LANGUAGE AND COMPUTER DESIGN FOR EFFECTIVE SOFTWARE SYSTEMS

by

ALAN W. LILLICH

B.S., Mathematics, Oregon State University, 1975
B.S., Computer Science, Oregon State University, 1975

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

THE FACULTY OF GRADUATE STUDIES

(Department of Computer Science)

We accept this thesis as conforming
to the required standard.

THE UNIVERSITY OF BRITISH COLUMBIA

January, 1979

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the Head of my Department or by his representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Computer Science

The University of British Columbia
2075 Wesbrook Place
Vancouver, Canada
V6T 1W5

Date Jan 11, 1979

Abstract

This thesis describes two distinct, but mutually supportive, research projects. The first is the design and implementation of a high level language intended to be suitable for writing operating systems among other large software products. It provides facilities for the creation and control of asynchronous processes along with powerful data and "sequential" control structures. The second project is the design and implementation of a machine architecture which is a congenial host for modern block structured languages. This machine has several advantages compared to most of todays computers: code generation is simple, the object code is very compact and the machine is reasonably fast.

Effective software systems are well designed, reliable, have "low" space-time products and are developed, maintained and used with a minimum amount of human effort. The work presented here is intended to be a viable first step towards the production of an environment for the production of effective software systems.

Table of Contents

Acknowledgements

I would like to thank:

My wife Pat, for her patience and support.

My advisor Bary Pollack, for his help and guidance.

Dave Brent, for the excellent work done on the compiler.

The department, for providing a salary for Dave.

My office mates, for their good company.

My bicycle, for teaching me the value of patience.

A. Introduction to WISCH

High level languages have firmly established themselves as the only sensible tool for so-called applications programming tasks, but with a few notable exceptions, the realm of "systems" programming has remained within the tight grasp of assembly language hackers. The combined effects of the inadequacy of commonly available languages, the need to "control the machine", the need for space/time efficiency and the terribly backward state of most contemporary computer architectures have made this area very resistant to the introduction of high-level languages.

WISCH provides one approach to language design for systems programming. It is a member of the growing family of PASCAL based languages, bearing closest resemblance to PLATON [SOR 75] and MODULA [WIR 76]. It provides powerful data and control structures for traditional sequential tasks along with mechanisms for the creation and control of groups of concurrent processes.[1] The philosophy of WISCH is that the programmer should concentrate on the high level structure of a system, and not on the details of how that structure is to be implememted. Thus for example, concurrent processing facilities are provided

---

[1]The word "sequential" here refers to the well-defined flow of control found in most languages. Programs written in FORTRAN, COBOL, PASCAL, etc. are all sequential programs. A "concurrent" program is a single program embodying two or more asynchronous processes. The moment-to-moment locus of control in a concurrent program may depend not only on the source program and data but also on external objects such as peripheral devices, schedulers, etc.

as language primitives with minimal specification of their implementation. In effect, the language provides a small operating system kernel, similiar to those constructed in more traditional manners (eg., see [KAH 78]). WISCH is also a strongly typed language, attempting to make maximal compile time checks on the validity of programs and using run-time checking where necessary.

The language should be useful for constructing concurrent systems where the data structures and algorithms used are more important than the low level details of how the individual tasks are implemented and controlled. Obvious areas of application include operating systems, terminal concentrators, laboratory instrument monitors, etc. With its emphasis on the high level aspects of inter-task communications rather than the low level details of a particular implementation, the language should also be useful in areas where "real" parallelism would be very useful but is not often found, such as simulation, graphics, multi-pass compilation, etc. (For a novel application of multi-programming see [DIJ 78].)

B. Excuses and rationalizations

Existing high-level languages have been used to write successful operating systems. A variant of ALGOL-60 has been used for some years by Burroughs for systems on their B5500 and later machines. MULTICS in PL/I and UNIX in C are probably the best known examples. Others include OS-6 in BCPL [STO 72] and SOLO in Concurrent Pascal [BRI 75]. EUCLID [LAM 77], MODULA, PLATON and SUE [ATW 72] also are interesting languages in which operating systems could (almost) be written. Given this body of existing work, why invent yet another version of the wheel?

In looking for a language to do systems programming on a minicomputer, several criteria were developed:

1. The language must provide powerful data and control structures. All large software systems are primarily collections of complex data structures and algorithms for their manipulation.

2. In balance with the above criterion, the language must not be excessively large.

3. The language must have built-in mechanisms for concurrent programming. (An arbitrary decision to research what could be done with such languages.)

4. The language must provide adequate flexibility to handle a wide range of "real" problems, particularly in the area of operating systems. It must be more than a toy.

All of the above languages fail to meet one or more of these criteria. BCPL and PLATON are very weak in the area of data structures (by the intent of their designers). Stoy and Strachey claim "The resources of the ideal software language should, in our opinion, be concentrated around control facilities, and matters concerning storage and representation left very much to the programmer." This is fundamentally opposed to the WISCH philosophy that the system programmer often needs powerful control <u>and</u> <u>data</u> structures. C is also weak on data structures, but to a much lesser extent. Concurrent Pascal is deplorably weak in procedural mechanisms. It not only forbids recursive routines and Algol-like multiple scope levels, it does not even allow nested routines. The data structures of PL/I are far too rich in some areas (eg. numeric precision) and too weak in others (eg. the lack of PASCAL-like subranges or enumerations). EUCLID and PL/I are both far too large to be pratical languages. Many features of PL/I are notoriously hard to implement, especially exceptional conditions. EUCLID is a new and still unimplemented language. A major goal of its design was to ensure that EUCLID programs could have their correctness mechanically verified. This has resulted in making the language rather complex and quite large. It also has several novel features that could be difficult to implement. Chief among these are the ability to parametrize type declarations and the module type (whose properties are quite different from the module of MODULA and WISCH). SUE is also a rather large and baroque language. Only Concurrent Pascal,

MODULA and PLATON have mechanisms for concurrent programming. (The multiprogramming facility of PL/I is archaic and unsuitable.) The "static" nature of COncurrent Pascal causes it to fall short on point 4. MODULA, never intended to be an operating system language, also fails to meet point 4.

Since no suitable language was found, the choice became one of designing a new language or adapting an existing one. To provide the necessary freedom in the design it was decided to create a new language from scratch rather than attempt to modify an existing one.

C. Overview

This section provides a brief introduction to the syntax and semantics of WISCH.

C.1. Programs and processes

At the outermost level, WISCH is a language of programs and processes. A process is a dynamic object. It is the unit of computational activity; that is, a processor executing some set of instructions and affecting some set of data objects. Each process executes its instructions in a purely sequential manner. However, different processes execute in "parallel" since they are essentially independant computations. All processes execute at a non-zero speed, but nothing may be assumed about the relative speeds among processes except that "high priority" processes are favored. (Process priority is described later under PROGRAM declarations and the CREATE statement. Facilities are provided for interprocess communication and synchronization. Processes have names and may be refered to by their creators.

Programs provide the static model of instructions and data objects from which processes may be created. Textually, programs look a lot like procedures. They have a heading with formal parameters, local declarations and a sequence of statements for a body. Programs may be nested, but unlike procedures, they have no knowledge of their outer environment.

A <u>WISCH</u> <u>system</u> consists of an outermost (level zero) program and all nested objects. That is, a WISCH system is a complete "operating system" written in WISCH. When a system begins execution an anonymous process is created (modeled after the level zero program). As this process executes, it may create other processes modeled after the programs immediately nested inside the level zero program. Because of the nested nature of programs, a WISCH system exhibits a tree structure of processes with the original process as the root.

