A STUDY OF SYSTEMS IMPLEMENTATION LANGUAGES FOR THE POCCNET SYSTEM

Victor R. Basili
James W. Franklin
Department of Computer Science
University of Maryland

August 27, 1976

Prepared for Goddard Space Flight Center
under Contract NAS 5-22581

ABSTRACT:

This report presents the results of a study of systems implementation languages for the Payload Operations Control Center Network (POCCNET). Criteria are developed for evaluating the languages, and fifteen existing languages are evaluated on the basis of these criteria.

Table of Contents

REFERENCES

[BAR74] Barth, C. Wrandle, STRCMACS: An Extensive Set of Macros for Structured Programming in OS/360 Assembly Language, Goddard Space Flight Center, Greenbelt, Maryland, 1974.

[BAS74] Basili, V. R., and Turner, A. J., SIMPL-T: A Structured Programming Language, Computer Science Center, Univ. of Maryland, Computer Note CN-14, 1974.

[BAS76a] Basili, Victor R., "The SIMPL Family of Programming Languages and Compilers", Graphensprachen und Algorithmen auf Graphen, Carl Hansen Verlag, Munich, Germany, 1976, 49-85. Also Computer Science Technical Report #305, Univ. of Maryland, June 1974.

[BAS76b] Basili, Victor R., Language as a Tool for Scientific Programming, Department of Computer Science, Univ. of Maryland, 1976.

[BEY75a] Beyer, Terry, FLECS: User's Manual, Computer Science Department, Univ. of Oregon, 1975.

[BEY75b] Beyer, Terry, FLECS General Information Letter, Computer Science Department, Univ. of Oregon, 1975.

[CHE68] Cheatham, T. E., et al., "On the basis for ELF - an extensible language facility", Proc. AFIPS 1968 FJCC, Vol. 33#2, 937-948.

[DEC74] BLISS-11 Programmer's Manual, Digital Equipment Corporation, Maynard, Mass., 1974.

[DES76a] desJardins, R., and Hahn, J., A Concept for a Payload Operations Control Center Network (POCCNET), Goddard Space Flight Center, Greenbelt, Maryland, 1976.

[DES76b] desJardins, Richard, Systems Definition Phase Project Plan for Payload Operations Control Center Network, Goddard Space Flight Center, Greenbelt, Maryland, 1976.

[FRE75] French, A., and Mott-Smith, J., Draft of AFSC HOL Standardization Program - Phase 1 Report, ESD/MCIT, Hanson Air Force Base, Bedford, Mass., 1975.

[HAM76] Hamlet, Richard, SIMPL-XI - An Introduction to High Level Systems Programming, Department of Computer Science, Univ. of Maryland, Lecture Note LN-4, 1976.

[HAN75a] Hansen, Per Brinch, CONCURRENT PASCAL Introduction, Information Science, California Institute of Tech., 1975.

[HAN75b] Hansen, Per Brinch, CONCURRENT PASCAL Report, Information Science, California Institute of Tech., 1975.

[HAN75c] Hansen, Per Brinch, The SOLO Operating System, Information Science, California Institute of Tech., 1975.

[IEE75] Proc. 1st National Conference on Software Engineering, IEEE Computer Society, Washington, D.C., 1975

[INT75a] CS-4 Language Reference Manual and CS-4 Operating System Interface, Intermetrics Inc., Cambridge, Mass., 1975.

[INT75b] HAL/S Language Specification, Intermetrics Inc., Cambridge, Mass., 1975.

[INTE74a] FORTRAN V Level 1 Reference Manual, Interdata Inc., Oceanport, N.J., 1974.

[INTE74b] FORTRAN V Level 1 User's Guide, Interdata Inc., Oceanport, N.J., 1974.

[INTE74c] FORTRAN V Level 1 Run Time Library Manual, Interdata Inc., Oceanport, N.J., 1974.

[JEN74] Jensen, K., and Wirth, N., PASCAL User Manual and Report, Lecture Notes in Computer Science Series, Springer-Verlag, New York, 1974.

[JOH73] Johnson, Mark S., et al., A Basic Guide to JOSSLE, Department of Computer Science, Univ. of California at Santa Barbara, 1973.

[KAF75] Kaffen, N., and Rodeheffer, T., PREST4 - A Highly Structured Fortran Language for Systems Programming, Computer Science Department, Ohio State Univ., TR-75-4, 1975.

[KER74] Kernighan, Brian W., "Programming in C - A Tutorial", Documents for Use with the UNIX Time-sharing System, Bell Laboratories, Murray Hill, N.J., 1974.

[LIS74] Liskov, B., and Zilles, S., "Programming with Abstract Data Types", Proc. Symposium on Very High Level Languages, SIGPLAN Notices, Vol. 9#4, April 1974.

[MAR74] Martin, Fred H., HAL/S - The Programming Language for Shuttle, Intermetrics Inc., Cambridge, Mass., 1974.

[MAR75] Martin, Fred H., JSC HAL Support Note # 15-75, Intermetrics Inc., Cambridge, Mass., 1975.

[MEI75] Meissner, Loren P., "On Extending Fortran Control Structures to Facilitate Structured Programming", SIGPLAN Notices, Vol. 10#9, Sept. 1975, 19-30.

[PRE73] Presser, L., and White, J., "A Tool for Enforcing System Structure", Proc. ACM 1973, Atlanta, 114-118.

[REI75] Reinschmidt, Marlene, JOVIAL/J3B Programmer's Guide, SofTech Inc., Waltham, Mass., 1975.

[RIC76] Richmond, George H., "PASCAL Newsletter", SIGPLAN Notices, Vol. 11#2, February 1976, 38-42.

[RIT74] Ritchie, Dennis M., "C Reference Manual", Documents for Use with the UNIX Time-sharing System, Bell Laboratories, Murray Hill, N.J., 1974.

[RUS76] Russell, D., and Sue, J., "Implementation of a PASCAL Compiler for the IBM 360", Software Practice and Experience, Vol. 6, 1976, 371-376.

[SDC70] SPL / Mark IV Reference Manual, System Development Corp., Santa Monica, Calif., 1970.

[SHI74] Shields, David, Guide to the LITTLE Language, New York Univ., 1974.

[SIG75] Proc. International Conference on Reliable Software, SIGPLAN Notices, Vol. 10#6, June 1975.

[SOF75] JOVIAL/J3B Language Specification - Extension 2, SofTech Inc., Waltham, Mass., 1975.

1. INTRODUCTION

This report presents an evaluation of systems implementation languages for the Payload Operations Control Center Network (POCCNET), which is a general hardware/software concept adopted by GSFC as a means of developing and operating payload operations control centers in the 1980's. The POCCNET system [DES76a,DES76b] will provide hardware and software resource-sharing via a distributed computer network and a package of standardized applications software. This report develops criteria for evaluating POCCNET implementation languages, and then compares fifteen existing languages on the basis of these criteria.

An attempt was made during this study to examine a wide range of existing languages, from a low level macro assembler to the very large and high level language CS-4. The following fifteen languages were examined in detail:

- BLISS-11 - A systems implementation language for the PDP-11 series.
- C - The language of the UNIX operating system.
- CONCURRENT PASCAL - A high level language for writing operating systems.
- CS-4 Base Language - An extensible language being developed for the Navy.
- FLECS - A Fortran preprocessor.
- HAL/S - The NASA language for the Space Shuttle program.
- INTERDATA FORTRAN V - An extension of ANSI Fortran.
- JOSSLE - A PL/I derivative for writing compilers.
- JOVIAL/J3B - A close relative of JOVIAL/J3, the Air Force standard language for command and control applications.
- LITTLE - A Fortran derivative that operates on bit strings of arbitrary length.

PASCAL	- A highly structured, general purpose language.
PREST4	- A Fortran preprocessor.
SIMPL-T	- The base member of a highly structured family of languages.
SPL / MARK IV	- A high level language with many machine-oriented features.
STRCMACS	- A collection of structured programming macros for IBM OS/360 assembly language.

The language evaluations in this report are based solely on the language reference manuals and other papers listed in the references. We have immediate access to the compilers for only two of the fifteen languages (C and SIMPL-T).

The criteria for evaluating the languages and the preliminary evaluations are presented in the second chapter of this report. Each evaluation is composed of two sections. The first section provides a detailed summary of the following <u>syntactic features</u> of the language:

(1) basic data types and operators
(2) control structures
(3) data structures
(4) other interesting features
(5) language syntax
(6) runtime environment .

The second section of each evaluation presents the <u>characteristics</u> of the language:

(1) machine dependence
(2) efficiency
(3) level of the language
(4) size of the language and compiler
(5) special system features
(6) error checking and debugging
(7) design support (modularity, modifiability, and reliability)
(8) use and availability of the language .

In the third chapter we give a summary of the functional subsystems in POCCNET, and then identify the programming application areas within the network. POCCNET will require a language or group of languages supporting general system programming, real-time processing, data base management, numerical processing, and data formatting and conversion. As can be seen, the application areas in POCCNET are diverse.

The fourth chapter contains a series of tables providing a cross reference between the language features and languages discussed in Chapter 2. Each table is devoted to one of the specific POCCNET requirements: each contains the language features contributing to the POCCNET requirement, and indicates for each language feature the presence or absence of that feature in the fifteen languages.

In the fifth and final chapter we give our recommendations and a discussion of possible candidates for the POCCNET implementation language.

2. CRITERIA AND EVALUATION OF THE LANGUAGES

In this chapter we give a detailed evaluation of the fifteen languages covered by this study. Each of the languages is evaluated on the syntactic features of the language (such as basic data types, control structures, and data structures) and on the characteristics of the language (such as machine dependence, efficiency, and design support). The evaluations are based solely on the language reference manuals and other papers listed in the references.

The section on language features contains the following subsections:

(1) A short introduction indicating the source of the language and the intended application area;

(2) The primitive data types of the language and the operators and functions for manipulating them;

(3) The control structures in the language. These are described using a simple, BNF-like metalanguage. Syntactic entities in the language are enclosed in the symbols "<" and ">", language keywords are always capitalized, and any optional features are enclosed in braces "{", "}". Where a choice is available between several features they are listed one above the other, single spaced. For example:

```
              IF <boolean-expr> THEN <stmt> { ELSE <stmt> }

              WHILE <boolean-expr> REPEAT <stmt-list> END ;
              UNTIL

              DO <stmt-#> <var> = <e-1>, <e-2> { ,<e-3> }
               <stmt-list>
     <stmt-#> CONTINUE
```

(4) The data structures in the language, and the operators for manipulating them. All but one of the languages in this study have arrays, others provide record structures, tables, sets, typed pointers, and file types;

(5) Any interesting features in the language not covered in the first four subsections. This typically includes macro processors, I/O facilities, CONSTANT declarations, and "include" statements for copying source files into a program;

(6) The approximate number of productions in the BNF grammar used to describe the language. Since the grammars used in the reference manuals vary from syntax charts to the grammars used by the production compilers, this number only provides a rough measure of the size and complexity of the language.

Any rules containing the BNF OR-operator "!" are considered to be multiple productions. Thus, the rule

```
<loop-stmt> ::= (WHILE ! UNTIL) <boolean-expr>
                          REPEAT <stmt-list> END
```

is considered to be two productions;

(7) The runtime environment required to support the language. For example, a language that permits recursive procedures will require a runtime stack, and languages with full character string processing will require a runtime stack or dynamic storage area to store temporary results during the evaluation of string expressions. Other languages require routines for process management, real-time scheduling, I/O, interrupt handling, and error monitors.

The section on language characteristics contains the following subsections:

(1) Machine dependence. Some of the languages in this report are truly transportable, while others contain machine or implementation dependent features such as inline assembly language, EQUIVALENCE statements for overlaying data items, user specified allocation of data items in records (word position and bit position within a word), and access to hardware registers;

(2) Efficiency of the language. Languages with high level operators and a structured control structure permit a great deal of optimization to be performed. Overlays, user specified allocation of records, and packing attributes on tables can be used to conserve storage space. Some of the languages have compiler directives for requesting that certain program variables be allocated in high speed storage, or to force procedures to be expanded inline at the point of invocation (rather than generating a calling sequence);

(3) Level of the language. The languages in this report range from very low level (STRCMACS) to high level (CS-4, HAL/S, PASCAL). The low level languages are typeless and generally have many machine-oriented features. The high level languages, on the other hand, are fully typed and have a large number of data types, data structures, and control structures. Machine dependent features are forbidden or carefully isolated, as in CS-4;

(4) Size of the language and compiler. The size and complexity of the language directly influences the effort required to learn the language and to implement a compiler for the language. The languages in this study range from very small (STRCMACS) to very large (CS-4). For some of the languages the actual size of the compiler in source language statements is known;

(5) Special system features. Most of the fifteen languages provide a number of features that would be particularly helpful for system implementation. These include inline assembly language, process management and real-time scheduling, bit and character data types, pointers and record structures, the ability to suppress type checking, reentrant or recursive procedures, and access to hardware registers;

(6) Error checking and debugging. Compilers for fully typed

languages can detect many errors during compilation that can not be detected until the debugging phase in the typeless languages. Typeless pointer variables are particularly troublesome. Languages that do not provide default declarations or automatic type conversion can also detect more errors at compile time.

