with the following exceptions:

FIXED DECIMAL of different precision.
FIXED DECIMAL with target scale larger than source.
FLOAT DECIMAL of different precision
FLOAT DECIMAL target and FIXED DECIMAL source

Expressions

No conversion is done to operands in expressions, so they must all be the same datatype. Some operations are only allowed where there is an explicit target - i.e. assignments of the form "A = B op C".

The <u>operators</u> are:

```
                                         with explicit target

+ -       FLOAT, FIXED
*         FLOAT, FIXED
/         FLOAT                          FIXED

& | ¬     BIT (same constant length)     same variable length
||                                       CHAR

= ¬= >    All scalars of same type,
< >= <=   CHARACTER of different lengths.
```

3. PROGRAM STRUCTURE

A program consists of several procedures. Each begins with a PROCEDURE statement and ends with an END statement. In between may appear DECLARE statements and internal procedures, then ON statements and then executable statements.

```
[(condition-commalist):][label:]

      PROCEDURE[(parameter-identifier-commalist)]
            [RETURNS(descriptor)];
```

Functions and subroutines can both be used, but each PROCEDURE must be one or the other. A function may not assign to its parameters. Constants or expressions may not be used as arguments if the called subroutine assigns to the corresponding parameter.

<u>Conditional control structures</u>

IF bit(1)-expression THEN clause [ELSE clause]

Each clause may be an executable statement, including IF, or a DO-group. They may not be labelled.

```
DO WHILE (bit(1)-expression); or
DO ctlvar = expr [BY expr2] TO expr3;
executable statements
END[label];
```

In the iterative form of DO, the expressions must be FIXED, and the BY-expression must be positive. The control variable must be an unqualified unsubscripted local automatic FIXED variable. It must not be referenced except in a DO-loop to which it is the control variable, and must not be set.

GO TO label;

This statement transfers control to the specified label, which must be in the same block as the GO TO statement.

A null statement, consisting of just a semicolon, can be used to carry a label or to express a null clause, after THEN for example.

```
SELECT [(integer-expression)];
WHEN (expression-commalist)clause;
  .
  .
[OTHERWISE clause;]
END [label];
```

The values of the SELECT and WHEN expressions determine which alternative is to be executed.

Each END statement terminates one PROCEDURE or DO or SELECT group. The label, if it appears, must match the label on the corresponding PROCEDURE etc statement.

LEAVE [label];

This statement transfers control to the statement after the END of the enclosing DO or SELECT group with the specified label, or the immediately enclosing group if none is specified.

<u>Note</u>:SELECT and LEAVE are not a subset of ANS PL/I, but of a proposal which is under consideration as an extension to the standard.

4. EXCEPTION HANDLING

In each procedure, action may be specified for each of various exception conditions using ON-statements

ON condition GO TO label;

Conditions are raised implicitly by various language constructs, e.g ENDFILE by READ, and can also be raised explicitly by the SIGNAL statement:

SIGNAL condition;

ON-statements must appear before any other executable statements in the block. If an exception occurs and there is no ON-statement in the procedure, the procedure activation is terminated and the same condition raised in the calling procedure. If there is no ON-statement in any procedure, system action is taken.

The conditions are:

```
OVERFLOW,SIZE,SUBSCRIPTRANGE
CONVERSION,UNDERFLOW,ZERODIVIDE
ENDFILE,KEY,RECORD,TRANSMIT,UNDEFINEDFILE
AREA,STORAGE,FINISH,ERROR
```

Notes: The restrictions on the placing of ON-statements mean that the translator knows that they apply throughout the block. This, and restricting condition prefixes to blocks means that exception handling information can be static tables for each procedure. No execution time is required to set these up on entry to the procedure. When a condition is raised, the appropriate ON-unit can be found from these tables and the normal call-chain.

No return from ON-units leads to better structured programs, and helps the efficient handling of conditions in called procedures. It makes the ENDPAGE condition rather useless, and so this is excluded, as is the PAGENO bif.

REVERT is not needed with this subset of ON-handling.

5. INPUT-OUTPUT

There are three kinds of files: stream files where character data (i.e. human readable) is transferred sequentially character by character, RECORD files where data in internal format is transferred sequentially a record at a time, and KEYED RECORD where data in internal format is transferred a record at a time but not sequentially - the record number is specified by a FIXED integer called the KEY.

Note

This subset of RECORD I/O is what in IBM PL/I is called REGIONAL(1). It provides the basic facilities provided by sequential and direct access devices. Content addressing by keys is device-dependent and complex indexes are implemented in software and could be written in this language as library routines.

Files must be explicitly opened before use and closed after use

```
OPEN FILE(filename) [LINESIZE(constant-expression)]
     {INPUT|OUTPUT|UPDATE};
CLOSE FILE(filename);
```

The allowed input-output statements for each file type are given in the table below:

	Stream	RECORD	KEYED RECORD
INPUT	GET[SKIP] EDIT	READ INTO	READ[KEY] INTO
OUTPUT	PUT[SKIP] EDIT	WRITE FROM	WRITE[KEYFROM] FROM
	PUT[SKIP] LIST		
UPDATE		READ INTO	READ[KEY] INTO
		REWRITE FROM	REWRITE[KEY] FROM

In each statement a FILE option must appear after the statement keyword; the options must appear in the order listed.

The options are:

```
FILE (filename)
INTO (reference)      )these references will normally
FROM (reference)      )be structures
KEY  (integer-expression)
KEYFROM (integer-expression)
SKIP
LIST (expression-commalist)
EDIT (expression-commalist)(format-commalist)
```

LIST output uses a standard format for each data type and successive items are printed with an intervening blank. (i.e. there is a tab at every position.

In EDIT output each item is printed in the format specified by the corresponding format item. EDIT input is the reverse.

```
format::= (expression) (format-commalist) |
          COL (expression) |
          SKIP [(expression)] |
          PAGE |
          X (expression) |
          A [(expression)] |        for CHARACTER items only
          B [(expression)] |        for BIT items only
          E (expr1 [,expr2]) |      for FLOAT items only
          F (expr1 [,expr2]) |      for FIXED items only
          P 'picture'               for FIXED items only
```

Pictures are provided only for FIXED items, the allowed picture characters are 9VZ*$S-.,B. All expressions in a format list must be constant integer expressions.

6. BUILTIN FUNCTIONS

The functions EMPTY, NULL, ONCODE and DIM are in the new language. Most other PL/I built-in functions could be defined in the new language, and thus can be used in the intersection. The few exceptions include ADDR.

References

1. Department of Defense Requirements for High Order Computer Programming Languages, "Ironman", 14 January 1977.

2. American National Standard Programming Language PL/I, X3.53-1976.

3. "Abstraction Mechanisms in CLU", B.Liskov et al., Communications of the ACM, August 1977.

4. "ILIAD: A High Level Language for Industrial Control Applications", M.S.Adix, General Motors Research Report GMR-2015A, August 1976.

5. "Some Ideas on Data Types in High Level Languages", D.Gries and N.Gehani, Communications of the ACM, June 1977.

6. "A Discipline of Programming", E.W.Dijkstra, Prentice-Hall, 1976.

7. "Design and Implementation of MODULA", N.Wirth, Software-Practice and Experience, Jan-Feb 1977.

8. "The Programming Language Concurrent PASCAL", P.Brinch Hansen, IEEE Transactions on Software Engineering, June 1975.

9. "Guide to PL/S II", IBM Manual GC28-6794, May 1974.