These concepts are illustrated by the following (much simplified) example:

```
1   PROGRAM Example;
2
3      PROGRAM prog_faz(CONST more: BOOLEAN);
4
5         PROGRAM prog_bazfaz;
6               {declarations for prog_bazfaz};
7            BEGIN prog_bazfaz
8               {body of prog_bazfaz};
9            END prog_bazfaz;
10
11        VAR process_f1, process_f2: PROCESS;
12
13        BEGIN prog_faz
14           CREATE process_f1 FROM prog_bazfaz;
15           START process_f1;
16           IF more THEN
17              CREATE process_f2 FROM prog_bazfaz;
18              START process_f2;
19           FI;
20           {rest of prog_faz};
21        END prog_faz;
22
23     PROGRAM prog_foo;
24           {declarations for prog_foo};
25        BEGIN prog_foo
26           {body of prog_foo};
27        END prog_foo;
28
```

```
29      VAR process_a, process_b, process_c : PROCESS;
30
31      BEGIN Example
32         CREATE process_a FROM prog_faz(true);
33         START process_a;
34         CREATE process_b FROM prog_faz(false);
35         START process_b;
36         CREATE process_c FROM prog_foo;
37         START process_c;
38         {rest of the body of example};
39      END Example .
```

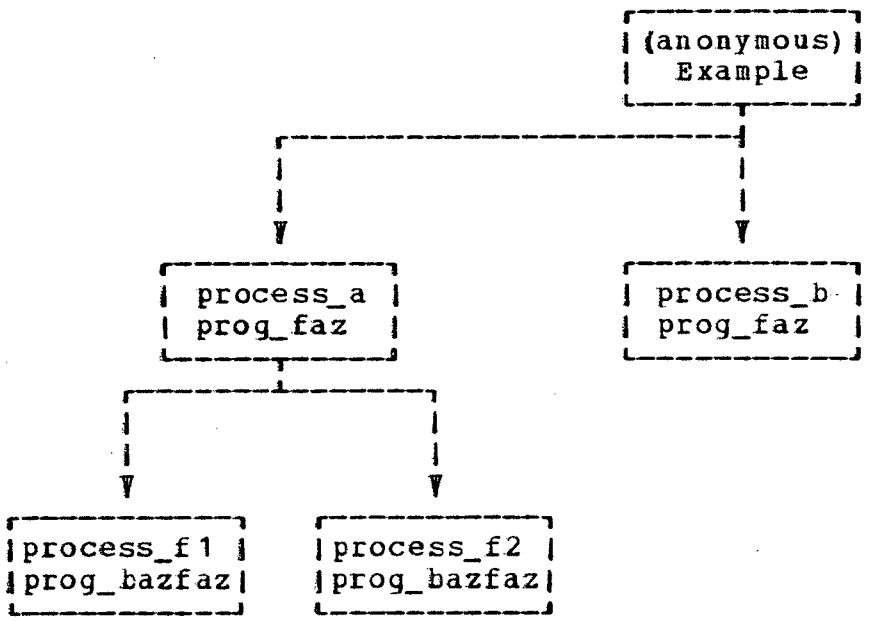
The nodes of the process tree are drawn:

```
┌──────────┐
│ process  │
│ program  │
└──────────┘
```

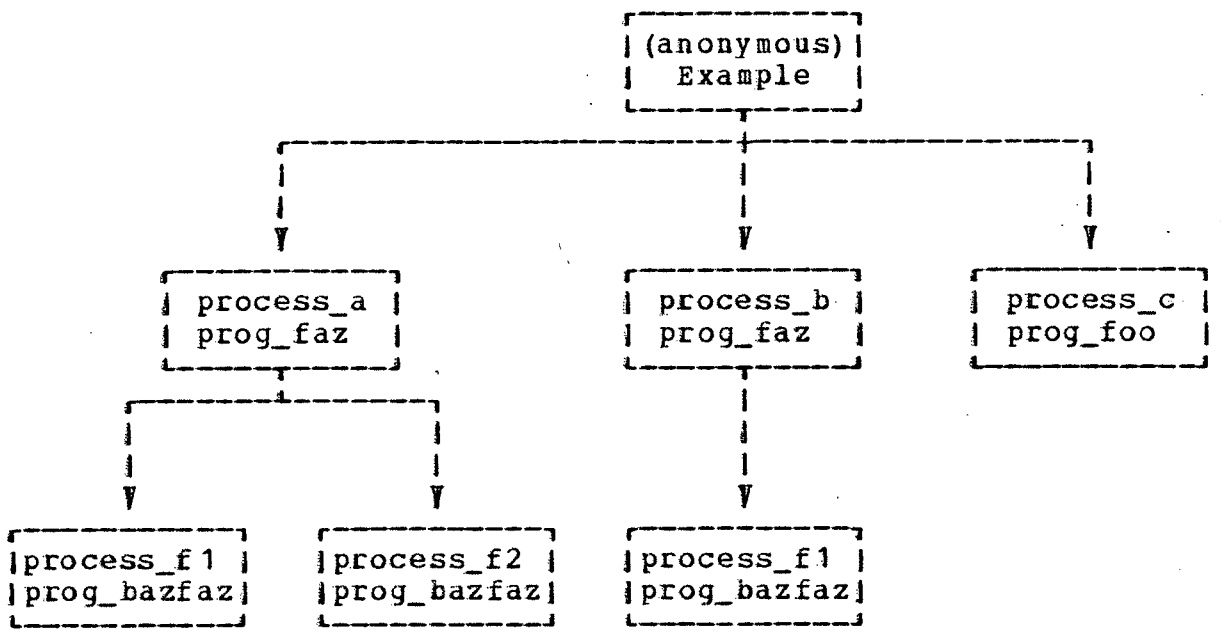
In this diagram, "process" is the name of the process variable and "program" is the name of the program that it is modelled after. Immediately before Example executes line 30, the tree is:

```
┌──────────┐
│(anonymous)│
│ Example  │
└──────────┘
```

Assuming that Example finishes line 33 just as process_a finishes line 19, but before process_b actually starts, the tree structure is:



Finally, at some later time the tree is:



Although there are two processes with names process_f1 and patterned after prog_bazfaz, they are <u>different</u> processes having

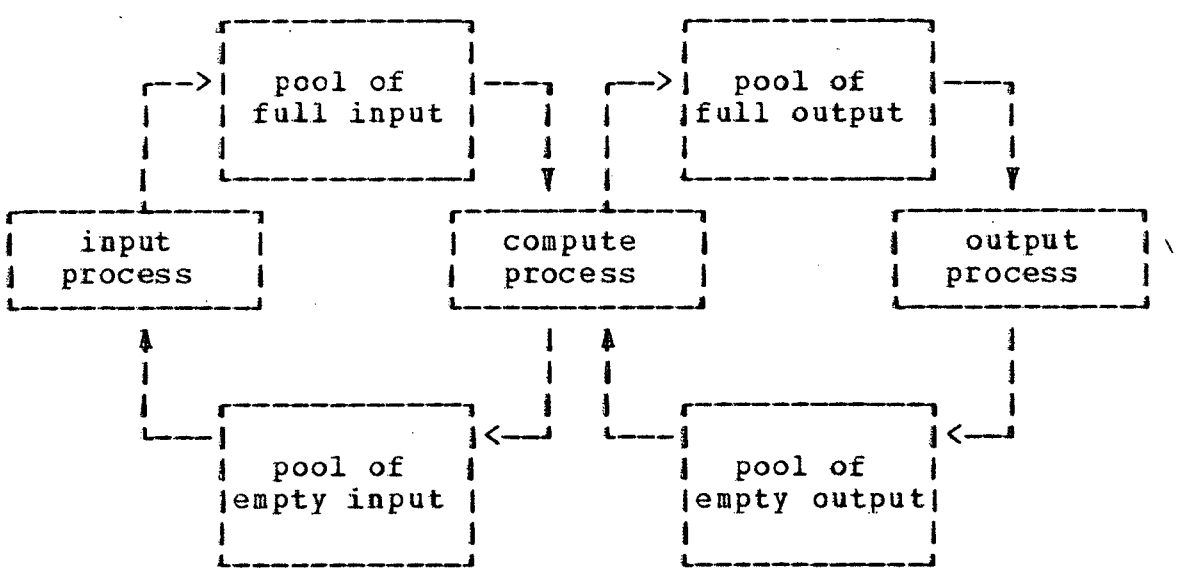
different parents. Since a program cannot declare two variables with the same name and can refer only to its immediate offspring processes by name, there is no confusion about which process is being refered to at any point. In other words, every process has a unique path name from the root process to itself. For example, the leftmost process on the bottom layer of the above diagram has a path name of (anonymous).process_a.process_faz1 and the righmost process on the bottom layer has a path name of (anonymous).process_b.process_faz1.

Communication and synchronization among processes uses a message passing facility modeled after that of PLATON. Programs may declare pools of shared objects, request objects from pools, and send objects to pools. Pools are actually linked lists of variables, all of the same type. If two different processes are passed the same pool as a parameter, they may communicate via it. (Note that the parameters syntactically belong to programs). When a process requests access to an object from a pool, it is given the first one if the pool is non-empty. If the pool is empty, the process is stopped and added to the end of a list of processes waiting for objects from that pool. When a process sends an object to a pool, if there are no processes waiting on that pool, the object sent is added to the end of the list. If there are processes waiting for objects from that pool, the first one is given the new object and allowed to continue. It is guaranteed that a shared object can be accessed from only one process at a time. Shared objects may be passed

by reference (processes are given pointers to them), allowing large blocks of data to be shared with little overhead.

For example, consider a very simple, dedicated "batch" system that reads input from one device, processes it, and writes output on another device. To achieve concurrency of I/O and processing the system is composed of three processes, an input process, a compute process and an output process. The computation process communicates with the input and output processes via pools of shared buffers.