A number of the languages provide special debugging tools, including traces of program variables, statement label flow history, execution statistics, timing information, and cross reference and attribute listings;

(7) Design support. Design support is broken down into three categories: modularity, modifiability, and reliability. Some of the features contributing toward modularity are a structured control structure, a data abstraction facility (as in CS-4), and independent compilation of procedures and functions. A macroprocessor and some form of "include" feature for copying source files into a program greatly enhances modifiability. High level data structures and operators also improve modifiability by making programs shorter and more readable.

Features contributing to reliability are full type checking, a data abstraction facility, a structured control structure, a small number of compiler-supplied defaults, and few or carefully isolated system features;

(8) Use of the language. This section includes information about the use of the language in large programming projects, what machines have compilers for the language, and how easily the compiler could be transported to other machines. Some of this information was found in [FRE75], the remainder was found in the language reference manuals.

The remainder of this chapter is devoted to the evaluations of the languages (listed in alphabetical order).

2.1. BLISS-11

2.1.1. LANGUAGE FEATURES

BLISS-11 [DEC74] is a systems programming language for the PDP-11 series that was developed by a group at Carnegie Mellon University with some assistance from Digital Equipment Corporation. Although the language is highly structured, it is typeless and generally low-level. BLISS-11 differs from conventional programming languages in several important ways. First, BLISS-11 is expression oriented, so that all control structures return a value. For example, P = (INCR I FROM 1 TO 10 BY 1 DO IF .A[.I] EQL 0 THEN EXITLOOP .I) is a legal BLISS-11 construction. Secondly, BLISS identifiers evaluate to a pointer to the named item, and not to the value of the item. A dot operator is provided for dereferencing these pointers. For example, if A is a BLISS identifier then the expression A evaluates to the address of item A, .A to the value of item A, and ..A to the value of the item pointed to by item A.

A. <u>Basic Data Types and Operators</u>

BLISS-11 is a typeless language. All operators operate on 16-bit words, and it is the user's responsibility to insure that the information contained in the operand word(s) is of the correct type for the operator. BLISS-11 allows five types of constants to appear in expressions: character strings, integers, real numbers, octal numbers, and pointers.

The following operators are provided for operating on 16-bit words:

arithmetic operators

+, -, *, /, unary minus

MOD, MAX, MIN

<expr-1> ^ <expr-2>

Shift operator yielding value of <expr-1> shifted left or right by <expr-2> bits. The sign of <expr-2> determines the direction of the shift.

<expr-1> ROT <expr-2>

Left or right circular shift.

relational operators

EQL, NEQ, LSS, LEQ, GTR, GEQ

EQLU, NEQU, LSSU, LEQU, GTRU, GEQU

Relational operators for signed and unsigned ("U") operands. The relational operators return an integer result (0 for false, 1 for true).

logical operators

NOT, AND, OR, XOR, EQV

Bitwise complement, and, or, exclusive or, and equivalence.

other

. <expr>

Pointer dereferencing operator yielding the object pointed to by the <expr>.

expr <pos,len>

Partword selector for extracting bits from a word.

<var> = <expr>

Assignment operator. The value of the expression is stored at the location pointed to by the <var>. Thus if A were a BLISS-11 identifier, the expression A = .A+1 would increment the value of A. Note that the pointer dereferencing operator must be used on right-hand side of the expression, but not on the left-hand side.

B. <u>Control Structures</u>

- IF <test-expr> THEN <expr> { ELSE <expr> } ;

  (Standard conditional.)

- BEGIN <expr-1>; ... <expr-k>; <expr-k+1> END ;

  (Compound expression.)

- WHILE <test-expr> DO <expr> ;
UNTIL

  (While and repeat loops with test performed before the body is executed.)

```
- DO <expr> WHILE <test-expr> ;
            UNTIL
```

(While and repeat loops with the test performed after the body is executed. The body will therefore be executed at least once.)

```
- INCR <var> FROM <e-1> TO <e-2> BY <e-3> DO <expr-body> ;
  DECR
```

(For loops. Programmer must choose a count-up or a count-down loop when the program is written.)

```
- CASE <expr-list> OF SET
    <expr-1> ;
        .
        .
        .
    <expr-k>
  TES ;
```

(Simple case statement. The expressions in the <expr-list> are evaluated, and then each is used to select some <expr-i> in the body of the CASE expression for execution.)

```
- SELECT <expr-list> OF NSET
    <select-expr-1> : <expr-1> ;
          .
          .
          .
    <select-expr-k> : <expr-k>
  TESN ;
```

(Select statement. The expressions in the <expr-list> are evaluated, and each one is then compared sequentially with the <select-expr-i>. If an expression matches some <select-expr-i> then the corresponding <expr-i> is executed. The keywords ALWAYS and OTHERWISE may be used in the <select-expr-i>; ALWAYS forces execution of its <expr-i>, OTHERWISE specifies that its <expr-i> is to be executed only if no preceding <expr-i> is executed.)

- ROUTINE <ident> ({<parameter-list>}) = <expr-body> ;

  (Standard function construct. Since all BLISS-11 constructs return a value, there is no procedure or subroutine construct. Functions may be recursive.)

- <ident> ({<arg-list>})

  (Call to a routine.)

- LEAVE <label> WITH <expr> ;

  (Exit the labeled construct with the value of the expression <expr>.)

- LEAVE <label> ;

  (Exit the labeled construct with a value of 0.)

- EXITLOOP <expr> ;

  (Exit the innermost loop with the value of <expr>.)

- RETURN <expr> ;

  (Return from body of a routine with the value of the <expr>.)

- SIGNAL <signal-expr> ;

  (Initiates scan of ENABLE blocks for a "handler" for condition <signal-expr>. The SIGNAL and ENABLE constructs provide a feature somewhat similar to user defined ON-conditions in PL/I.)

- ENABLE

  ```
    <expr-1> : <handler-expr-1> ;
        .
        .
        .
    <exp-k> : <handler-expr-k>
  ELBANE ;
  ```

  (Used in conjunction with the SIGNAL construct. On execution of a SIGNAL <signal-expr>, control passes to the most recently executed ENABLE block. The <signal-expr> is then compared with the <expr-i> in the ENABLE statement; if some <expr-i> matches the <signal-expr> then the <handler-expr> is executed, and control passes out of the block containing the ENABLE block. If no <expr-i> matches the <signal-expr> then control will pass to the next most recent ENABLE block, and the search for a handler continues. SIGNAL and ENABLE provide a "software interrupt" capability, although no return from the interrupt is possible.)

C. Data Structures

BLISS-11 has two constructs for creating more complex data structures. The first (STRUCTURE) defines a data structure and an access method for the data structure, and the second (MAP) is used to "map" or overlay a structure onto a previously unstructured block of core. The declaration

```
STRUCTURE <ident> [<parameter-list>] =
        [<structure-size-expr>] <access-method-expr> ;
```

defines the structure <ident> by specifying the number of storage locations required for the structure, and an expression defining an access method for the structure. The expressions defining the structure size and access method can use any of the parameters in the <parameter-list> of the structure. The structure <ident> can then be used to declare new objects of that type using the the OWN statement, or it can be mapped over some other variable. The statement

```
MAP <structure-ident> <identifier-list> <size> ;
```

maps the specified structure onto the identifiers in the identifier list. The identifiers can then be referenced as if they had been declared to have been structures of type <structure>. The MAP statement allows the programmer to access a block of core under a number of different formats.

For example, the following BLISS-11 segment defines a lower-triangular byte matrix structure:

```
BEGIN
  STRUCTURE LTRIAG[I,J] =
   [I*(I+1)/2]  (.LTRIAG + .I * (.I-1)/2 + .J - 1);
  OWN LTRIAG M[5,5];
  OWN N[15];
```

```
MAP LTRIAG N;
M[1,1] = N[1,1] = 16;
```

BLISS-11 has a predefined structure called VECTOR that can be used to declare one dimensional arrays, and the user can define arrays with more dimensions by using the STRUCTURE statement. Finally, the untyped pointers in BLISS can be used to create arbitrary linked data structures.

D. Other Features

BLISS-11 has several features that would make BLISS programs easy to modify. The BIND statement

BIND <ident> = <expression> ;

equates <ident> with the text of the <expression>. This text is used to replace any occurences of the <ident> in the rest of the source program. BLISS-11 also has a powerful macroprocessor that provides simple replacement macros, parameterized replacement macros, and recursive and iterated macros. Source text from a program library can be included into a BLISS program using the REQUIRE statement. BLISS-11 has no I/O facilities.

E. Runtime Environment

BLISS-11 is a low-level language and will probably run on a bare machine.

F. Syntax

BLISS-11 has a BNF grammar with approximately 150 productions.

2.1.2. CHARACTERISTICS

A. Machine Dependence

BLISS-11 is a systems programming language for the PDP-11 series and is highly machine dependent. The machine dependent features include inline assembly language instructions, the partword operator for extracting bits, and the TRAP, EMT, WAIT, and RESET statements for controlling the PDP-11.

B. Efficiency

BLISS-11 is quite efficient, and will compare favorably with assembly language programs.

C. Level of the Language

The BLISS-11 language is typeless and low-level.

D. Size of the Language and the Compiler

The language is small, and the compiler should be the same.

E. Special System Features

BLISS-11 provides the following system features:

(a) Assembly language statements can be inserted into a BLISS-11 program using the INLINE statement:

INLINE ("any character string").

The character string is passed unaltered to the assember.

(b) The programmer can request that local variables be allocated in machine registers using the REGISTER statement: REGISTER <ident>; . The variable is allocated in one of the machine registers, although the programmer has no control over which register is used.

(c) The LINKAGE statement gives the programmer control over the type of calling sequence generated for a function call. The user can specify that function parameters are to be placed on the runtime stack or in selected registers, and the language used to write the subroutine. Six calling sequences are available: BLISS (default), FORTRAN, INTERRUPT, EMT, TRAP, and IOT.

(d) BLISS-11 has six functions providing access to the hardware on PDP-11 machines:

TRAP(<trap-number>) - Generate program interrupts.
EMT(<trap-number>)

```
IOT(<trap-number>)
HALT()                    - Halt all execution.
RESET()                   - Reset all devices on the UNIBUS.
WAIT()                    - Wait for an interrupt.
```

(e) The ENABLE and SIGNAL constructs provide a type of software interrupt for handling user-defined exceptional conditions.

(f) BLISS-11 has pointer variables, a partword operator for extracting bits from a word, character strings, record structures, and the MAP feature for accessing a block of core under several different formats.

F. Error Checking and Debugging

Because of the absence of types, there is little that BLISS can do in the way of compile or runtime error checking. The BLISS-11 pointers are completely unrestricted, and it is therefore possible to create pointers that will generate addressing exceptions, cause branches into the middle of data, access data under the wrong format, and so forth.

BLISS-11 has a compiler option that will provide an interface for the SIX12 debugging package.

G. Design Support

(a) modularity

Modularity in BLISS-11 is good. BLISS-11 supports independent compilation of routines, and communication via GLOBAL variables or registers. User control over calling sequences makes interfacing with assembly language or FORTRAN routines fairly easy.

(b) modifiability

BLISS-11 has a very powerful macro processor and a large number of control structures. The BIND statement makes it easy to alter the constants used throughout a BLISS program. Finally, the REQUIRE statement allows the programmer to include source files into a program.

(c) reliability

BLISS-11 requires very careful programming because of the lack of type checking and the unrestricted pointers. It will be much harder to insure the reliability of a BLISS-11 program than an equivalent program written in a language like PASCAL or HAL/S.

H. Use

BLISS-11 has been implemented on the PDP-11 series, and the language could not be implemented on other machines unless the special system features for the PDP-11 were removed (TRAP, WAIT, RESET, and so forth).

2.2. C

2.2.1. LANGUAGE FEATURES

The language C [RIT74,KER74] is a systems programming language developed at Bell Laboratories by D. M. Ritchie. C is a structured, medium level language with a terse syntax and a profusion of built-in operators. The language was originally designed for the PDP-11 series, although it has since been implemented on other machines (HIS 6070 and the IBM 360 and 370 series). The UNIX operating system and a substantial portion of the software in the UNIX timesharing system are written in C.

A. Basic Data Types and Operators

C has four basic data types; INT, CHAR (single character), FLOAT and DOUBLE (single and double precision floating point). The language is fully typed, although automatic conversion between the four basic types is provided in many instances. In particular, a CHAR expression can be used anywhere that an INT expression can be used. Five types of constants are permitted in expressions: integers, character constants of one or two characters, strings of characters (treated as character arrays), and floating point numbers.