Schematically, the system is:



Note that we really have two classical producer/consumer relationships here. The input process produces for the compute process to consume, and the compute process produces for the output process to consume.

A skeletal version of the code for this system follows. (For now, simply note that a PERVASIVE name is known everywhere and that the notation "@ POOLED type" denotes a pointer to a shared object of the specified type.)

```
PROGRAM batch;

   PERVASIVE CONST
      in_len = {an appropriate value};
      out_len = {an appropriate value};
      no_in_bufs = {an appropriate value};
      no_out_bufs = {an appropriate value};

   PERVASIVE TYPE
      in_buf = STRING (in_len);
      out_buf = STRING (out_len);

   VAR
      empty_in, full_in : POOL OF in_buf;
      empty_out, full_out : POOL OF out_buf;
      input, compute, output : PROCESS;

   PROGRAM input_prog (VAR empty, full : POOL OF in_buf);
      VAR buffer : @ POOLED in_buf;
      BEGIN input_prog
         FOR i := 1 TO no_in_bufs LOOP {allocate buffers}
            NEW(buffer);
            SEND buffer TO empty
         NEXT;
         LOOP {read input forever}
            AWAIT buffer FROM empty;
            {fill the buffer};
            SEND buffer TO full RESPOND empty;
         NEXT;
      END input_prog;

   PROGRAM compute_prog
            (VAR full_in : POOL OF in_buf;
             VAR empty_out, full_out : POOL OF out_buf);
      VAR
         inbuf : @ POOLED in_buf;
         outbuf : @ POOLED out_buf;
      BEGIN compute_prog
         FOR i := 1 TO no_out_bufs LOOP {allocate buffers}
            NEW(outbuf);
            SEND outbuf TO empty_out
         NEXT;
         LOOP {process input forever}
            AWAIT inbuf FROM full_in;
```

```
45              AWAIT outbuf FROM empty_out;
46              {produce the output from the input};
47              SEND inbuf TO SENDER;
48              SEND outbuf TO full_out RESPOND empty_out;
49           NEXT;
50        END compute_prog;
51
52     PROGRAM output_prog (VAR full : POOL OF out_buf) ;
53        VAR buffer : @ POOLED out_buf;
54     BEGIN output_prog
55        LOOP {write the output forever}
56           AWAIT buffer FROM full;
57           {output the buffer}
58           SEND buffer TO SENDER;
59        NEXT;
60     END output_prog;
61
62     BEGIN batch
63        CREATE input FROM input_prog(empty_in,full_in);
64        START input;
65        CREATE compute
66           FROM compute_prog(full_in, empty_out, full_out);
67        START compute;
68        CREATE output FROM output_prog(full_out);
69        START output;
70     END batch.
```

C.2. Statements

WISCH provides a pleasing, eclectic blend of the better control structures from such languages as PASCAL, ALGOL-68, EUCLID and PLATON, along with some recently published proposals. The structures provided facilitate the clear and easy expression of a wide variety of algorithmic concepts.

All "compound" statements (IF, FOR, etc.) have a fully delimited syntax with short, mnemonic keywords delimiting all major syntactic units. This "closed" syntax allows an arbitrary sequence of statements to be used any place that a single

statement can with no extraneous BEGIN-END pairs. The philosophy of WISCH is to be easy to read and reasonable to write. Verbosity and redundancy are used to improve readability and prevent simple errors from going undetected, while avoiding the COBOL-esque extreme of hindering the programmer.

C.3. Data types, literal denotations and expressions

In the tradition of PASCAL, the primitive data types, type declaration facilities and operators of WISCH provide the programmer with a powerful tool for the manipulation of problem oriented data. Elements are extracted from PASCAL, PLATON, MODULA, BCPL and PL/1. Included are almost all of the type concepts from PASCAL, and the shared variable and process structuring type concepts from PLATON. The primitive types include boolean, character, integer, real, storage (bitstring), character string and process. Among the type declaration concepts are enumerations, subranges, sets, pointers, arrays, records and pools (of shared variables).

The literal denotations for the standard primitive types are quite familiar. Of note is the ability to write structured literals of set, array and record types. Note that a literal is a denotation for a data value that represents itself, such as the integer value 100 or the string value "abc". Thus, the values of all literals (and literal expressions) are known at compile time. A literal expression is one involving only

literals, declared constants known at compile time, data attributes known at compile time and standard function references with literal expressions as arguments. The word <u>constant</u> is used in WISCH to denote a name for a value which cannot be altered by a program (as opposed to a variable). The value of a constant is often a literal value, but doesn't have to be. For example, a procedure might define a constant whose value is computed using a parameter of the procedure. The value of this constant is computed at every procedure entry, but once initialized, the constant cannot be altered.

Apart from the widely used relational, arithmetic and boolean operators, WISCH provides operations on bitstrings, sets and character strings. All expressions are "self-typing", that is no context external to an expression is needed to determine the result type. There are are almost no automatic type conversions associated with operators. Programmed conversions are available though, along with the values of some attributes for most data types. Literal expressions are guaranteed to be evaluated at compile time, with possible (but not guaranteed) effects on the compiled code. For example, the source line:

```
IF b THEN s1 ELSE s2 FI
```

might get compiled as simply:

```
s1
```

if b is a literal expression with the value TRUE.

C.4. Modules

WISCH has a modularization facility similiar to that of MODULA. This allows the programmer to build semipermeable walls around sections of declarations, explicitly controlling what portions of the outside world are known within the module and what objects declared inside the module are known outside of it. The module provides a means of hiding implementation details from portions of the code which have no need or business knowing them.

D. Primitive elements

This section describes WISCH at the lowest lexical level, that of the strings of characters composing all programs. WISCH is a free format language, allowing source text to be written anywhere on a source line and allowing blanks and/or comments to occur between any syntactic objects. However, as opposed to some languages, physical source lines are sometimes significant. Comments and string literals are not allowed to cross line boundaries. The identifiers after the BEGIN and END of program, procedure and function bodies must be on the same lines as the BEGIN and END. In the tradition of other ALGOL-like languages, semicolons are used to separate major syntactic constructs. WISCH allows an optional terminating semicolon for those who prefer the PL/1-like rule that constructs are terminated with semicolons (the examples given throughout this report are written in this manner). It is desirable, but not required, that an implementation allow a programmer to omit semicolons where possible.

D.1. Alphabet, identifiers and comments

```
Letters =   A B C D E F G H I J K L M
            N O P Q R S T U V W X Y Z
            a b c d e f g h i j k l m
            n o p q r s t u v w x y z

Digits  =   0 1 2 3 4 5 6 7 8 9

Others  =   . , ; : ' " _ ( ) { } < >
            = + - * / ¬ & | % @ ! $ #
```

The set of letters in WISCH consists of the 72 English upper and lower case letters. No distinction is made between the cases <u>anywhere</u> in the language except for character and string literals. For example:

and, And, AND

are three ways to spell the same word. The programmer can pretend that all letters (except those in character or string literals) are converted to upper case (or lower case) by a preprocessor and not go wrong. The set of digits is composed of the ten decimal digits. Identifiers are written as strings of letters, digits and underscores (_). The first character must be a letter and the last must not be an underscore. Identifiers may be of arbitrary length, but an implementation may place a limit on the number of significant characters in an identifier. This limit must not be less than fifteen however.

The set of "other" characters listed above includes all other characters used as part of the language. They are all found in the 128 character ASCII set. Of course, characters not used in the language but in the implementation set may be used in character and string literals. Suggested transliterations for some special symbols are given below in the section on special symbols.

Comments in WISCH extend from a left brace ({) to a right brace (}) or end of the source line (which ever comes first). A single comment may not cross a line boundary. For example:

```
IF a < b THEN a := b; {Select the larger of a and b.
```

or:

```
VAR (i,j):=0 {# used}, k:=-1 {next free} : INTEGER;
```

With this convention it is impossible to write "reversed" programs and it is much easier to avoid inadvertant inclusion of code in comments. Comments may be used anywhere in the source program that a "trivial" blank may. Comments in WISCH may also be nested, so that the following is a line that is completely commented out by the first left brace and the end of line:

```
{   dump_status(current_state); {for debugging only}
```

D.2. Reserved words

The following table lists the reserved words of WISCH:

ABS	ELSE	FORK	NEXT	SENDER
ARRAY	END	FORWARD	OF	SET
ASSERT	ENDBLOCK	FROM	OPTION	STACK
AWAIT	ENDFINAL	FUNCTION	OTHER	START
BEGIN	ENDINITIAL	HEAP	PERVASIVE	STOP
BIND	ENDMODULE	IF	POOL	STRING
BLOCK	ENDRECORD	IMPORT	POOLED	THEN
BY	ERROR	IN	PRIORITY	TIME
CASE	EXIT	INITIALLY	PROCEDURE	TYPE
CONST	EXPORT	ITERATE	PROGRAM	UNBIND
CREATE	EXTERNAL	JOIN	RECORD	UNLESS
DESTROY	FI	LOOP	REM	VAR
DIV	FIN	MOD	RESPOND	WHEN
DOING	FINALLY	MODULE	RETURN	XOR
ELIF	FOR	MYSELF	SEND	

These words may not be used as user defined identifiers. Again, although they are written in upper case throughout this report, they may legally be written using any combination of upper and lower case.