C has a large number of operators for manipulating the basic data types. The operators and the data types on which they operate are listed below:

logical operators (INT and CHAR operands only)

! <expr>	1 if <expr> = 0, and 0 otherwise.
~ <expr>	Bitwise complement of <expr>.
<e1> & <e2>	Bitwise AND of <e1>, <e2>.
<e1> ! <e2>	Bitwise OR.
<e1> ^ <e2>	Bitwise exclusive OR.
<e1> << <e2>	Left logical shift of <e1> by <e2> bits.
<e1> >> <e2>	Right arithmetic shift.
++ <variable>	Auto-increment and auto-decrement operators
-- <variable>	corresponding to the PDP-11 series machine

<variable> ++ instructions. In the prefix form the
<variable> -- variable is incremented or decremented by 1 and the value of the variable becomes the value of the expression. In the postfix form the value of the variable becomes the value of the expression, and the variable is then incremented or decremented by 1.

logical operators (all basic types)

<e1> ? <e2>:<e3>	Selection operator equivalent to if <e1> then <e2> else <e3>.
<e1> && <e2>	1 if <e1> and <e2> are non-zero, and 0 otherwise.
<e1> !! <e2>	1 if <e1> or <e2> is non-zero, 0 otherwise.
<e1> , <e2>	The expressions <e1> and <e2> are evaluated from left to right, and <e2> becomes the value of the entire expression.
SIZEOF <expr>	Size of the expression in bytes.

arithmetic operators

<e1> % <e2>	Remainder function (<e1> modulo <e2>). The operands <e1> and <e2> must be INT or CHAR.
+, -, *, /	Standard arithmetic operators. The operands may be INT, CHAR, FLOAT, or DOUBLE. Automatic conversion is performed between the types.

relational operators (All types)

=, != <, >, <=, >=	All the relational operators yield an integer result (1 or 0). All combinations of operand types are permitted, and conversion is performed between unequal types.

assignment operators

C has a standard assignment operator of the form <variable> = <expr>. Automatic type conversion is performed if the types do not match. In addition to this standard operator, C combines the assignment operator with many of the previously discussed operators. For each of the following operators, <variable> =op <expr> is equivalent to <variable> = <variable> op <expr>:

=+, =-, =*, =/

=>>, =<<

=&, =!, =^

B. Control Structures

- { <stmt-1>; ... <stmt-k>; }

  (Compound statement formed by placing statement in braces. Since C uses the characters { and } as part of the language syntax, we will use [and] to denote any optional features in the language.)

- IF (<expr>) <stmt-1>; [ELSE <stmt-2>;]

  (Conditional statement with optional ELSE part.)

- WHILE (<expr>) <stmt>;

  DO <stmt> WHILE <expr>;

  (Standard while loop with the loop test before and after the loop body.)

- FOR (<expr-1>; <expr-2>; <expr-3>) <stmt>;

  (For loop. The expression <expr-1> defines the loop variable and the initial value, <expr-2> the loop test, and <expr-3> the increment statement. For example:

```
SUM = 0;
FOR (I=0; I<n; I++) SUM =+ VECTOR[I];
```

)

- SWITCH (<case-expr>)

```
{ CASE <constant-expr-1>:  <stmt-list-1>;
    .                .                  .
    .                .                  .
    .                .                  .
  CASE <constant-expr-k>:  <stmt-list-k>;
  [ DEFAULT: <stmt-list>; ]
};
```

(Case statement with an optional DEFAULT clause. No two of the constant expressions may have the same value. The <case-expr> is evaluated, and the value is compared with the constant expressions in an unspecified order. If a matching constant expression is found then the corresponding <stmt-list> is executed; the DEFAULT <stmt-list> is executed only if no matching constant expression is found. Note: the case prefixes do not alter the flow of control within the SELECT statement. Thus, if <stmt-list-i> is selected for execution by the <case-expr>, then control will flow through <stmt-list-i> into <stmt-list-i+1> unless some statement in <stmt-list-i> causes an exit from the SELECT statement.)

- BREAK;

  (Exit the innermost WHILE, DO, FOR, or SWITCH statement.)

- CONTINUE;

  (Continue next iteration of the innermost WHILE, DO, of FOR statement.)

- GOTO <label-expression>;

  (Unconditional branch to a label within the current function.)

- RETURN [(<expr>)] ;

  (Return from current function with an optional result.)

- <type> <ident> (<parameter-list>) <body>

  (Standard function definition. For example:

```
INT FACTORIAL (N)
    INT N;
    RETURN (N<2 ? 1 : N*FACTORIAL(N-1));
```

  As the example illustrates, functions can be called recursively. All parameters are passed by value.)

C. Data Structures

C has three features for building more complex data structures from the basic data types:

(1) typed pointer variables

The statement

* <type> <ident>;

declares <ident> to be a pointer to an object of type <type>. The following operators are provided for manipulating pointers:

* <pointer-expr> - Yields object pointed to by the pointer expression.

& <variable> - Yields address of the variable.

<structure-pointer> -> <structure-member>
- Accesses the specified member of the structure pointed to by the structure pointer.

<pointer> + <integer-expr>
<pointer> - <integer-expr>
- When an integer is added to or subtracted from a pointer of type X, the integer is first multiplied by the length of an object of type X. Thus if P points into an array of record structures, then P+1 is a pointer to the next record structure in the array.

==, !=, <, >, <=, >=
- Pointers can be compared with other pointers or integers using the relational operators. Integers are multiplied by the object length (as discussed under the + operator).

(2) arrays

The statement

<type> <ident> [<#-of-elements>] { [<#-of-elements>] } ;

declares <ident> to be an array of <#-of-elements> objects of type <type>. Arrays can have an arbitrary number of dimensions. Array indexing begins at 0, and elements of an array are accessed using standard subscript notation:

<ident> [<subscript>] { [<subscript>] }

Arrays need not be fully dereferenced by the subscript operator. For example, if X was declared by the statement INT X[5][20][8] then X[3] yields a 20x8 integer array. Note: the assignment operator can not be used to copy an entire array from one variable to another.

(3) record structures

The statement

STRUCT <ident> { <type-declaration-list> };

declares <ident> to be a record structure composed of the objects listed in the <type-declaration-list>. The dot operator "." is used to access a member of a structure: <structure-name>.<member-name>. Note: The address operator & is the only other operator that can be applied to an entire structure. The assignment operator can not be used to copy an entire record structure, and entire structures can not be passed into functions as parameters or compared with other structures. A pointer to a structure can be passed into a function, however.

D. Other Features

C has an optional preprocessor pass which allows the user to include source files into the program text, and to use simple replacement macros. Files are included into the source program by the statement #INCLUDE "file-name". The statement

#DEFINE <ident> <character-string> is used to define simple replacement macros. All occurrences of the identifier in the source text are replaced by the character string.

C has no statements for performing I/O, but the C function library contains routines for formatted and unformatted I/O.

E. Runtime Environment

C requires a runtime stack because all functions are potentially recursive.

F. Syntax

The BNF grammar for C has approximately 120 productions.

2.2.2. CHARACTERISTICS

A. Machine Dependence

C has no machine dependent features and could be implemented on almost any machine.

B. Efficiency

C requires a runtime stack. C also converts all FLOAT expressions to DOUBLE expressions during the evaluation of any expression or function call. Various other automatic conversions are performed if the programmer mixes types in expressions. In all other respects C should compare favorably with assembly language programs.

C. Level of the Language

C is a medium level language. The language has records, arrays, typed pointers, structured control structures, and many operators.

D. Size of the Language and Compiler

C is a relatively small language with no complicated control structures. The compiler should also be fairly small.

E. Special System Features

C has typed pointers, record structures, recursive (and therefore reentrant) functions. The SIZEOF operator would be helpful when passing arrays or structures to assembly language routines. C also allows the programmer to request (via the REGISTER statement) that certain variables be allocated in machine registers instead of main storage. There is no way to select specific registers, however.

F. Error Checking and Debugging

Although the language is fully typed, C provides automatic type conversion between most of the data types. This will hide a number of errors (such as misspelling) unless the compiler prints warning messages when conversions are performed.

The manual does not indicate that any special debugging features are available.

G. Design Support

(a) modularity

C allows independent compilation of programs, and provides communication through external variables. The language also has a number of control structures.

(b) modifiability

C has a primitive macro processor, the #INCLUDE statement for including source files into a program, and the basic structured programming control structures.

(c) reliability

C programs are very difficult to read because of the terse syntax. Many operators are used both as binary and unary operators, with no relation between the operations being performed (e.g., & is used to take the address of a variable and as the logical AND function.) Spaces around operands are critical in some situations. The statements I=-J and I = -J perform completely different operations, for example.

The automatic type conversion performed by C can hide a number of errors caused by improper use of variables. Finally, the pointer variables in C require careful use. It is possible to generate pointers that will cause addressing errors when used, or to branch into the middle of the program's data area by using the GOTO statement with a pointer expression.

H. Use

C has been implemented on the PDP-11 series, the HIS 6070, and the IBM 360 and 370 series. The compiler is written in C itself, so the language could be implemented on other machines using standard bootstrapping techniques. C has been used extensively in the UNIX operating system and the software for the UNIX timesharing system.

2.3. CONCURRENT PASCAL

2.3.1. LANGUAGE FEATURES

CONCURRENT PASCAL [HAN75a,HAN75b,HAN75c] is a high level language developed by Per Brinch Hansen at the California Institute of Technology for use in writing operating systems. The language extends the PASCAL language with three facilities for concurrent programming: concurrent processes, monitors for providing controlled access to data structures shared by a group of processes, and data abstractions called classes. CONCURRENT PASCAL has all the basic data types and control structures of PASCAL, although some of the data structures have not been included. In particular, CONCURRENT PASCAL does not have the pointer or file type of sequential PASCAL.

A. Basic Data Types and Operators

CONCURRENT PASCAL has four basic data types: INTEGER, REAL, BOOLEAN, and CHAR (single character). Full type checking is performed at compile time, and no automatic conversions are performed between the basic types. The following types of constants are permitted in expressions: integer, real, boolean, character, and string (treated as an array of characters).

The operators and the data types on which they operate are listed below:

arithmetic operators and functions (INTEGER and REAL operands)

+, -, *	- Standard arithmetic operators for INTEGER or REAL operands.
/	- Division operator for REAL operands.
DIV, MOD	- Division and modulus operators for INTEGER operands.
ABS(<expr>)	- Absolute value of REAL or INTEGER expression.
SUCC(<expr>) PRED(<expr>)	- Functions yielding successor and predecessor of the INTEGER expression.

CONV(<expr>) - Converts INTEGER expression to REAL.
TRUNC(<expr>) - Truncates a REAL expression to INTEGER.

logical operators (BOOLEAN operands)

AND, OR, NOT - The BOOLEAN operators yield a BOOLEAN result.

relational operators (all basic types)

=, <>, <, >, <=, >=

- The two operands must have the same type. The relational operators yield a BOOLEAN result.

character operators

SUCC, PRED - Successor and predecessor functions.
CHR(<expr>) - Yields i-th character in the character set, where i is the value of <expr>.
ORD(<char>) - Ordinal position of the character in the character set.

B. <u>Control Structures</u>

- BEGIN <stmt-list> END
  (Compound statement.)

- IF <boolean-expr> THEN <stmt> { ELSE <stmt> }
  (Standard conditional with optional ELSE clause.)

- WHILE <boolean-expr> DO <stmt>
  (While loop.)

- REPEAT <stmt-list> UNTIL <boolean-expr>
  (Until loop. The body of the loop will be executed at least once.)

- CYCLE <stmt-list> END;
  (Unbounded repetition of the <stmt-list>.)

- FOR <var> := <expr-1> TO
DOWNTO <expr-2> DO <stmt>

  (For loops with implied increments of +1 and -1.)

```
- CASE <scalar-expr> OF
    <constant-list-1> : <stmt-1>
            .                 .
            .                 .
            .                 .
   <constant-list-k> : <stmt-k>
  END
```

(Case statement. The <scalar-expr> can be INTEGER, CHAR, BOOLEAN, or any user-defined scalar or subrange type (scalar and subrange types will be described later in Section C). The constant lists must contain constants of the same type as the <scalar-expr>. The <scalar-expr> is evaluated, and the constant lists are scanned to find a constant equal to the expression. If a match is found then the corresponding statement is executed; if no match is found then the effect of the CASE statement is undefined.)

```
- WITH <variable-list> DO <stmt>
```

(Executes <stmt> using the record variables in the <variable-list.> Any expression in <stmt> may refer to subcomponents of the records without fully qualifying the subcomponent. For example, if X is a record with subcomponents A, B, and C, then

```
WITH X DO BEGIN
   A := A + 1.0;
   B := A < 10.0;
   C := 'G'
END
```

is equivalent to

```
X.A := X.A + 1.0;
X.B := X.A < 10.0;
X.C := 'G';
```

)

```
- PROCEDURE {ENTRY} <proc-name>
      { (<parameter-list>) };  <proc-body>

  FUNCTION { ENTRY } <func-name>
```

{ (<parameter-list>) } : <type> ; <func-body>

(Procedure and function definitions. Neither may be recursive. If the ENTRY attribute is specified then the procedure or function may be called by an external PROCESS, MONITOR, or CLASS (see Section D for a discussion of these system types). The user can request that procedure parameters be passed by value or by reference, but all function parameters are passed by value.)

- <func-name> { (<argument-list>) }

  <proc-name> { (<argument-list>) }

  (Invoke a function or procedure.)