D.3. Special symbols

The special symbols used in WISCH can be divided for convenience in syntactic separators and operators. Some of the reserved words are actually operators, so they are listed again here.

The syntactic separators are:

.	,	;	:	..	!	$	@	#
'	"	(	)	{	}	[	]	=
:=	:=:	+:=	:+=	-:=	:-=	*:=	:*=	
/:=	:/=	%:=	:%=	&:=	:&=	|:=	:|=	
**:=	:**=	&&:=	:&&=	||:=	:||=	IN:=	¬IN:=	
DIV:=	:DIV=	REM:=	:REM=	MOD:=	:MOD=	XOR:=	:XOR=	
<:=	:<=	¬<:=	:¬<=	<=:=	:<==	¬<=:=	:¬<==	
>:=	:>=	¬>:=	:¬>=	>=:=	:>==	¬>=:=	:¬>==	
=:=	:==	¬=:=	:¬==					

The operators are:

+	-	*	/	DIV	REM	MOD	ABS
**	%	¬	&	|	&&	||	XOR
<	¬<	<=	¬<=	>=	¬>=	>	¬>
=	¬=	IN	¬IN				

Suggested transliterations for non-ASCII character sets include:

(* and *) for { and }
(. and .) for [and]

D.4. Standard identifiers

The following table lists the standard identifiers of WISCH.

ADDR	FALSE	MAX	SHIFT
ATAN	FIND_CHAR	MAX_POS_REAL	SIGN
BOOLEAN	FIRST	MIN	SIN
CARD	FIRST_ELEM	MIN_POS_REAL	SQRT
CEIL	FLOOR	NEW	STORAGE
CHAR	INCR	NIL	SUCC
COS	INTEGER	PAD_STRING	TAN
DECR	LAST	PRED	TRUE
DISPOSE	LAST_ELEM	PROCESS	TRUNC
EMPTY	LENGTH	REAL	UP_BND
EXP	LO_BND	ROTATE	VALUE_SIZE
	LOG	ROUND	WAITING
	LOG10		

These identifiers are treated as though they are declared in a scope surrounding the WISCH program, procedure or function being compiled. The standard definitions for each are given where they are introduced in the body of the report. Most of them are treated exactly as though they are ordinary, programmer defined names and may be redefined if desired. However, the standard type names (BOOLEAN, CHAR, INTEGER, PROCESS, REAL, and STORAGE), the boolean values (TRUE and FALSE) and the nil pointer value (NIL) are pervasive names and <u>may not</u> be redefined. Thus, to the programmer these names look like reserved words.

D.5. A note on the syntax graphs

Many sections of this report include syntax charts relevant to their topic. The "rules" of these charts are as follows:

1. Words written in all CAPITAL letters represent reserved words. They are in all capitals for identification only. WISCH allows them to be written as any combination of upper and lower case. Standard names are written in upper case also.

2. Items written in lower case and enclosed in angle brackets (<,>) are syntactic objects described with charts of their own.

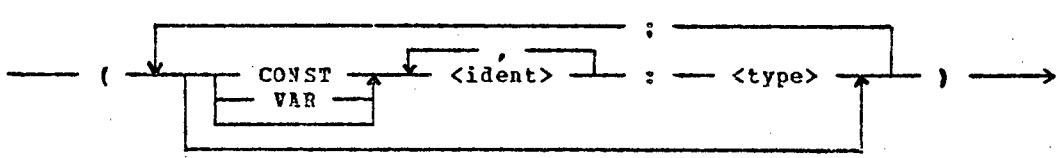
3. Sequences of special characters are literals representing themselves.

4. All charts have a single entry at the left and a single exit at the right.

5. Flow in the charts is basically counter-clockwise, that is, from left to right; or down, from left to right and back up; or up, from right to left and back down. Arrows on the charts help emphasize this.

As an example consider the chart for program, procedure and function formal parameters:

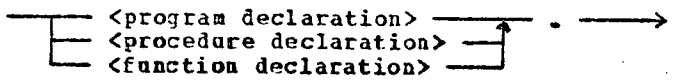
prog/proc/func formal params :



The parameters consist of a left parenthesis (), followed by one or more declarations separated by semicolons, followed by a right parenthesis ()). A declaration consists of the word CONST or the word VAR or neither, followed by one or more identifiers separated by commas, followed by a colon and a type. Identifier and type are defined by other charts.

E. Compilation unit

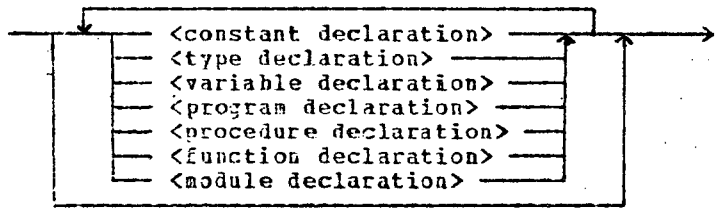
compilation unit :



The the unit of compilation in WISCH is a single program, procedure or function. A compilation defines a completely closed scope, so a separately compiled piece of code knows nothing about the environment it is eventually used in. That is to say, while an inner procedure of a program has access to all objects declared in outer scopes, a separately compiled procedure declared at the same level has no access to any of those outer objects. A separately compiled program, procedure or function is declared by specifying that its body is EXTERNAL. (See the sections below on PROGRAM, PROCEDURE and FUNCTION declarations.)

F. Declarations

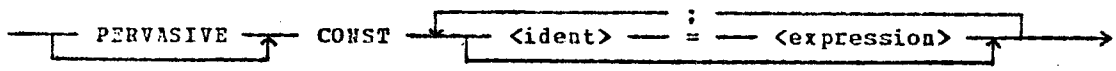
declarations :



WISCH has six forms of declarations: constant, type, variable, program, procedure and function. They may be mixed in any order, provided that all identifiers are declared before they are used. (A small exception is made for the object type of pointers.) It is also illegal to redefine a name in a scope once it has been used in that scope. (This is explained more fully below in the section on scope of names.)

F.1. Constants

constant declaration :



The CONST declaration allows identifiers to be equated with unalterable values. The expression may or may not be a literal expression, its type is the type that will be associated with the identifier. If the expression cannot be evaluated at compile time, the CONST declaration essentially declares a

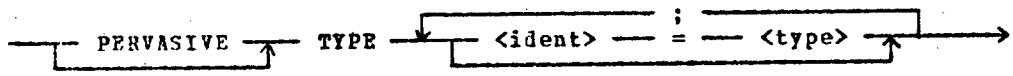
variable which is initialized at scope entry and cannot otherwise be assigned to. The PERVASIVE attribute is discussed in the section on the scope of names.

Some examples of constant declarations are:

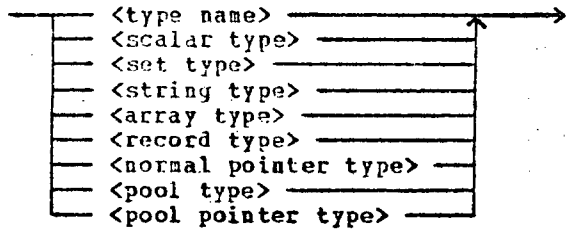
```
TYPE
   chars = SET OF CHAR;
   vec = ARRAY (1..5) OF INTEGER;

CONST
   size = 100;
   ibm = "IBM";
   hp = "HP";
   target = hp;
   digit_set = chars('0..'9);
   lower_case_set = chars('a..'z);
   alpha_set = lower_case_set + digits;
   codes = vec(0,2,4,6,8);
```

F.2. Types

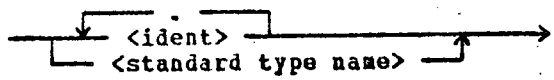
type declaration :



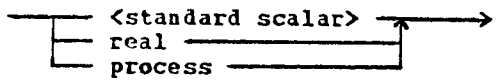
type :



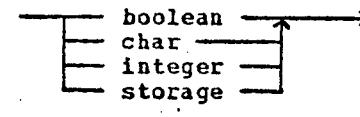
type name :



standard type name :



standard scalar :



This section describes data types in detail. The syntax and semantics presented here also apply to types used in variable, parameter and function declarations. WISCH is a strongly typed language, that is types must be known at compile time for all objects and checking of all operands is enforced. A large variety of types are provided, including six standard types and ten constructors.