C. Data Structures

CONCURRENT PASCAL has seven constructs for creating more complex data structures from the basic data types:

(1) scalar type

The scalar type statement

```
TYPE <type-ident> = (<object-1>, ..., <object-k>) ;
```

defines an ordered set consisting of <object-1>, ..., <object-k>. For example:

```
TYPE MONTH = (JAN,FEB,MAR,APR,MAY,JUN,JUL,AUG,
              SEP,OCT,NOV,DEC) ;
```

The set is ordered, so the relational operators =, <>, <, >, <=, >=, the assignment operator :=, and the functions SUCC, PRED, and ORD can be applied to any scalar type. Note: the basic types INTEGER, CHAR, and BOOLEAN are predefined scalar types.

(2) subrange types

Subrange types are subranges of scalar types, and they also form ordered sets of objects. The statement

```
TYPE <type-ident> = <object-1> .. <object-m> ;
```

defines a subrange type. There must be a scalar type containing both objects, and the first object must be less than the second. For example:

```
TYPE SPRING = MAR .. MAY;
TYPE DIGIT = '0' .. '9';
TYPE INDEX = 0 .. 100;
```

All the operators for scalar types can be applied to subrange types.

(3) arrays

The statement

```
TYPE <type-id> = ARRAY [<dimension-list>] OF <type> ;
```

defines an array type. Arrays can have an arbitrary number of dimensions, and the <type> can be any type except a system type. The dimensions are specified by subrange types. For example:

```
TYPE MATRIX = ARRAY[1..3, 1..3] OF REAL;
VAR VECTOR : ARRAY[1..10] OF REAL;
VAR JOBSRUN : ARRAY[1968..1973, JAN..DEC] OF INTEGER;
```

Array elements are referenced by listing the subscripts in brackets:

```
<ident> [<subscript-list>] .
```

The relational operators = and <> can be use to compare two arrays of the same type, and the assignment operator := can be used to copy an entire array.

(4) sets

The statement

```
TYPE <type-ident> = SET OF <base-type> ;
```

defines a type consisting of all possible subsets of the <base-type>, which must be a scalar or subrange type. For example:

```
TYPE DAY = (M,T,W,TH,F,SA,S);   {Define scalar type}
VAR DAYSOFF : SET OF DAY;       {Now use it for a set}
VAR DIGITS : SET OF 0..9;
```

The following operators are available for manipulating set types:

```
[ <element-list> ]   - Set constructor yielding set.
                       The list may be empty.
OR, -, AND           - Set union, difference, and
```

intersection.

<=, >= - Tests on set inclusion.

IN - Membership operator yielding true if element is in set.

(5) record structures

A record type is declared with a statement of the form

```
TYPE <type-ident> = RECORD
  <member-1> : <type-1>
       .            .
       .            .
       .            .
  <member-k> : <type-k>
END ;
```

Records can contain an arbitrary number of members, and each member can be of any type except a system type. The following operators are provided for manipulating record types:

<record-var> . <member-name>

- Dot operator for accessing member of a record.

=, <> - Tests for equality (records must have same type).

:= - Assignment operator for copying an entire record.

The WITH statement discussed in Section B can be used to avoid qualifying each member of a record with the record name.

(6) queues

Queues, which are used within MONITORs to suspend and resume processes, are declared with a statement of the form

```
TYPE <type-ident> = QUEUE ;
```

A queue can only hold a single PROCESS, but arrays of queues can be defined. The following queue functions are available:

EMPTY(q) - Returns true if the queue is empty.

DELAY(q) - Delay the currently executing process in the queue (execution of the process is suspended and the MONITOR is freed for use by other processes).

CONTINUE(q) - Reactivate a stalled process. The currently executing process returns from the MONITOR. If the queue contains a process then that process resumes execution in the MONITOR routine that DELAYed it.

(7) system types

System types are defined with a statement of the form

```
                       PROCESS
TYPE <type-ident> =    MONITOR  { (<parameter-list>) }
                       CLASS

  <private-sector> <routine-entries> <initial-stmt>
```

The parameter list of a system type defines the constants and other system types which the system type can access. Data declared in the <private-section> is accessible only within the system type, and the <routine-entries> define a set of routines that may be called by other system types. The <initial-stmt> specifies any initialization to be performed when the system type is first activated.

A program in CONCURRENT PASCAL consists of an arbitrary number of independent, concurrently executing PROCESSes. Each PROCESS defines a data structure and a sequential program for operating on the data structure. A PROCESS can only communicate with another PROCESS by calling a MONITOR: MONITORS are used for synchronization and data sharing. A MONITOR also defines a data structure and an arbitrary number of operations that can be performed on the data structure by concurrent PROCESSes. A CLASS is similar to a MONITOR, except that a CLASS may only be accessed by a single PROCESS.

System types are initially activated with the INIT statement:

```
INIT <sytem-type> { (<parameter-list>) } ;
```

The INIT statement defines the access rights (the other system types which can be accessed) by the system type, and executes the initial statement of the system type.

D. Other Features

CONCURRENT PASCAL requires the declaration of all variables, functions, and procedures prior to their use. The language has a declaration of the form CONST <ident> = <expr>;
for declaring program constants. The identifier can be used in any expression, but the value of the identifier can not be altered. CONCURRENT PASCAL does not support the pointer type, the "variant field" in records, or the dynamic storage allocation provided by sequential PASCAL. CONCURRENT PASCAL does not provide dynamic arrays or even array dimensions as parameters, as in the following FORTRAN segment:

```
SUBROUTINE XYZ(ARRAY,N,M)
INTEGER N,M,ARRAY(N,M)
```

Thus, it is not possible to write a CONCURRENT PASCAL program that manipulates arrays of arbitrary sizes. Finally, the language does not permit external functions or procedures: a CONCURRENT PASCAL program consists of a main program and an arbitrary number of nested functions and procedures, and the entire program must be compiled as a unit.

E. Runtime Environment

CONCURRENT PASCAL does not require a runtime stack, since recursive procedures and functions are not permitted. The language does not require a dynamic storage allocator either, since the pointer type and the NEW statement of sequential PASCAL have been eliminated. However, CONCURRENT PASCAL does needs a runtime executive for time-slicing concurrent processes.

F. Syntax

CONCURRENT PASCAL has a BNF grammar with approximately 150 productions.

2.3.2. CHARACTERISTICS

A. Machine Dependence

The UNIV attribute on procedure and function parameters can be used to write machine dependent programs. In all other respects CONCURRENT PASCAL is not machine dependent, and could be implemented on almost any machine.

B. Efficiency

CONCURRENT PASCAL is an efficient programming language. The language requires no runtime stack or dynamic storage allocation, and the language features have been carefully selected to permit efficient implementation of the language. Sets can be represented by bits strings; the set union, intersection, and difference operators can then be implemented in just a few instructions. Scalar and subrange types are equivalently simple. The structured control structures also permit better code optimization.

The manual for the PDP-11/45 implementation of CONCURRENT PASCAL contains tables indicating the execution times for many of the operators and control structures. These tables can be used by the programmer to minimize the number of expensive constructs in a program (for example, the DELAY and CONTINUE statements causing process switching take approximately 100 times as long to execute as an integer assignment operation).

C. Level of the Language

CONCURRENT PASCAL is a high level language.

D. Size of the Language and Compiler

The CONCURRENT PASCAL language is moderate in size. The compiler (which is written in sequential PASCAL) is only 8500 statements.

E. Special System Features

CONCURRENT PASCAL has record types, the set type (which can be used as bit strings), and the system types PROCESS, MONITOR, and CLASS for concurrent programming.

Another useful feature is UNIV parameters in procedures and functions. Declaring a parameter to be UNIV suspends the normal type checking that would be performed for the parameter, and thus allows the programmer to access a block of core under a number of different formats. For example, an array of characters could be passed into a procedure in which the corresponding formal parameter was declared to be an array of integers. Within the procedure body the formal parameter would be treated as an array of integers.

F. Error Checking and Debugging

CONCURRENT PASCAL performs full type checking at compile time for any program not using UNIV parameters. The CONST feature permits the declaration of "read only" variables. CONCURRENT PASCAL also has a hierarchical structure that forces the programmer to specify the access rights of all system types, and the compiler enforces these access rights. The subrange types also allow the implementation to perform runtime checks on variables to insure that the values are within the subrange. Such a feature would be very helpful in a diagnostic compiler.

The manual for CONCURRENT PASCAL does not indicate that any special debugging tools are available.

G. Design Support

(a) modularity

Modularity in CONCURRENT PASCAL is fair. The language has a full set of structured control structures, and internal procedures and functions are provided. However, CONCURRENT PASCAL does not permit external procedures or functions. This makes it costly to use existing programs (in a system library, for example), since the programs must be recompiled each time they are used.

(b) modifiability

As discussed previously, CONCURRENT PASCAL has no provisions for external procedures or functions. This would be a serious weakness in large systems (10,000 lines), where the most trivial modification in one of the programs would require the recompilation of the entire system. However, CONCURRENT PASCAL does have the CONST feature for declaring program constants, high level data structures and operators, the subrange type, and the control structures for structured programming. The CLASS and MONITOR types also provide a data abstraction facility. All these features make programs easier to read and modify.

(c) reliability

CONCURRENT PASCAL performs complete type checking at compile time (including procedure and function parameters). CONCURRENT PASCAL is also a high level and well structured language, so that programs should be smaller and more self-documenting than programs written in languages with fewer data or control structures. It should be considerably easier to write reliable programs in CONCURRENT PASCAL than in a language like FORTRAN.

H. Use

CONCURRENT PASCAL has been implemented on the PDP-11/45. The compiler is written in sequential PASCAL, so the language could easily be transported to other machines. CONCURRENT PASCAL has been used to implement part of the SOLO operating system (a single-user operating system for the PDP-11/45).

2.4. CS-4 Base Language

2.4.1. LANGUAGE FEATURES

CS-4 [INT75a] is a large, general purpose language currently being developed by Intermetrics for the Navy. The language is fully typed, block structured, and offers many of the features found in PL/I and HAL/S. CS-4 is an extensible language, and many of the high level features in the language are constructed from lower level features using the CS-4 data abstraction facility.

Because CS-4 is currently under development, only the CS-4 base language will be examined in this report (in the remainder of this section the CS-4 base language will be referred to as CS-4).

A. Basic Data Types and Operators

CS-4 has ten basic data types: INTEGER, REAL, FRACTION, COMPLEX, VECTOR (vector of REALs), MATRIX (NxM matrix of REALs), BOOLEAN, STATUS, SET, and STRING (fixed and varying length ASCII character strings). The STATUS type is equivalent to the PASCAL scalar type. Mixed mode arithmetic expressions are permitted, but in general no automatic type conversions are performed. Five types of literals can appear in CS-4 expressions: integer, real, boolean, status, and string.

The operators for manipulating these data types are listed below:

arithmetic operators (INTEGER, REAL, FRACTION, and COMPLEX operands)

+, -, *, /, **

IDIV	- Integer division for integer operands.
ABS	- Absolute value.
SGN	- Signum function.
SQRT	- Square root function for real and fraction operands.

FLOOR, CEIL - Floor and ceiling functions for real operands.

REAL-EQ
REAL-NE
REAL-LT
REAL-GT
REAL-LE
REAL-GE - Variable precision comparison functions for real operands. The relational operators can be used for fixed precision comparisons.

FRACTION-EQ
•
•
•
FRACTION-GE - Similar functions for fractions.

COMPLEX-EQ
COMPLEX-NE - Similar functions for complex operands.

REALPART, IMAGPART
- Real and imaginary part of a complex operand.

CONJUGATE - Complex conjugate.

ANGLE - Angle in polar coordinates of a complex operand.

MAG - Magnitude of a complex operand.

Log, exponential, and normal, inverse, hyperbolic, and inverse hyperbolic trigonometric functions are available for real operands.

boolean operators

NOT, AND, OR, XOR, NAND, NOR, EQV
- All the boolean operators yield a boolean result.

relational operators

=, ~=, <, >, <=, >=
- All the relational operators yield a boolean result. The operands being compared must have the same type. The operators = and ~= can be applied to any of the basic data types, but <, >, <=, >= can only be used with INTEGER, REAL, FRACTION, or STATUS operands.

status operators

PREDECESSOR, SUCCESSOR
- Successor and predecessor functions.

string operators

FLAVOR - Determines string type (fixed or varying).

LENGTH - Returns length of a fixed length string.

CURRENT-LENGTH - Returns length of a varying string.

MAX-LENGTH - Returns maximum length for a varying string.

<string-var> (<subscript>)
- Pseudo operator for accessing single characters in a string.

SUBSTR - Pseudo-variable for accessing substrings.

!! - Concatenation.

ASCII - Converts a string of characters to an array of integers.

PAD - Pads blanks onto the end of a string.

vector operators

<vector-var> (<subscript>)
- Accesses element of a vector.

+, - - Element-wise addition and subtraction.

* - Vector dot product.

OUTER - Vector outer product.

CROSS - Vector cross product.

VECTOR-SIZE - Returns length of a vector.

MAG - Magnitude of a vector.

UNIT - Unit vector.

VECTOR-EQ
VECTOR-NE - Variable precision comparison functions.

matrix operators

<matrix-var> (<subscript>,<subscript>)

+, - - Element-wise addition and subtraction.

* - Mattix dor product. The * operator can also be used to form the dot product of compatible matrices and vectors.

TRACE, TRANSPOSE, DETERMINANT, INVERSE

- Standard matrix operators.

MATRIX-SIZE - Returns length of first or second dimension.

MATRIX-EQ, MATRIX-NE
- Variable precision comparison functions.

set operators

NOT, AND, OR, NAND, NOR, XOR
- Set complement, intersection, union, complemented intersection and union, and exclusive union.

SUBSET - Determines if a set is a subset of another.

EMPTY - Determines if a set is empty.

<set-var> (<set-member>)
- Returns TRUE if the member is contained in the set.

B. <u>Control Structures</u>

- BEGIN <stmt-list> END

  (Compound statement. Any data declared within the BEGIN statement is local to the BEGIN statement.)

- IF <boolean-expr> THEN <stmt-list> { ELSE <stmt-list> } FI

  (Conditional statement with optional ELSE part.)

- CASE <case-expr>

```
  CASE <case-expr>
     OF <constant-list> :: <stmt-list>
           .                  .
           .                  .
           .                  .
     OF <constant-list> :: <stmt-list>
     { OTHERWISE <stmt-list> }
  END
```

  (Case statement. The <case-expr> can be an INTEGER, STRING, or STATUS expression, and the constant lists must contain constants of the same type as the <case-expr>. The <case-expr> is evaluated, and the constant lists are scanned to find a constant equal to the expression. If a match is found then the corresponding statement list is executed; if no match is found then the OTHERWISE clause is executed.)

```
- WHILE <boolean-expr> REPEAT <stmt-list> END
```

(Standard while loop.)

```
- FOR { <var> IS }  INTEGER (RANGE: <expr> THRU <expr>)
                    STATUS (<status-literal-list>)
    { WHILE <boolean-expr> } REPEAT <stmt-list> END
```

(For loop specifying a number of iterations of a statement list. No loop variable is required if the loop body does not need one. If a loop variable is specified then its value may not be altered by the loop body.)

```
- UPDATE (<shared-variable-list>) <stmt-list> END
```

(Update block for controlling access to shared variables by concurrent tasks. A variable declared with the SHARED(PROTECTED) attribute may only be referenced in an update block, and a task executing an update block will be stalled until the locked variables in the update block are no longer being accessed in an UPDATE block of any other task.)

```
- GOTO <label>
```

(Unconditional transfer. The <label> can not be the label of a statement located outside of the procedure that contains the GOTO statement.)

```
- EXIT <label>
```

(Exits the BEGIN block, UPDATE block, WHILE or FOR loop having the specified label.)

```
- RETURN { <result-expr> }
```

(Return from a procedure or function.)

```
- <handler-name> : PROCEDURE ({ <parameter-list> })
                        ATTR (HANDLES (<signal-name-list>)):
                   <stmt-list>
                   END <handler-name>
```

(Declaration of a signal handler. Signals and signal handlers are similar to PL/I ON-conditions and ON-units, respectively. A signal can be generated by a hardware interrupt, runtime error-checking code, or a SIGNAL statement. Signals generated with the SIGNAL statement can pass parameters to a signal handler. Signal handlers can handle an arbitrary number of signals.)

- SIGNAL <signal-name> { (<parameter-list>) }

  (Raises the specified signal. If there is an active signal handler for the signal then it will be invoked. The parameter list can be used to pass additional information to the handler.)

- RESIGNAL

  (Can only appear in a signal handler. The RESIGNAL statement raises the signal that caused the signal handler to be invoked.)

- ABORT to <label>

  (Can only appear in a signal handler. The <label> must be the label of a statement in the block containing the signal handler. The ABORT statement transfers control to the labeled statement, thereby terminating execution of the handler and all dynamically intervening procedures between the handler and the origin of the signal.)

- <proc-name> : PROCEDURE ({ <parameter-list> })

```
                        OPEN
      { <type> }   ATTR (CLOSED) ;
                        MOPEN
    <stmt-list>
    END <proc-name>
```

  (Definition of a procedure or a function. The <parameter-list> defines the procedure parameters and indicates for each parameter the method used to pass the parameter (call by value, reference, or name) and whether the parameter is to be used as an input, output, or input/output parameter. Parameters can be declared to be optional by specifying a keyword to identify the optional parameter and a default value to be used when the parameter is not supplied.

If the procedure is declared with the OPEN attribute then the procedure body will be substituted inline whenever it is invoked: no calling sequence will be generated. A normal procedure call is generated whenever a CLOSEd procedure is invoked. Finally, procedures declared as MOPEN are both OPEN and "mode-unresolved", that is, the type information used in the declaration of procedure parameters and in the body of the procedure need not be complete. When the procedure is substituted inline at the point of invocation, the type of the actual arguments is used to specify the type information for the procedure body. The MOPEN attribute provides a macro-like capability.

Procedures and functions can not be recursive.)

C. Data Structures

CS-4 has four constructs for creating more complex data structures from the basic data types:

(a) data abstractions

The MODE statement for defining CS-4 data abstractions requires the user to specify the data representation for the new mode and a set of procedures (operators) for manipulating the data representation.

```
<mode-name>:  MODE ({ <parameter-list> })
                 ATTR( CAPABILITY( <proc-name-list> ));
              <data-representation>
              <proc-definition>
                      •
                      •
                      •
```

```
&lt;proc-definition&gt;
END &lt;mode-name&gt;
```

The <mode-name> can then be used in type declarations to define objects with the new type. The <parameter-list> is used to "tailor" the new type to the needs of the program referencing the type. The parameters can be constants to be used in array declarations or elsewhere, or types to be used in type declarations. For example, we could define a new mode called STACK with two parameters -- one indicating the size of the stack, and one indicating the type of objects to be stored in the stack. The mode STACK could then be invoked to define a stack of integers, or reals, or boolean data.

The data represention section defines the actual representation used for the object, and the CAPABILITY section lists all of the procedures (operators) that can be used to manipulate the object. The data defined in the representation section can only be accessed by these procedures.

The assignment operator := and the relational operators =, ~= can be used to copy or compare entire data abstractions, as long as the two operands are compatible.

(b) arrays

Arrays are declared with a statement of the form

```
VARIABLE <ident> IS ARRAY( <dimension-list>, <type> )
```

Arrays can have an arbitrary number of dimensions, and each dimension is specified by a subrange of the integers or a STATUS set. For example;

```
VARIABLE XYZ IS ARRAY( [0 TO 7, STATUS("A","B","C")],
                                              BOOLEAN)
```

Array elements are referenced using the subscript operator <ident> (<subscript-list>). The type of the subscripts must match the type of the corresponding dimension. For example, XYZ(3,"B") is a legal array reference for the array in the previous example. As in PL/I and HAL/S, a * can be used as a subscript to reference all of the corresponding dimension.

The assignment operator and the relational operators =, ~= can be used to copy or compare compatible arrays.

(c) structures

CS-4 structures are delared with the statement

VARIABLE <ident> IS STRUCTURE (<member-list>)

The identifiers used to define the members need not be distinct from identifiers used elsewhere in the program. The dot operator is used to access members in a structure: <structure-var> . <member> . The assignment operator and the relational operators =, ~= can be used to copy or compare compatible structures.

(d) unions

The declaration of union variables is similar to the declaration of structured variables:

VARIABLE <ident> IS UNION(<member-list>)

The <member-list> defines the set of possible types that the union variable can represent. A union variable has a "field tag" indicating which member of the <member-list> is currently being stored, and the value for that member. The field tag of a union variable can be read using the built-in function TAG, which returns a STATUS literal indicating the name of the member. The value of a union variable can be accessed using the $ operator and the current field tag: <union-var> $ <field-tag> . For example;

```
{Define U as a union of integer, string, and boolean.}
VARIABLE U IS UNION (I IS INTEGER(RANGE: 1 THRU 10),
                     STR IS STRING(20,"VARYING"),
                     B IS ARRAY(0 THRU 3, BOOLEAN))
                [STR: 'A B C']  {Initial value for U.}
{At this point we have  TAG(U) = "STR"    }
{and U$STR = 'A B C' .                    }

   .
   .
   .
```

```
CASE TAG(U)
  OF "I" :: U$I := U$I+1
  OF "STR" :: U$STR := ´Z´
  OF "B" :: U$B(3) := FALSE
END
```

The relational operators =, ~= can be used to compare two union variables, and the assignment operator can be used to change the value of a union variable. However, the only way to change the field tag of a union variable is to assign it another union variable that already has desired field tag. This seriously restricts the usefulness of the UNION type.

D. Other Features

CS-4 is a block structured and fully typed language. Complete type checking (including procedure parameters) is performed at compile time. The language also has a CONSTANT attribute for declaring program constants.

CS-4 has an operating system interface that provides I/O and process management capabilities. The I/O system includes a hierarchical file system, file protection, and sequential, direct access, and indexed sequential files. The process management system provides features for scheduling processes, terminating processes, and communicating between processes. No additional language statements are required to support the operating system interface: the CS-4 MODE declaration is used to define data abstractions for files and processes.

E. Runtime Environment

CS-4 needs routines for process management, interprocess communication, I/O, and interrupt handling. A runtime stack or dynamic storage area will also be required to support the string concatenation operator !!.

F. Syntax

The BNF grammar for the CS-4 base language has approximately 500 productions.

2.4.2. CHARACTERISTICS

A. Machine Dependence

The language has several machine dependent features, including user specified allocation of data items, inline assembly language code, and user control over calling sequences. However, all of the machine dependent features have been carefully isolated. Inline assembly language, for example, is restricted to a special class of procedures called MPROCEDUREs.

B. Efficiency

CS-4 should be moderately efficient. It has many high-level operators and a structured control structure, so a great deal of optimization can be performed. The user can also request that procedures be expanded inline, so that there will be very little overhead in the use of data abstractions.

C. Level of the Language

CS-4 is a high level language.

D. Size of the Language and Compiler

The CS-4 base language is large and will require a large compiler. The full CS-4 language will require a very large compiler.

E. Special System Features

The language has a large number of special system features. The MPROCEDURE statement permits the user to declare structures that include information about the allocation of the structure members (bit or byte position within a word, and storage alignment). The MSTRUCTURE can also specify the absolute storage location at which the structure is to be allocated.

The MPROCEDURE statement provides the capability of writing procedures which contain assembly language code. User control over calling sequences is provided by the EPROCEDURE (external procedure) declaration, which permits the user to specify which registers will be modified by the called procedure and how parameters should be passed.

CS-4 also has the data abstraction facility. When combined with the MSTRUCTURE statement, data abstractions can be created for bit strings and pointers. The language also has records, arrays, character strings, signal handlers for processing exceptional conditions, the UPDATE block for controlling access to shared data, and the operating system interface (which includes I/O facilities and real-time process scheduling).

F. Error Checking and Debugging

CS-4 performs complete type checking at compile time (including procedure parameters), and provides no default declarations or automatic type conversion. This will allow many program errors to be detected during compilation.

Runtime checks are performed for many conditions (such as array subscript errors, CASE statements, and division by zero) unless the programmer uses compiler directives to disable the checking. The signal handlers also provide the user a means of intercepting runtime errors.

The language manual does not indicate that any special debugging tools are available.

G. Design Support

(a) modularity

CS-4 is a modular, block structured language. The language has a structured control structure, the MODE declaration for defining abstract data types, procedures can be separately compiled, and BEGIN blocks can be used to declare local data.

(b) modifiability

CS-4 programs should be easy to modify. The language is well structured, with a large number of data types, and a data abstraction facility. Status variables can be used to improve the readability of programs.

(c) reliability

The language has a number of features that would aid in the writing of reliable programs. It is well structured, many data types are provided, full type checking is performed, declaration of variables is mandatory, no automatic type conversion is performed (other than mixed-mode arithmetic), and there are only five compiler-supplied defaults for the entire base language.

H. Use

CS-4 is currently under development and has not been used for any major programming projects.

2.5. FLECS

2.5.1. LANGUAGE FEATURES

FLECS [BEY75a,BEY75b] is a preprocessor for Fortran developed by T. Beyer at the University of Oregon. FLECS supports all features of ANSI standard Fortran IV, and provides a large number of structured programming constructs. No special characters (e.g. $, %) are used to delimit the structured programming constructs. In the remainder of this section, the FLECS language is considered to be Fortran IV augmented by the FLECS preprocessor.

A. <u>Basic Data Types and Operators</u>

FLECS supports the five basic data types of Fortran IV: INTEGER, REAL, DOUBLE PRECISION, COMPLEX, and LOGICAL. The language permits mixed-mode expressions and will automatically convert between integer, real, and double precision numbers. Constants used in expressions can have the following types: integer, real, double precision, complex, logical, and character strings.

The operators and the data types on which they operate are listed below:

arithmetic operators (INTEGER, REAL, and DOUBLE PRECISION operands)

+, -, *, /, **

logical operators (LOGICAL operands)

.NOT., .AND., .OR.

relational operators

.EQ., .NE.	All types.
.LT., .LE., .GT., .GE.	INTEGER, REAL, or DOUBLE PRECISION operands only.