A clear understanding of the data type nomenclature used throughout this report is essential to understand the following sections, so the following table is presented:

type	class				
	standard	scalar	assignable	shareable	structured
process	x				
real	x		x	x	
boolean	x	x	x	x	
character	x	x	x	x	
integer	x	x	x	x	
storage	x	x	x	x	
enumerated		x	x	x	
subrange		x	x	x	
string			x	x	
set			x	x	x
array			<1>	<2>	x
record			<1>	<2>	x
normal pointer			x		
pool pointer					
pool					
module					

<1> - Arrays and records are assignable iff all components are.
<2> - " " " " shareable " " " "

The standard types are the five built-in types. Scalar types are those with values that can be mapped 1 to 1 with some finite subset of the integers. Assignable types are those for which variables of that type may be assigned to. Shareable types are those for which objects of that type may be shared among different processes (pooled objects). Structured types are those for which "structured" values may be written.

F.2.1. Assignable types

The assignable types are those that may be involved in an assignment statement, that is a variable of that type may be assigned to.

F.2.1.1. Scalar types

The scalar types are the simplest data types. The values of any scalar type are finite (in number and magnitude) and are totally ordered. There exists a (machine-dependant) one-to-one mapping between the values of any scalar type and a subset of the integer values represented on the machine. All scalar types share the relational operators <, ¬<, <=, ¬<=, =, ¬=, >=, ¬>=, >, ¬>, which have the obvious meanings (less than, not less than, less than or equal to, etc.). The relational operators have priority 0, lower than all other operators.

There are six standard functions defined for all scalar types. The functions FIRST and LAST accept the name of a scalar type or variable, and return the lowest and highest values of the type respectively. The functions SUCC and PRED accept a scalar expression and return the next larger and next smaller values respectively. An attempt to take the SUCC of the largest value or the PRED of the smallest value is an error. The functions MIN and MAX accept two scalar expressions as arguments and return the value of the smaller and larger respectively.

Two standard procedures, INCR and DECR are provided to increment and decrement scalar variables (by "one"). They are also checked for range errors.

In addition, the name of any scalar type may be used as a generic transfer function to convert a value from any scalar type to the named one, with appropriate range checking. The conversion is defined by the implementation defined mapping between any scalar type and the integers. For example, on a machine using the 128 character ASCII set, CHAR(7) is the bell character, INTEGER('a) is 97 and CHAR(300) is illegal.

F.2.1.1.1. Boolean

The type boolean is denoted by the standard type name BOOLEAN. The values of boolean are the words TRUE and FALSE, with FALSE < TRUE. There are three operators which accept boolean arguments and return a boolean value:

operator	priority	operation
&	2	logical conjunction (and)
|	1	logical disjunction (or)
¬	1	unary logical negation (not)

The operators & and | are evaluated in a left-to-right "McCarthy" manner, that is a string of and'ed terms is not evaluated past the leftmost term that is FALSE, and a string of or'ed terms is not evaluated past the leftmost term that is TRUE. Because of this McCarthy evaluation, boolean operators

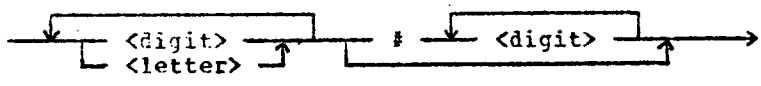
cannot be said to associate.

F.2.1.1.2. Character

The type character is denoted by the standard type name CHAR. The values of type character are an implementation defined character set. An implementation may (should) also define other specific character types, such as ASCII_64, ASCII_128 or EBCDIC. There are no operators on character values except the relational ones. Character literals are written as a single character preceded by an apostrophe ('), for example 'A. An apostrophe is written as two consecutive apostrophes, that is ''. Non-printable characters may not be denoted directly, but may be denoted using the CHAR transfer function mentioned above with a literal argument. There are no direct denotations for alternate character types, but they may be written using the built-in transfer functions mentioned above. For example, ASCII_64('A) would return the ASCII_64 representation representation of the character 'A.

F.2.1.1.3. Integer

integer literal : (no embedded blanks)



The type integer is denoted by the standard type name INTEGER. the values of type integer are an implementation defined subset of the integers. Negative integer values are represented internally in an implementation defined manner. Literals are written as a sequence of digits, optionally followed by an octothorpe (#) and an unsigned decimal integer specifing the number base. There can be no characters between the last digit of the number and the octothorpe or between the octothorpe and the first digit of the radix. If the base specifier is ommited, the number is assumed to be decimal. The allowable radix values must include at least 2, 8, 10 and 16. The digits of the number must be legal for the base used, for example, 459#8 is illegal. There are no predefined constants for the minimum and maximum integer values since they may be obtained via FIRST(INTEGER) and LAST(INTEGER).

There are ten operators which accept integer operands and return an integer result:

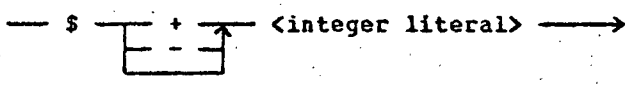
operator	priority	operation
**	3	integer exponentiation
*	2	multiplication
DIV	2	division (truncate towards zero)
REM	2	division remainder i REM j = i - (i/j)*j the sign of (i REM j) is always the same as the sign of i
MOD	2	mathematical modulus the sign is always positive for i>=0, i MOD j = i REM j else, i MOD j = (i REM j)+(ABS j)
+,-	1	addition and subtraction
+,-	1	unary plus and unary minus
ABS	1	unary absolute value

Integer exponentiation is defined in the obvious manner. That

is, a**b is defined as a*a* ... *a (i.e., b occurances of a) if b is positive, 1 if b is zero, and 1 DIV (a**-b) if b is negative. Association of all integer operators except ** is left-to-right, ** associates right-to-left. A standard function, SIGN, is provided to extract the numeric sign of an integer expression. It accepts a single argument and returns -1 if it is negative, 0 if it is zero and +1 if it is positive.

F.2.1.1.4. Storage

storage literal : (no embedded blanks)



The type storage is denoted by the standard type name STORAGE. The values of type storage are bit patterns. Literals are written as signed integer literals preceeded by a dollar sign ($), for example $64 and $100#8 both represent the bit pattern 1000000 (ignoring leading zeros). The range of integers allowed and their corresponding bit patterns is implementation defined.

There are four operators which accept storage arguments and return a storage result:

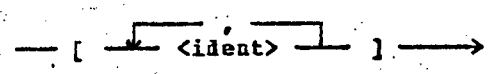
operator	priority	operation
&&	2	bitwise and
||	1	bitwise inclusive or
XOR	1	bitwise exclusive or
¬	1	unary bitwise complement

Note that although these operators look similiar to the boolean operators &, | and ¬, they are indeed very different. In particular, the storage operators do not work in a "McCarthy" manner. Association of all storage operators is left-to-right. The meaning of relational comparisons between storage values is implementation defined.

In addition, there are two functions which accept a storage first argument and an integer second argument. They are the shifting functions, SHIFT and ROTATE. SHIFT is an end-off, zero-fill shift and ROTATE is a circular rotation. The integer argument is the number of bits to shift, a positive value indicates a left shift and a negative value indicates a right shift. (Easy to remember if left shifts are thought of as being essentially multiplications by a power of two.)

F.2.1.1.5. Enumeration

enumeration :



Enumeration types are declared by giving a list (enumeration) of unique identifiers which are the values of the type. The values are ordered left to right, from lowest to highest. The value identifiers may not be redefined in any

inner scope that has access to the type. There are no operators for values of enumeration types except the relational ones. Some examples of enumeration types are:

```
TYPE
   color = [black,red,green,blue,white];
   disk_op = [read,write,seek,reset];
```

The type color has five values. literals of type color are denoted by the identifiers black, red, green, blue and white. The lowest value is black, the highest is white.