B. <u>Control Structures</u>

Note: In all the following control structures the symbol

<body> may be replaced by <stmt> or <stmt-1> ... <stmt-k> FIN. For example:

```
     WHEN (I .LT. MAXVAL) CALL PROCESS1(I,J)
     ELSE CALL BADVAL(I)
          I = MAXVAL
          RETURN
          FIN
```

```
- IF (<logical-expr>) <body>
```

(Simple if statement.)

```
- WHEN (<logical-expr>)
    <body>
  ELSE
    <body>
```

(Compound if statement.)

```
- UNLESS (<logical-expr>) <body>
```

(Equivalent to IF (.NOT. <logical-expr>) <body>; the <body> is executed if the <logical-expr> is false.

```
- WHILE (<logical-expr>) <body>
  UNTIL
```

(While and until loops with test performed before execution of the <body>.)

```
- REPEAT WHILE (<logical-expr>) <body>
         UNTIL
```

(While and until loops with tests performed after execution of the <body>. The <body> will therefore be executed at least once.)

```
- CONDITIONAL
    (<logical-expr>)  <body>
         .
         .
         .
    (<logical-expr>) <body>
    { (OTHERWISE) <body> }
  FIN
```

(LISP-like conditional statement. The

<logical-expr>'s are evaluated sequentially until some expression evaluates to .TRUE., and the corresponding <body> is then executed. The <body> of the optional OTHERWISE clause is executed only if all preceding <logical-expr> evaluated to .FALSE.

```
- SELECT (<select-expr>)
    (<expr>) <body>
        .
        .
        .
    (<expr>) <body>
    { (OTHERWISE) <body> }
  FIN
```

(Case statement. The <select-expr> is compared sequentially with the <expr>'s in the body of the SELECT statement. The first <body> whose <expr> matches the <select-expr> is executed, and all remaining bodies are skipped over. The <body> of the OTHERWISE clause is executed only if no preceding <expr> matched the <select-expr>.)

- DO (<variable> = <expr-1>, <expr-2> {, <expr-3>}) <body>

  (For loop with optional increment.)

- TO <internal-suproutine-name> <body>

  (A parameterless, internal subroutine. The subroutine name can contain any number of letters, digits, or hyphens, as long as it begins with a letter, and contains at least one hyphen. For example: INITIATE-VEHICLE-TRACKING.)

- <internal-subroutine-name> (Call of an internal subroutine. Note that no parameters can be passed to the subroutine.)

FLECS also supports the control structures of standard Fortran: the logical and arithmetic IF, the DO statement, the simple, ASSIGNed, and computed GOTO, and FUNCTIONS and SUBROUTINES. The section in this chapter concerning

Interdata Fortran V gives a detailed description of these constructs.

C. Data Structures

FLECS has only one feature for building more complex data types: arrays of up to 7 dimensions. The declaration

DIMENSION <ident> (<dimension-list>)

declares <ident> to be an array. Elements of an array are accessed using standard subscript notation <ident> (<subscript-list>).

D. Other Features

FLECS is essentially a Fortran language with some additional constructs for structured programming. The language has no block structure or recursion. FLECS provides statement functions, EQUIVALENCE, COMMON, and DATA statements, and the Fortran I/O statements. Comments are denoted by a "C" in the first column of the input card. FLECS also produces a "prettyprinted" output listing - statements are automatically indented to show program structure.

E. Runtime Environment

FLECS has no dynamic storage allocation or recursion, so no stack or heap is needed. Except for I/O and type conversion routines, FLECS should run on a bare machine.

F. Syntax

Fortran IV (and therefore FLECS) has a BNF grammar, but a compiler would probably not use it. Fortran compilers tend to use ad hoc compiling techniques.

2.5.2. CHARACTERISTICS

A. Machine Dependence

ANSI standard Fortran IV (and therefore FLECS) is fairly machine independent. Fortran programs can usually be transported to different machines with only minor modifications (e.g. different I/O unit numbers).

B. Efficiency

Fortran IV formatted I/O must be performed interpretively and is therefore quite slow. In all other respects Fortran IV and FLECS are efficent programming languages. We note, however, that the additional structuring of FLECS programs that would be very helpful to a code optimizer is not available to the Fortran compiler; all the structured statements are converted to IF and GOTO statements before reaching the compiler.

C. Level

FLECS is a medium level language.

D. Size of Language and Compiler

Because of the EQUIVALENCE statement, the unstructured nature of Fortran programs (optimization is difficult), and the preprocessor pass, FLECS will require a fairly large compiler.

E. Special System Features

FLECS has no special systems features.

F. Error Checking and Debugging

Fortran compilers have traditionally had very poor compile and runtime diagnostics, so FLECS diagnostics will probably be poor. The preprocessor phase of FLECS does print error messages when illegal FLECS statements are detected.

G. Design Support

(a) modularity

FLECS supports independent compilation of subroutines and functions, and communication through COMMON blocks.

(b) modifiability

FLECS has a large number of structured programming constructs. However, the language has no macroprocessor, no feature like the PASCAL constant statement for declaring program constants, no significant features for constructing complex data structures, and no "include" statement for copying source files.

(c) reliability

The structured programming constructs make FLECS a great improvement over Fortran IV. However, FLECS has no character or string operators and data types, and does not have sufficient data structuring capabilities. The lack of these features requires FLECS programs to simulate any character processing, list processing, or record processing with Fortran code. FLECS programs will therefore tend to be longer than necessary and more difficult to understand.

H. Use

The FLECS preprocessor is written in Fortran and could be implemented on almost any machine. FLECS is available on the CDC 6000, 7000, and Cyber series, the IBM 360 and 370 series, the PDP 8, 10, and 11, and the UNIVAC 1100 series. The source code for FLECS is available from its author (T. Beyer) at a nominal cost.

2.6. HAL/S

2.6.1. LANGUAGE FEATURES

HAL/S [INT75b,MAR74] is a high-level aerospace language developed by Intermetrics for the Space Shuttle program. Although the language is a dialect of PL/I, several of the more serious weaknesses in the PL/I language have been eliminated (for example, HAL/S pointers are fully typed, procedure parameters are checked for valid type, and the programmer must specify which parameters will be assigned values by the procedure body). Extensive subscripting capabilities, matrix and vector operators, and control structures for real-time control and concurrent processes are also provided.

A. Basic Data Types and Operators

HAL/S has eight basic data types:

INTEGER
SCALAR - floating point numbers
VECTOR - 1xN vector of SCALAR objects
MATRIX - NxN matrix of SCALAR objects
BIT - bit string
CHARACTER - variable length character string
BOOLEAN
EVENT - binary semaphores for process control. An event may be latched or unlatched; a latched event holds its value of TRUE of FALSE until set or reset, an unlatched event remains FALSE until signaled, whereupon it momentarily toggles to true, and then reverts back to FALSE. Process scheduling is invoked any time that an event is set, reset, or signaled.

Some implicit conversion is performed between these basic data types, and a set of conversion functions is provided: the functions INTEGER, SCALAR, VECTOR, MATRIX, BIT, CHARACTER, SINGLE, and DOUBLE provide conversion between the data types and possible precisions.

The operators and the data types on which they operate are listed below:

arithmetic operators (INTEGER, SCALAR, and MATRIX operands)

+, -, /
blank - multiplication
* - cross product
. - dot product

Note: some combinations of the operand types are not permitted.

bit operators (BIT and EVENT operands)

AND, OR, NOT
CAT - concatenation
SUBBIT$_{i\ TO\ j}$(bit-expr) - pseudo-variable for inserting or extracting bits.

character operators (CHARACTER operands)

CAT - concatenation
char-expr$_{i\ TO\ j}$ - substring insertion or extraction

boolean operators (BOOLEAN operands)

AND, OR, NOT

relational operators (all types)

=, ~ =
<,>,<=,>= - only for INTEGER, SCALAR, or CHARACTER operands

B. <u>Control Structures</u>

- IF <expr> THEN <basic-stmt> ELSE <stmt>;
  (Standard conditional, but basic-stmt may not be an IF statement.)

- DO; <stmt-list> END ;

(Compound statement.)

```
- DO WHILE <expr>; <stmt-list> END ;
     UNTIL
```

(Standard while and repeat loops.)

```
- DO CASE <arith-expr>; { ELSE <stmt>; }
            <stmt-1>; ... <stmt-k>;  END ;
```

(Simple case statement.)

```
- DO FOR <var> = <expr-list> { WHILE <expr> };
                               UNTIL
            <stmt-list> END ;
```

(For-loop with list of values to be assigned to <var>.)

```
- DO FOR <var> = <expr> TO <expr>
                  { BY <expr> }  { WHILE <expr> } ;
                                   UNTIL
            <stmt-list> END ;
```

(Standard for-loop with optional WHILE or UNTIL clauses.)

```
- EXIT <label>;
```

(Exit the DO group specified by the label.)

```
- REPEAT <label>;
```

(Continues next iteration of the specified DO group.)

```
- GOTO <label>;
```

(Branch to label in current namescope - can not be used to branch out of a procedure body.)

```
- RETURN { <expr> };
```

(Return from a procedure or function.)

```
- CALL <identifier> { (<expr-list>) }
                 { ASSIGN (<variable-list>) };
```

(Call statement for a procedure - only those variables in the ASSIGN list may be altered by the procedure.)

```
- <proc-name>: PROCEDURE { (<ident-list>) }
                         { ASSIGN (<ident-list>) }
```

```
                { EXCLUSIVE } ;
                  REENTRANT
       <procedure-body> CLOSE <proc-name> ;
```

(Procedure definition specifying input arguments, output arguments (the ASSIGN list). If the EXCLUSIVE attribute is specified then any concurrent task attempting to execute the procedure will be blocked as long as any other task is executing it.)

```
- <function-name>: FUNCTION { (<ident-list>) }
                    <type> { EXCLUSIVE } ;
                             REENTRANT
                 <function body> CLOSE <function name> ;
```

(Standard function definition, but function body may not cause side effects by altering the input parameters (there is no ASSIGN list).)

```
                       AT <expr>
- SCHEDULE <ident> {   IN <expr>           }
                       ON <event expr>
           PRIORITY (<expr>)  { DEPENDENT }
           { , REPEAT AFTER <expr> }
                      EVERY
             WHILE <event expr>
           { UNTIL <event expr> } ;
             UNTIL <expr>
```

(Scheduling statement for concurrent tasks. A task may be scheduled immediately, AT a specific time, IN a certain number of clock ticks, or ON the value of an event. The PRIORITY is used in scheduling ready tasks. If the DEPENDENT attribute is used then the scheduled task will be terminated if the scheduling task does. The scheduling task can be REPEATed AFTER a specified time, or EVERY <expr> clock ticks. Finally, a WHILE or UNTIL clause may be attached to control this rescheduling.)

```
- CANCEL <ident-list> ;
```

(Stop rescheduling of all the tasks in the list, but allow any currently executing tasks to finish.)

```
- TERMINATE <ident-list> ;
```

(Stop rescheduling of all tasks in the list, and terminate any tasks that are currently executing.)

```
          <arith expr> ;
- WAIT    UNTIL <arith-expr> ;
          FOR DEPENDENT    ;
              <event-expr>
```

(Stalls the current process for a certain number of clock ticks, UNTIL a specific time, until all DEPENDENTs have terminated, or until some event occurs.)

- UPDATE PRIORITY <ident> TO <arith expr> ;

  (Changes priority of a previously scheduled task.)

- SIGNAL <event var> ;

  RESET <event var> ;

  SET <event var> ;

  (Used to alter event variables and thereby schedule or control tasks. SIGNAL is used for unlatched events; when an event is SIGNALed all tasks WAITing on that event are placed in the ready state. SET and RESET are used for latched events. SET forces an event to the TRUE state and frees any tasks waiting for the TRUE value of the event, RESET forces the value back to FALSE and frees any tasks waiting for the FALSE value.)

- UPDATE; <stmt-list> CLOSE;

  (Update block for controlling access to shared variables by concurrent tasks. A variable declared with the LOCK attribute (LOCK (<lock number>)) may only be referenced in an update block, and a task executing an update block will be stalled until the locked variables in the update block are no longer being accessed in an UPDATE block of any other task.)

```
- ON ERROR { <error group> : <error number> }
             <error group>

                             SIGNAL
     { SYSTEM } {AND         SET     <event var> } ;
       IGNORE                RESET
```

ON ERROR { <error group> : <error number> } <stmt> ;
<error group>

OFF ERROR { <error group> : <error number> } ;
<error group>

(Similar to PL/I on-conditions. Each implementation will assign error groups and error numbers to the standard system errors (such as division by zero, illegal instruction); the user may use unassigned error groups and numbers for user defined conditions. The ON and OFF statements obey the HAL/S namescoping rules, so any modifications to the condition handling environment by an ON or OFF statement is removed on exit from the enclosing block.

- SEND ERROR <error group> : <error number> ;

(Simulates an occurrence of the specified error number.)

C. Data Structures

HAL/S has three constructs for creating more complex data structures from the basic data types:

(1) structures

The statement

STRUCTURE <template-name> { DENSE / ALIGNED } { RIGID } :

<level number> <ident> <attribute>,

.
.
.

<level number> <ident> <attribute> ;

declares <template-name> to be a structure template This template can be used in declaring a structured variable: DECLARE <ident> <template name> { (<arith expr>) }.

A structured variable can be dimensioned, and the components of a structure are referenced by the dot operator: <ident> . <component>

The assignment operator and the relational operators

=, ~= can be used to copy or compare compatible structures.

(2) arrays

A declaration of the form

DECLARE <ident> ARRAY (<dimension list>) <type specification> ;

declares <ident> to be an array of the specified type. Array elements, rows, or subarrays are accessed using the subscript operator <ident>$_{\text{<subscript list>}}$,

where a single subscript can be any of the following:

<#-of-elements> AT <start-pos>

(Selects a range of elements starting at the specified position.)

<arith-expr> TO <arith-expr>

(Selects a range of elements.)

<arith-expr>

(Selects a single element.)

*

(Selects all elements in the corresponding dimension.)

The assignment operator and the relational operators =, ~= can be used to copy or compare compatible arrays.

(3) pointer variables

HAL/S has fully typed pointer variables declared by statements of the form:

DECLARE <ident> NAME <type specification> ;

When a pointer of type X is used in an expression or on the left hand side of an assignment statement an automatic dereferencing takes place. For example, if P points to a variable of type INTEGER then the statement P = P+1; will increment the integer variable (the value of the pointer P is not altered).

A pseudo variable NAME is used to take the address of an object or to assign a value to a pointer variable:

```
NAME(<pointer var>) = NAME(<non-pointer var>);
NAME(<pointer var>) = NAME(<pointer var>)
```

In the first case the pointer variable is assigned the address of the non-pointer variable, in the second case the pointer variable is simply assigned the value of the pointer variable on the right hand side. Note that this implies that a pointer may not point to another pointer.

Pointer variables may be compared with the relational operators = and ~ =. Finally, if a pointer points to a structure then the dot operator may be used to access the components of the structure, and if a pointer points to a dimensioned object (ARRAY, MATRIX, or VECTOR) then subscripting may be applied.

D. Other Features

HAL/S is block structured language with reserved words, and comments in /* */ pairs. A simple replacement and a parameterized macro facility is provided by the REPLACE statement. The language also provides "inline functions"; function bodies as part of expressions. For example,

```
STRUCTURE X:                         /* Define a record   */
  1 A SCALAR,                        /* structure X.      */
  1 B INTEGER,
  1 C NAME X-STRUCTURE;              /* Now use it to     */
DECLARE XSTRUC X-STRUCTURE;          /* declare XSTRUC.   */
XSTRUC = FUNCTION X-STRUCTURE;       /* Initialize XSTRUC */
         DECLARE Y X-STRUCTURE;      /* using an inline   */
         Y.A = 0;                    /* function that     */
         Y.B = 0.0;                  /* returns an object */
         NAME(Y.C) = NULL;           /* of type           */
         RETURN Y;                   /* X-STRUCTURE.      */
         CLOSE;
```

The inline function is most powerful when combined with the macro facility (for example, the inline function in the previous example could be declared to be a macro called INIT. A statement of the form XSTRUC = INIT; would then initialize the variable XSTRUC.)

The language has a data declaration facility called COMPOOL that is somewhat similar to the Fortran COMMON statement:

```
<label>: COMPOOL { RIGID };
            <data declarations>
            CLOSE <label>;
```

COMPOOL blocks can be compiled independently from other programs, and the declarations in the COMPOOL block can then be included into a program by invoking the name of the COMPOOL block. The RIGID attribute forces allocation of the data in the order specified within the COMPOOL block.

HAL/S also provides for initialization of variables in DECLARE statements, and a CONSTANT attribute for declaring program constants. The language does not allow dynamic arrays, matrices, or vectors, but '*' bounds (as in PL/I) are allowed for formal parameters. Finally, HAL/S produces a standard output listing for all programs (programs are "prettyprinted" to show statement nesting, and subscripts or superscripts are printed on separate lines).

E. Runtime Environment

HAL/S requires a run-time stack, I/O routines, and scheduling routines for activating, suspending, and synchronizing tasks.

F. Syntax

The BNF grammar for HAL/S has approximately 450 productions.

2.6.2. CHARACTERISTICS

A. Machine Dependence

Except for the SUBBIT operator for extracting bits from an object, HAL/S is not machine dependent.

B. Efficiency

HAL/S is an efficient language. The language does not provide dynamic allocation of structures (as the PL/I ALLOCATE statement) or dynamic arrays, forbids branching out of procedure bodies, and has no BEGIN blocks. The high level operators and statements (matrix multiply, the UPDATE block, the SCHEDULE statement) should provide room for a great deal of optimization.

In a test performed by Intermetrics as part of the HAL/S acceptance tests [MAR75], the HAL/S compiler for the IBM 360 series generated code that was faster and required less core than IBM Fortran H (OPT=2). The benchmark included numerical analysis programs and bit and character processing programs.

C. Level of the Language

HAL/S is a high level language.

D. Size of Language and Compiler

HAL/S is a large language (comparable in size to PL/I), and the compiler is written in XPL. The compiler is probably large.

E. Special System Features

HAL/S has many features that would be useful in systems programming. The language allows DO-loop variables to be declared as TEMPORARY variables within the loop body. Variables declared to be TEMPORARY will be allocated in the fastest storage locations available.

example.

```
DO FOR TEMPORARY INTEGER I = 1,100;
   .
   .
   .
END;
```

A variable declared to be a TEMPORARY loop variable can not be accessed outside the loop body.

To allow for special extensions (possibly machine dependent) to HAL/S, a type of procedure or function called the %-macro was added to the language. %-macros may be implemented by inline substitution of the procedure body or by standard procedure call. As an example, the %-macro %NAMECOPY(A,B) will assign the pointer variable B to the pointer variable A without requiring type checking (thereby allowing any structure to be overlayed on any other structure).

HAL/S also has the RIGID attribute for COMPOOL or STRUCTUREs, the STRUCTURE and NAME types, the SUBBIT operator, the exception handling statements (ON, OFF, SEND ERROR), the UPDATE block for shared variables, and the extensive real-time processing statements (SCHEDULE, WAIT, CANCEL, TERMINATE). All of these features would be very helpful in systems programming.

F. <u>Error Checking and Debugging</u>

HAL/S is fully typed, so many compile time checks can be performed. The ON and OFF ERROR statements would be useful in monitoring runtime errors.

The language manual does not indicate that any special debugging aids are available.

G. <u>Design Support</u>

(a) modularity

HAL/S is quite modular. The COMPOOL block would be very useful in insuring that separately compiled programs use the same data structures. The LOCK and ACCESS attributes for program variables permit controlled sharing of program variables. Finally, HAL/S programs, procedures, functions, or COMPOOL blocks can be compiled independently (the first three generating object modules, the fourth generating an entry in the library of COMPOOL blocks for the installation).

(b) modifiability

The language has a number of features that would make HAL/S programs easy to modify. The REPLACE statement provides simple and parameterized macro replacement, the CONSTANT attribute can be used to declare program constants, and the COMPOOL feature allows a programmer to make minor changes to a data structure used by all programs in a project simply by changing a single COMPOOL block. Finally, the high level operators and structured programming constructs would also make program modification easier.

(c) reliability

According to its implementors, HAL/S was designed to improve software reliability. The language allows full type checking to be performed at compile time, and provides many structured programming constructs. The LOCK attribute in conjunction with the UPDATE block permits reliable data sharing, and the SCHEDULE, WAIT, CANCEL and TERMINATE statements provide high level features for real-time processing. The formatted output listings would also enhance reliability.

H. Use

HAL/S has been implemented on the IBM 360 series, the Data General NOVA, and the Shuttle flight computer (IBM AP-101). The compiler is written in XPL, so it shouldn't be terribly difficult to transport HAL/S to other machines. The language was designed and implemented by Intermetrics, and has been used extensively by NASA in the Space Shuttle program.

2.7. INTERDATA FORTRAN V

2.7.1. LANGUAGE FEATURES

INTERDATA FORTRAN V [INTE74a,INTE74b,INTE74c] is an extension of ANSI Standard Fortran, the major extensions being the ADDRESS (pointer) type and the ENCODE and DECODE statements for memory to memory data transfers under format control. The Fortran language, which was originally designed in the late 1950's, was the first algorithmic language to achieve widespread acceptance. The language has been used extensively for scientific programming, but the limited number of data types and control structures has hindered the use of Fortran for system-oriented problems. Two Fortran preprocessors (FLECS and PREST4) which allow the programmer to use structures programming control structures have also been included in this report.

A. Basic Data Types and Operators

FORTRAN V supports the five basic data types of ANSI Fortran (INTEGER, REAL, DOUBLE PRECISION, COMPLEX, and LOGICAL) as well as the pointer type ADDRESS. The language has no character or string data type, so alphanumeric data must be packed into INTEGER variables. Fortran V allows mixed mode expressions and will automatically convert between INTEGER, REAL, and DOUBLE PRECISION values. Character and address constants can be used in INTEGER expressions.

FORTRAN V allows the following types of constants to be used in expressions: integer, floating point, double precision floating point, complex, logical, data or statement addresses, character, and hexadecimal. The operators and the data types on which they operate are listed below:

arithmetic operators (INTEGER, REAL, DOUBLE PRECISION, COMPLEX, and ADDRESS operands)

+, -, *, /, ** - Standard arithmetic operators. ADDRESS type can only be used in INTEGER expressions. FORTRAN V also has a extensive library of mathematical functions.

relational operators

.EQ., .NE., .LT., .GT., .LE., .GE.

logical operators (LOGICAL operands only)

.NOT., .AND., .OR.

pointer operators and functions

A'<name>' - Yields the address of the object <name>, where <name> can be a simple variable name, array, array element, or a statement label.

IVAL(<address expr>)

FVAL(<address-expr>)

DVAL(<address-expr>)

- Functions for obtaining the INTEGER, REAL, or DOUBLE PRECISION value pointed to by the address expression. It is the user's responsibility to insure that the address expression is pointing to meaningful data. Note: there is no way to alter the value of the object pointed to by the address expression.

B. <u>Control Structures</u>

- IF (<logical-expr>) <stmt>

  (Simple conditional statement with no provision for an ELSE part.)

- IF (<arith-expr>) <label-1> , <label-2> , <label-3>

  (Three-way arithmetic if statement. A transfer is made to label-1, label-2, or label-3 depending on whether the arithmetic expression is negative, zero, or positive.)

- DO <stmt-no> <var> = <var-1>, <var-2>, <var-3>

```
        <stmt-list>
    <stmt-no> CONTINUE
        (For loop.  The variables var-1, var-2, var-3 must be
        INTEGER variables, and their values must be greater
        than 0.)

- GOTO <stmt-no>
  GOTO <assign-var>
  GOTO (<stmt-no-1>, ..., <stmt-no-k>), <var>
        (Unconditional, ASSIGNED, and computed goto
        statements.)

- <type> FUNCTION <func-name> (<parameter-list>)
        <stmt-list>
  END

  SUBROUTINE <subr-name> { (<parameter-list>) }
        <stmt-list>
  END
        (Standard function and subroutine definition.
        Neither can be recursive.  Both functions and
        subroutines can have multiple entry points.)

- RETURN
        (Return from a function or subroutine.)

- CALL <subr-name> { (<argument-list>) }
  <func-name> (<argument-list>)
        (Invoke a subroutine or function.)
```

C. <u>Data Structures</u>

FORTRAN V has only one feature for building more complex data types: arrays of up to three dimensions. The declaration

<type> <ident> (<dimension-list>)

declares <ident> to be an array of the specified type. The type can be any of the basic types, and array elements are extracted using the subscript notation <ident> (<subscript-list>) .

D. <u>Other Features</u>

INTERDATA FORTRAN V has an extensive library of built-in functions and subroutines including

BCLR - Bit clear.
BCMPL - Bit complement.
BSET - Bit set.
BTEST - Bit test.
ICBYTE - Byte clear.
ILBYTE - Byte load.
INBYTE - Byte complement.
ISBYTE - Byte store.
IAND - Bitwise AND, OR, exclusive OR, complement,
IOR and shift.
IEOR
NOT
ISHFT

FORTRAN V does not require that scalar variables be declared. A variable that is not explicitly declared is assumed to be INTEGER or REAL, the choice depending on the first character in the variable name.

FORTRAN V has formatted and unformatted sequential and direct access I/O facilities. In addition, the ENCODE and DECODE statements provide a means of transferring data for one memory buffer to another, the data being translated according to format control. The ENCODE and DECODE statements can be used for converting between character data and the six basic types.

Finally, FORTRAN V has a conditional compilation feature. Any statements with an X in card column 1 will be treated as comments unless the compiler debug option is on, in which case they are compiled as ordinary statements. The conditional compilation feature is very helpful for inserting debugging statements into a program.

E. Runtime Environment

FORTRAN V requires no runtime stack or dynamic storage allocator. However, the language does have fairly complex I/O facilities, so FORTRAN will require a number of I/O routines.

Still, the runtime environment for FORTRAN will be considerably simpler than the runtime environment for HAL/S, SPL, or JOVIAL.

F. Syntax

FORTRAN V probably has a BNF syntax, but compilers would not use it. FORTRAN statements are easy to parse, and most FORTRAN compilers use ad hoc parsing techniques.