F.2.1.1.6. Subrange

subrange :

```
— <expression> — .. — <expression> ——→
```

Subrange types are based on other scalar types. They are declared by providing literal expressions of the basis type for the initial and final values of the subrange. The lower bound must not be larger than the upper bound. The values of a subrange are the values of the basis type between the values of the initial and final expressions, inclusive. Literal denotations and operators are the same as for the basis type. Some examples of subrange types are:

```
TYPE
   hue = red..blue;
   capital_letter = 'A..'Z;  {dangerous with EBCDIC!}
   table_index = 0..size-1;
```

Subrange types may also be defined for which the exact limits are not known at compile time. If either (or both) of the subrange bound expressions are not literal expressions, the subrange defined is a "dynamic" subrange. In this case, the limits of the range are not fixed until scope entry at run time. The range expressions are evaluated exactly once for each explicit usage of a dynamic subrange. For example, given the code segment:

```
TYPE
   range = 1 .. no_entries+1;

VAR
  old, new: ARRAY(range, range) OF CHAR;
  i1, i2, i3: 0 .. no_entries+1;
  j1, j2, j3: 0 .. no_entries+1;
```

the expression "no_entries+1" is evaluated three times, once in the declaration of the type "range", once in the declaration of the "i" variables and once for the "j" variables. It is <u>not</u> reevaluated when "range" is used in the array type specification. The dynamic bounds are reevaluated though whenever a variable is created with the NEW procedure (see pointer types for a description of NEW). For example, given the code:

```
VAR
   (lwb, upb):=0 : INTEGER;

TYPE
   index = lwb-1..upb+1;
   dyn_array = @ ARRAY (index) OF INTEGER;

VAR
   table1, table2: dyn_array;

BEGIN
   lwb := 0;    upb := 19;
   NEW(table1);
```

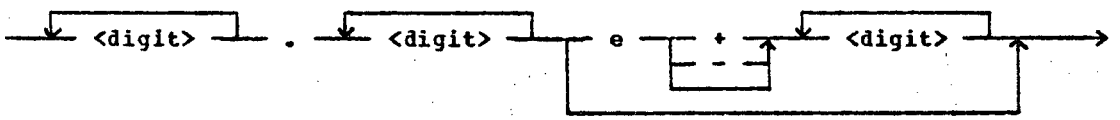
```
lwb := 1;   upb := 100;
NEW(table2);
```

the variable table1 will point to an array with 22 elements indexed from -1 to 20 and table2 will point to an array with 102 elements indexed from 0 to 101. The expressions "lwb-1" and "upb+1" are evaluated twice, once each for the creation of the arrays. Dynamic subranges thus allow the creation and manipulation of data structures whose size is not known til run time. They also allow procedures to accept variably sized arrays as parameters, for example:

```
FUNCTION mean (lwb, upb: INTEGER;
               values: ARRAY (lwb..upb) OF REAL): REAL;
```

F.2.1.2. Real

real literal : (no embedded blanks)



The type real is denoted by the standard type name REAL. The values of type real are an implementation defined subset of the rational numbers. Literals are written as a sequence of decimal digits, a point (.), another sequence of decimal digits and an optional exponent. Note that there must be at least one digit <u>on each side</u> of the point and that no embedded blanks are allowed. The exponent is written as the letter e (or E)

followed by an optional sign and a decimal integer literal, again with no embedded blanks. The range and precision of real values is implementation defined.

There are eight operators which accept real arguments and return a real result:

operator	priority	operation
**	3	exponentiation
*	2	multiplication
/	2	division
+,-	1	addition and subtraction
+,-	1	unary plus and unary minus
ABS	1	unary absolute value

Association of all real operators except ** is left-to-right, ** associates right-to-left. If the operators for exponentiation, multiplication, addition or subtraction have one real argument and one integer argument, they will yiels a real value. The operator for real division <u>always</u> yields a real value, either or both arguments will be converted if necessary. The relational operators defined above for scalar values are of course also defined for real values. Exponentiation is guaranteed to be implemented as repeated multiplications for some implementation defined range of integer exponents. The actual algorithm used is implementation defined, it is only guaranteed that logarithms and exponentials will not be used. For example:

```
x ** 9
```

might be implemented as:

```
x * x * x * x * x * x * x * x * x
```

or as:

$$x * ((x^2)^2)^2$$

that is, with 4 multiplications instead of 8. Note that exponentiation by a real value is indeed allowed. The point made above is that exponentiation by certain integer values may be faster and more accurate than exponentiation by equivalent real values.

There are several standard functions for conversion between integers and reals and for mathematical computation. The type name REAL may be used as a function to convert integer values to real values (but is actually superfluous since this will be done automatically). There are four functions to convert real values to integers, TRUNC, ROUND, FLOOR and CEIL. TRUNC gives a truncation towards zero. ROUND gives a rounding away from zero. FLOOR gives the largest integer that is less than or equal to the real value. CEIL gives the smallest integer that is greater than or equal to the real value. The mathematical functions are SQRT, EXP, LOG (natural), LOG10 (common), SIN, COS, TAN and ATAN. The standard functions MIN and MAX defined above for scalar values also accept real arguments, as does the function SIGN defined above for integer values. All three return real values when given real agruments. There are two standard constants of type REAL, MAX_POS_REAL and MIN_POS_REAL. MAX_POS_REAL is the largest real value that may be represented and MIN_POS_REAL is the smallest positive (non-zero) value that may be represented.

F.2.1.3. Powerset

```
set type :
   ── SET ── OF ── <scalar type> ──►
```

Powerset types (hereafter simply called set types) are defined based on some scalar type (other than a dynamic subrange). The values of a set type are elements of the powerset of the scalar basis type. For example the type "SET OF hue" has eight values (with hue as defined above in the subrange type examples). They are the sets (red), (green), (blue), (red,green), (red,blue), (green,blue), (red,green,blue) and () (the empty set).

Values of a set type are written as the type name followed by a parenthesized list of expressions in the basis type (the elements). If all elements are literal expressions, then the denotation is a set literal. For example, given the declaration:

```
TYPE
   hues = SET OF hue;
```

one could write:

```
hues(red,green)
```

A "subrange" notation may also be used in set values. An element of the form a..b denotes all values from a to b inclusive, where a and b are arbitrary expressions of the basis type. For example:

```
chars('+,'0..'5,'-)
```

is equivalent to:

```
chars('+,'0,'1,'2,'3,'4,'5,'-)
```

If the lower bound is larger than the upper bound, the "null range" is used. For example:

```
ints(0,1,7..3,10)
```

is equivalent to:

```
ints(0,1,10)
```

A particular implementation may place restrictions on the allowable cardinality of the basis type and on the allowable "magnitude" of the basis values, but must at least allow for the declaration of a SET OF CHAR.

There are four operators which accept set arguments and return a set value:

operator	priority	operation
*	3	set intersection
+	2	set union
-	1	set difference in terms of bitwise logic, a - b = a XOR (a && b)
-	1	unary set complement

Association of all set operators is left-to-right. The relational operators defined above for scalar values may also be used with set values. They test the subset relation, for example, set_a < set_b is TRUE if the value of set_a is a proper subset of the value of set_b. There are also two operators which accept a scalar left argument, a set right argument and return a boolean value. They are the set element operators, IN and ¬IN. IN returns TRUE if the left argument is an element of the right, FALSE otherwise, ¬IN does just the opposite. Note that if the value of the left operand is not within the set basis type IN returns FALSE (¬IN returns TRUE), agreeing with normal intuition. Like the other relational operators, these have a priority of 0.

For example, given the following code segment:

```
TYPE
   chars = SET OF char;
   nums = SET OF 0..10;

VAR
   char_digits : chars;
   num_digits : nums;

BEGIN
   char_digits := chars('0..'9);
   num_digits := nums(1,3,5,7,9);
```

the following expressions are TRUE:

```
'3  IN char_digits
10 ¬IN num_digits
```

and the following are FALSE:

```
'A  IN char_digits
 4  IN num_digits
500 IN num-digits
```

There are three standard functions dealing with sets. The

functions FIRST_ELEM and LAST_ELEM accept a set as an argument and return the scalar value of the smallest and largest elements respectively. It is an error if either is passed the empty set. The function CARD accepts a set as an argument and returns the cardinality of it.

F.2.1.4. String

```
string type :
   ── STRING ── ( ── <expression> ── ) ──►
```

The string types are denoted by the reserved word STRING followed by a literal expression giving the maximum string length. String variables in WISCH may have values of a varying length, up to the declared maximum. An implementation may place limits on the range of allowed maximum lengths. Values of string types are sequences of characters from an implementation defined set (the same as the set used for the type CHAR). Literals of string types are written as sequences of characters enclosed in quotes ("). If an implementation provides alternate character types, it must also provide string types for each character set. The maximum length of a literal is its exact length, for example the literal "a string" is a STRING(8) value. The quote may be included in a string literal by writing it twice, for example "a quote "" " is a STRING(10) (<u>not</u> 11) value.

The empty string is "". String literals are <u>not</u> allowed to cross line boundaries in the source program. It is illegal to attempt to assign a value to a string variable which is longer than the variable's maximum length. Non-printable characters may not be denoted directly in a string literal.

There is one operator on string values, %, the concatenation operator which has a priority of 1. This operator also accepts character values as either or both arguments, treating them as though they were STRING(1) values. The relational operators for inequality (<, ¬<, etc) perform standard lexical comparisons, with the collating sequence being the ordering of the appropriate character set. If all "existing" characters match, the shorter string is always "less" than the longer. These operators will also accept mixtures of string and character expressions.