2.7.2. CHARACTERISTICS

A. Machine Dependence

FORTRAN is as machine dependent as any of the other widely used programming languages. Almost all commercial computer systems provide a FORTRAN compiler, and FORTRAN programs can usually be transported to other facilities with out a great deal of effort. Note: one of the sources of difficulty in transporting FORTRAN programs is the difference in word sizes between the two machines. Since FORTRAN has no character or string data type, programs using character data must pack characters into INTEGER variables. Unless the packing density is set at one character per word (very expensive if there is much character data), the resulting programs will not be transportable to other machines without modification.

B. Efficiency

Optimized FORTRAN programs compare favorably with assembly language programs. The only operation in FORTRAN that is inefficient is formatted I/O, which must be interpreted at runtime.

C. Level of the Language

Fortran is a medium level language.

D. Size of the Language and Compiler

The FORTRAN language is moderate in size, and the compiler should be too.

E. Special System Features

FORTRAN V has a very limited form of pointer variables, and many logical (bit and byte) functions. The EQUIVALENCE and COMMON statements can be used to access a block of core under various formats.

F. Error Checking and Debugging

FORTRAN compilers have traditionally had poor compile and run time diagnostics. The lack of a character data type requires the compiler to accept character strings as part of INTEGER expressions - no type checking can be performed for characters. The pointer type ADDRESS can be used to point at any data item or statement in a FORTRAN program, and no type checking can be performed. It is therefore the user's responsibility to insure that pointers are used in a proper manner.

The INTERDATA implementation of FORTRAN V provides the following debugging features:

- $COMP - Turns on conditional compilation of source statements with an X in column 1.
- $TRCE - Turns on trace of all or selected program variables.
- $TEST - Turns on checking of array subscripts and DO-Loop indices for 0 or negative values.

G. Design Support

(a) modularity

FORTRAN V supports independent compilation of subroutines and functions. Data sharing is provided by the COMMON and EXTERNAL statements. FORTRAN is seriously lacking in structured control structures, however.

(b) modifiability

FORTRAN V has no macro processor, no CONSTANT statement for defining program constants, no INCLUDE feature for including source files into a program, and no data structures other than arrays. FORTRAN programs are often hard to read because of the lack of control structures. FORTRAN programs would be considerably harder to modify than programs written in PASCAL, for example.

(c) reliability

FORTRAN V can not perform any compile-time type checking of subroutine or function parameters, or check that variables declared in one COMMON block are consistent with variables declared in the same COMMON block by another function or subroutine. The ADDRESS type in FORTRAN V requires careful programming. It is the user's responsibility to insure that pointers are pointing to objects of the correct type. Also, the lack of control structures means that IF and GOTO statements must be use to simulate if-then-else statements, while and until loops, and case statements. This can greatly obscure the structure of a program. Finally, FORTRAN has no bit or character data types, requiring any program that uses these data types to pack characters or bits into words.

H. Use

FORTRAN V is implemented on the INTERDATA series of minicomputers. The FORTRAN language has been implemented on almost all commercial computer systems (although the implementations are all slightly different), and in the past few years a number of preprocessors have been written that permit the use of structured programming control structures in FORTRAN programs. The languages FLECS and PREST4 discussed in this chapter are two examples of this type of preprocessor.

2.8. JOSSLE

2.8.1. LANGUAGE FEATURES

JOSSLE [JOH73,PRE73] is a high level language developed by John White and Leon Presser at the University of California. Although it was designed to be used in implementing compilers, the language is general purpose (JOSSLE is loosely based on PL/I) and could be applied to most system-oriented problems. JOSSLE provides some special features for managing shared data in programs, and a hierarchical control structure that tends to force top-down development of programs.

A. Basic Data Types and Operators

JOSSLE has four basic data types: INTEGER, REAL, CHAR (character string), and BIT (bit string). Complete type checking is performed at compile time, and no automatic type conversion is peformed between the basic types. However, the language does provide a function CONVERT for requesting explicit data conversions.

The operators and the data types on which they operate on are listed below:

logical operators (BIT operands)

```
    ~ <expr>            - Bitwise complement, AND, and OR.
    <expr> & <expr>
    <expr> ! <expr>
```

relational operators (all basic types)

```
    =, ~=               - Operands can be INTEGER, REAL, CHAR,
                          BIT.  Both operands must have same
                          type.
    <, >, <=, >=        - Operands can be INTEGER, REAL, or BIT.
                          Both operands must have the same type.
                          Note that there is no implicit ordering
                          of the character set.
```

arithmetic operators (INTEGER and REAL operands)

+,-,*,/ - Operands can be INTEGER or REAL, but both must have same type.

MOD - Modulo operator. Operand must be INTEGER.

character operators and functions (CHAR operands only)

!! - Concatenation.

SUBSTR - Substring function. SUBSTR is not a pseudo-variable in the PL/I sense - it can not be used on the left-hand side of an assignment statement.

B. Control Structures

- BEGIN <stmt-list> END;

  (Compound statement.)

- IF <bit-expr> THEN <stmt> { ELSE <stmt> } ;

  (Standard conditional statement.)

- LOOP <stmt-list> END LOOP;

  (Unbounded repetition of the <stmt-list>. Each LOOP statement must contain an EXIT statement to provide termination of the loop.)

- CASE <integer-expr> OF

```
  <stmt-1>;
     .
     .
     .
  <stmt-k>;
END CASE;
```

  (Simple case statement. If the value of the expression is i then the i-th statement is executed. A runtime error message is produced if i is less than 1 or greater than k.)

- EXIT { IF <bit-expr> } ;

  (Unconditional and conditional exit of innermost LOOP statement.)

- RETURN;

  (Return from a procedure.)

- RETURN WITH <expr>;

  (Return from a function with a result.)

- CALL <proc-name> { (<parameter-list>) };

  <function-name> { (<parameter-list>) };

  (Invoke a procedure or function.)

- PROCEDURE <proc-name> { (<argument-list>) } ;

  <procedure-body>

  END PROCEDURE <proc-name>;

  PROCEDURE <func-name>

  { (<argument-list>) } RETURNS <type> ;

  <function-body>

  END PROCEDURE <func-name> ;

  (Standard procedures and functions. Neither can be recursive, and all parameters are passed by value.)

Note: JOSSLE has no GOTO statement.

C. Data Structures

JOSSLE has a number of constructs for creating more complex data structures from the basic types:

(1) one-dimensional arrays

The statement

<ident> LINLIST (<number-of-elements>) OF <type>;

declares <ident> to be a one-dimensional array. The type can be any one of the basic types or a record structure defined by the user. Array elements are accessed using the subscript operator <ident> (<subscript>) , and the assignment operator <- can be used to copy an entire array.

(2) record structures

The user can define record structures using the NEWTYPE statement:

<type-ident> = NEWTYPE

```
        <member-1> <type-1>;
            .          .
            .          .
            .          .
        <member-k> <type-k>;
    END NEWTYPE;
```

The <type-ident> can then be used anywhere that a basic type can be used. For example:

```
ERRORMSG = NEWTYPE           /* Define record structure */
   TEXT CHAR(20);            /* for an error message.   */
   ERROR-NO INTEGER;
   PRINT-FLAG BIT(1);
   NEXT-MSG PTR2A ERRORMSG;
END NEWTYPE;
DECLARE                      /* Now use the structure   */
   SIZE-ERROR ERRMSG;        /* to declare some things. */
   OTHER-ERRORS LINLIST(10) OF ERRMSG;
END DECLARE;
```

The syntax for referencing structure components is <structure-var> : <member-name> { . <member-name>} . The assignment operator <- can be used to copy an entire record from one variable to another.

(3) typed pointers

The declaration

```
<ident> PTR2A <type>;
```

declares <ident> to be a pointer to an object of type <type>. The type can be a user defined record structure. All pointer variables are initialized to the constant NULL. The following operators and JOSSLE statements are provided for manipulating pointer variables:

```
=, ~=,            - Equality and inequality.
<ptr-var> :>      - Object pointed to by the pointer
                    variable. Can appear on either side
                    of an assignment statement.
<ptr-var> :> <structure-member> { . <member> }
                  - Component in a structure pointed to by
```

the pointer variable.

ADDRESS(<variable>)

- Yields address of the variable.

CONTENTS(<ptr-var>)

- Yields value of object pointed to by the pointer variable. Can not appear on the left-hand side of an assignment statement.

ALLOCATE <type> SETTING <ptr-var>;

- Allocate statement that causes dynamic allocation of an object of type <type>, and the setting of <ptr-var> to the address of the new object.

FREE <type> PTD2BY <ptr-var>;

- Statement that deallocates the core block, pointed at by the pointer variable, and sets the pointer to NULL.

(4) stacks and queues

The JOSSLE declarations

<ident> STACK OF <type> ;

<ident> QUEUE OF <type> ;

are used to define stacks and queues. The type can be any basic type or a user defined record structure. Stacks and queues are initially empty, and objects can be pushed on or popped off a stack or queue with the following two operators:

<ptr-var> <== <stack-or-queue-var>

- Sets <ptr-var> to the address of the object in the stack or queue and then pops the object off the list. If the stack or queue is initially empty the pointer is set to NULL.

<stack-or-queue-var> <== <ptr-var>

- Pushes the object pointed to by the

pointer variable onto the stack or queue.

D. Other Features

JOSSLE provides several features for managing shared data and for structuring systems of programs. JOSSLE permits internal procedures (that is, nested procedures), but unlike other block structured languages an internal procedure does not automatically inherit all variables declared in outer blocks. An internal procedure can request the use of such variables using the KNOWN statement:

```
KNOWN
  <identifier-list>
END KNOWN.
```

This feature prevents internal procedures from modifying a variable declared at an outer level without gaining explicit permission to use it.

A system of JOSSLE programs is formed by creating a COMMUNICATION REGION specifying the member programs in the system and the data to be shared among the programs. The syntax for a COMMUNICATION REGION is as follows:

```
COMMUNICATION REGION <ident>
   <record-structure-definitions>
   <shared-variable-declarations>
   MEMBERS
      <main-program>
      <sub-program-list>
   END MEMBERS;
END COMMUNICATION REGION <ident>;
```

The statement defines <ident> to be a "task" composed of a main program and a list of subprograms which communicate only through the variables in the <shared-variable-list>. A MEMBER program can only be called by other programs in the same COMMUNICATION REGION. Each MEMBER program can be an independently compiled JOSSLE program or another COMMUNICATION REGION. A COMMUNICATION REGION is activated by a call to the identifier <ident>, which causes control to pass to the main program in the MEMBERS list.

JOSSLE has a CONSTANT declaration for declaring program constants, and primitive I/O facilities.

E. Runtime Environment

JOSSLE prohibits recursive procedures, so no runtime stack is required. However, JOSSLE does require a dynamic storage allocator and some form of garbage collector for compacting the dynamic storage area.

F. Syntax

JOSSLE has a BNF grammar with approximately 150 productions.

2.8.2. CHARACTERISTICS

A. Machine Dependence

JOSSLE has no machine dependent features and could be implemented on almost any machine.

B. Efficiency

JOSSLE has no recursion (and therefore no runtime stack), and the language does not permit dynamic arrays or varying length character or bit strings. Procedure parameters are all passed by value. These restrictions would tend to make JOSSLE efficient. However, JOSSLE programs that use pointer variables to dynamically allocate storage, or that perform a great deal string concatenation will require a dynamic storage allocator and garbage collector. Garbage collection can be very expensive.

C. Level of the Language

JOSSLE is a high level language.

D. Size of the Language and Compiler

JOSSLE is a fairly large language, and the compiler will also be large.

E. <u>Special System Features</u>

JOSSLE has record structures, bit and character strings, fully typed pointer variables, dynamic storage allocation, and the STACK and QUEUE data structures. All of these features would be helpful in systems programming.

F. <u>Error Checking and Debugging</u>

JOSSLE performs complete type checking at compile time and performs no automatic conversions between the data types. Default runtime checks include array subscript checking, CASE expression out bounds, data conversion errors from the CONVERT function, and substring length errors.

The JOSSLE manual does not indicate that any special debugging features are available.

G. <u>Design Support</u>

(a) modularity

Modularity in JOSSLE is excellent. The language provides the COMMUNICATION REGION concept, independent compilation of programs and COMMUNICATION REGIONS, and restricted inheritance of global variables. JOSSLE also has a small number of structured programming control structures.

(b) modifiability

JOSSLE programs should be very well structured because of the COMMUNICATION REGION concept and the declarations for controlling shared data. However, the language has no macro processor, and the set of control structures is fairly limited (no WHILE, FOR, or REPEAT UNTIL loops, and only a simple form of the CASE statement). Because of this, JOSSLE programs will be harder to modify than programs written in HAL/S or PASCAL.

(c) reliability

JOSSLE performs complete type checking at compile time. This permits a large number of errors to be detected at compile time that would go undetected in a language like Fortran. Like most languages with pointer variables, however, JOSSLE requires careful programming. There is nothing to prevent a user from using the ADDRESS function to point at a static variable, and then subsequently attempting to free that variable using the FREE statement.

H. Use

JOSSLE is implemented on the IBM 360 and 370 series. However, the language is machine independent and could be implemented on other machines.