Note that since WISCH allows constant expressions, the restriction to single line strings and no non-printables is really no problem at all:

```
CONST
   alpha_string = "abcdefghijklmnopqrstuvwxyz" %
                  "ABCDEFGHIJKLMNOPQRSTUVWXYZ" ;
   special_string = "ascii control chars:" %
                    " bell " % CHAR(7) %
                    "   cr " % CHAR(13) %
                    "   lf " % CHAR(12) ;
```

Substrings may be selected by suffixing a string variable with a substring selector, two integer expressions specifying the initial character for the substring and the length of it. For example:

```
string_var(start,length)
```

Characters in a string are numbered from the left starting at one. The value of a substring selection is the "existing" portion of the selected string. For example, given the code segment:

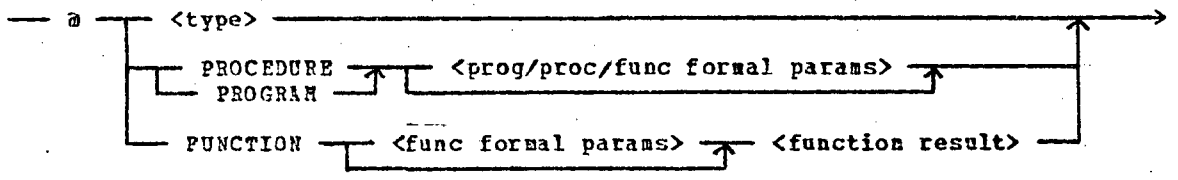
```
VAR  s : string(10);
     s := "12345";
```

the value of s(3,5) is "345" and the value of s(7,4) is the null string.Assignment to a substring is allowed and causes <u>replacement</u> of the selected string by the right hand side. The start of the substring must not be beyond the end of the current string value. (This is to prevent the subsequent value from having "holes" in it.)

There are two standard functions and one standard procedure for strings. The function LENGTH accepts a string as an argument and returns its current length. The function FIND_CHAR accepts a string and a character as arguments, returning the position of the first occurence of the character in the string. It returns zero if the character is not present. The procedure PAD_STRING accepts a string variable and a character as arguments. It pads the value of the variable to its maximum length with the character.

F.2.1.5. Normal pointer

normal pointer type :



The space for most variables is allocated at scope entry and deallocated at scope exit. It is also possible to allocate data objects dynamically and have them continue to exist for an indefinite period. Such objects cannot be referenced in the usual manner, but must be accessed via "pointers" to them. The "normal" pointers described here (as opposed to "pool" pointers) may point to dynamically created objects used completely within one program. For the rest of this section (only), the word pointer means normal pointer.

Pointer types are declared by preceding a declaration of the object type with an at-sign (@). To allow self-referencing types, a pointer type may be defined before or inside of its object type. Pointers values may only be references to values of the declared object type. Pointer types may also be declared to point to programs, procedures or functions. Their values are then references to a code object instead of a data object.

There is only one literal of any pointer type, the standard name NIL. The value of NIL is a reference to no object. There are only two operators that accept pointer arguments, the equality operators = and ¬=. The object pointed to is obtained by suffixing a pointer variable name with an at-sign (@).

Data objects are created by the standard procedure NEW. It accepts a pointer variable as an argument, creates a data object of the appropriate type and assigns a reference to the object to the pointer variable. These objects are created from a heap data area, separate from the area where normal variables are allocated. WISCH <u>does</u> <u>not</u> provide any automatic garbage collection. Dynamic objects which are no longer needed may be returned to the heap by the standard procedure DISPOSE. It accepts a pointer variable as an argument, returns the object that it references to the heap and sets the variable to NIL. Note that the security of the heap <u>is</u> <u>not</u> guaranteed if the programmer uses DISPOSE. It is the programmer's responsibility to eliminate "dangling references" to recycled storage.

Pointer types that are declared to reference programs, procedures or functions also have the value NIL and the equality operators, but since code objects cannot be created dynamically, the NEW procedure cannot be used with them. Instead, the standard function ADDR may be used to obtain a reference to a code object. The ADDR function may only be applied to (and hence references may only be obtained to) programs, procedures

and functions which are external or are declared in the outermost scope of the program. Dereferencing a pointer to a program may only be done in a CREATE statement and yields the program pointed to. Dereferencing a pointer to a procedure or function causes it to be called. Although the formal parameter names of the object type are included in the declaration of the pointer, they are present for mnemonic reasons only and are completely ignored. The formal parameters of the objects referenced must only agree in number, type and order with the pointer's object type.

Some examples of pointer types and their use are:

```
TYPE
   int = INTEGER;
   int_ptr = @ INTEGER;
   node_ptr = @ node;
   node = RECORD
             op : CHAR;
             left: node_ptr; {node_ptr and "@ node"}
             right: @ node;  {are equivalent types.}
          ENDRECORD;
   func_ptr = @ FUNCTION (i: int) RETURNS (j : int);
   foo = @ PROCEDURE (bar: foo); {note circular reference}

VAR
   i_p : int_ptr;
   root : node_ptr;
   func : func_ptr;
   i : INTEGER;

FUNCTION factorial (n: int) RETURNS (i : int);  EXTERNAL;

BEGIN
   NEW(i_p);
   i_p@ := 0;

   root := NIL;

   func := ADDR(factorial);

   i := func@(i);
```

F.2.2. Array

array type :

```
— ARRAY — ( —↓— <scalar type> —┘— ) — OF — <type> ——→
                  ,
```

The array types are composite types, with all components of the same type. The components are selected by indices which may be of any scalar type. An array type is assignable and/or sharable if its component type is. There are no operators on values of array types except the equality tests which are defined as an element by element test. An array value is written similiarly to a set value. The "subrange" notation allowed for set values is also allowed here. In addition, a repetition factor may be applied to a component. This is written as "repetitions ! component". If all components are literal expressions, the value denoted is an array literal. Note that a component of a one dimensional array is a simple element, of a two dimensional array is a row, etc. For example, given the declarations:

```
TYPE
   vector = ARRAY (1..5) OF INTEGER;
   vecvec = ARRAY (1..2) OF vector;
   matrix = ARRAY (1..2, 1..5) OF INTEGER;
```

the following:

```
vector(-1,3!0,1)
```

```
vecvec(2 ! vector(1,2,3,4,5))

matrix(2!(5,4,3,2,1))
```

are equivalent to:

```
vector(-1,0,0,0,1)

vecvec(vector(1,2,3,4,5),vector(1,2,3,4,5))

matrix((5,4,3,2,1),(5,4,3,2,1))
```

Some examples of array types are:

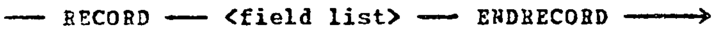
```
CONST  prime = {an appropriate value} ;

TYPE
   name_table = ARRAY(0..prime-1) OF STRING(10);
   vec  = ARRAY (-1..+1) OF INTEGER;
   mat1 = ARRAY (1..2) OF vec;
   mat2 = ARRAY (1..2, -1..+1) OF INTEGER;

VAR
   old_names, new_names : name_table;
   nums1 : mat1;
   nums2 : mat2;
   letter_counts : ARRAY ('A..'Z) OF INTEGER;

BEGIN
   FOR ch := 'A TO 'Z LOOP letter_counts(ch) := 0 NEXT;
   nums1 := mat1(vec(1,2,3),vec(4,5,6));
   nums2 := mat2((1,2,3),(4,5,6));
   FOR i := -1 TO +1 LOOP nums1(1)(i) +:= nums1(2)(i) NEXT;
   FOR i := -1 TO +1 LOOP nums2(1,i)  +:= nums1(2,i)  NEXT;
```

Note in particular the similarities and differences between the denotations of values of types mat1 (array of arrays) and mat2 (two dimensional array).

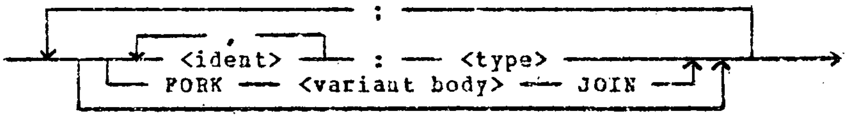
The standard functions LO_BND and UP_BND may be used to find the lower and upper limits of any dimension of an array. They accept an array type or expression and an integer as arguments. The integer argument selects the dimension whose bound is wanted. It may be omitted for one dimensional arrays.

F.2.3. Record

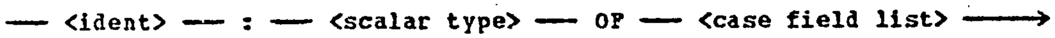
record type :



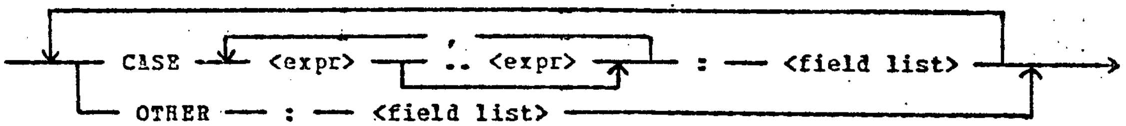
field list :



variant body :



case field list :



The record types are composite types with components of possibly different types. The components are selected by identifiers known as field selectors. The full name of a record field is the record name, a dot (.) and the field identifier. The identifiers of all fields in a record must be different, but may be the same as identifiers in other (possibly nested) records or identifiers outside the record. Specifically, the <u>full</u> names of all record fields must be unique in all scopes where that name is known.

Record types may also have "variant" portions whose exact structure is determined by a "tag field". The value of the tag field determines which of the cases in the FORK clause apply. Field names in different cases must be different from each other and different from all other field names in the record.

Values of a record type (with no variants) are written like array values, except that the components are of possibly different types and the repetition notation may not be used. Values may not be written for records with variant portions. Some examples of record types are:

```
TYPE
   binary_tree_node =
      RECORD
         kind : 0..10;
         left, right : @ bin_tree_node;
      ENDRECORD;

   dev_kind = [disk, mag_tape, paper_tape, terminal];

   status_rec =
      RECORD
         up : BOOLEAN;
         channel : INTEGER;
         io_wait_list : @ process_list_node;
         FORK kind : dev_kind OF
            CASE disk :
               block_size : INTEGER;
            CASE mag_tape :
               density : INTEGER;
               nine_track : BOOLEAN;
            OTHER : {empty};
         JOIN;
      ENDRECORD;
```

F.2.4. Pool pointer

```
pool pointer type :
   ── @ ── POOLED ── <type> ──────>
```

Pool pointers are the second pointer type in WISCH. Where normal pointers may point only to heap or "code" objects known within a single program, pool pointers may only point to pooled data objects. Since pools may only contain objects of a shareable type, pool pointers may only point to objects of a shareable type. They are not assignable and may only be compared with the value NIL (not with each other since they will never be equal). Upon creation, all variables of pool pointer types are initialized to the value NIL. Shared variables are created by the standard procedure NEW, but are not necessarily allocated from the same heap as non-shared variables. The procedure DISPOSE may also be used to return shared variables to free storage. A reference to a shared object may be assigned to a pool pointer only by the NEW procedure or the AWAIT statement. In either case the pointer must have the value NIL immediately prior to the assignment. This reference is removed when the pointer is subsequenly used in a SEND statement. It is "guaranteed" that no shared object can be pointed to by more than one pool pointer at a time. (The guarantee is not absolute since there are ways in WISCH to circumvent all checks if one is

really so inclined.) Pool pointers may not be dereferenced except via the BIND statement.

F.2.5. Pool

```
pool type :
   ── POOL ── OF ── <type> ──►
```

The basic semantics of pools have already been discussed. Pools are collections of message buffers that are shared among processes for the purpose of communication. Pools may only contain objects of shareable types. The objects in the pool actually consist of an inaccessible portion used for linkage, etc., and the accessible buffer portion. They may be viewed as linked lists of data objects, all of the same type. All pools are empty when created. If the object type of a pool is or contains a dynamic subrange, the range expressions are not evaluated and associated with the pool variable. When a shared variable is created by the NEW procedure, any dynamic subrange bounds are evaluated and fixed for that one object. The same pool pointer may later be used to create another shared variable with different subrange limits. Since a pool variable is really just a header for a linked list of shared variables, the pool may contain several variables with different limits for any dynamic subranges. Thus, one could create a pool which

contained varying length one dimensional vectors of storage units. For example:

TYPE buffer_pool = POOL OF ARRAY (lwb..upb) OF STORAGE;

where lwb and upb are variables declared elsewhere. There are no literals, constants, or operators for pool types. Only variables and VAR parameters may be of pool types. Variables of pool types must be declared in the outermost scope of a program.

There are two standard functions, EMPTY and WAITING, which accept a pool variable and return a boolean value. EMPTY returns TRUE if there are no objects left in the pool. WAITING returns TRUE if there are any process waiting for access to objects from the pool. Note that WAITING implies EMPTY. References to pooled objects are obtained and relinquished using the AWAIT and SEND statements.

All pooled objects have a pool associated with them, the reply pool, which may be used to tell a process where to return the object. The reply pool is initially a standard "garbage" pool, but may be changed in the SEND statement.

F.2.6. Process

The type process is denoted by the standard type name PROCESS. Only variables and VAR parameters may be of type process. Variables must be declared in the outermost scope of a program. There are no literals or constants of process type,

and no operators on values of process type. Process variables may only be acted upon by the CREATE, START, STOP and DESTROY statements.

A process may be in one of four states. It may not exist (a process variable which could be said to have a void value). It may exist, but be stopped, in which case it cannot execute further until restarted. It may be ready, in which case it is ready to be executed by a processor at any time. Lastly, it may be blocked, this occurs when it is waiting on some external event (usually waiting on a request for a pooled object). There is a standard enumeration type, process_status_type, and a standard function, process_status, that may be used to determine the current status of a process. They are defined as:

```
TYPE
   process_status_type = [void, stopped, blocked, ready];
FUNCTION process_status
         (CONST proc : PROCESS) : process_status_type;
         {return the current status of proc};
```

F.2.7. Compatability, coercions and attributes

Two levels of compatibility are defined in WISCH. A type x is weakly compatable with a type y if the intersection of their sets of values is not empty. A type x is strongly compatable with a type y if the values of x are a superset of (or the same as) the values of y.

For example, given the declarations:

```
TYPE
   zero_to_ten = 0..10;
   one_to_nine = 1..9;
```

The type zero_to_ten is strongly compatable with one_to_nine, but one_to_nine is not strongly compatable with zero_to_ten. They are of course weakly compatable with each other.

Note that weak compatability is a commutative relation but strong compatability is not. Also, x strongly compatable with y implies at least y weakly compatible with x. Intuitively, if x is weakly compatable with y, then values of type y may be assigned to a variable of type x with appropriate run time checks for range errors. If x is strongly compatable with y, then values of type y may <u>always</u> be assigned to a variable of type x with <u>no checking</u> at run time.

The standard types BOOLEAN, CHAR, INTEGER, REAL, STORAGE and PROCESS are of course compatable (strongly and weakly) with themselves.

Any two subrange types are weakly compatable if they are defined on the same basis type and have a non-null intersection. A scalar type x is strongly compatable with a scalar type y if y is a subrange of x. (Note that any scalar type is a subrange of itself.)

Two set types are compatable (weakly and strongly) if their basis types are the same.

Any two string types are weakly compatable. String type x is strongly compatable with string type y if the maximum length of x is at least as large as that for y.

Two normal pointer types are compatable (weakly and strongly) if their object types are strongly compatable.

Two pool pointer types are compatable (weakly and strongly) if their object types are strongly compatable.

Two pool types are compatable (weakly and strongly) if their object types are strongly compatable.

Any two module types are weakly (strongly) compatable if all exported names are the same and are weakly (strongly) compatable.

Two array types are weakly (strongly) compatable if their index types are identical and their component types are weakly (strongly) compatable.

Two record types are compatable (weakly and strongly) if all fields have the same names (in the same order) and are all of strongly compatable types.

The use of (and need for) type compatablilty is discussed in the sections about programs, procedures, functions, the assignment statement and expressions.

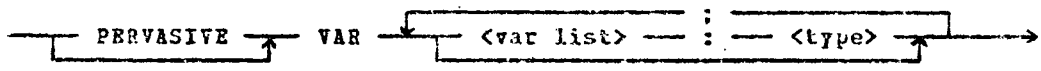
There are only two type coercions in WISCH. Where appropriate, the types CHAR and STRING(1) are coerced to each other. and INTEGER may be coerced to REAL. Note that these coercions <u>only</u> apply to uses in expressions. For example, an integer value may be used as an argument to a procedure where the corresponding formal parameter is a CONST or plain parameter of type REAL, but it <u>may not</u> be used if the corresponding formal parameter is a VAR parameter of type REAL. (See the discussion on parameters of programs, procedures and functions for an explanation of CONST, VAR and plain parameters.)

However, transfer functions do exist for programmed conversions. Recall that any scalar type name may be used as a generic transfer function to convert scalar values to the named type. Inside a machine dependent program, procedure or function, <u>any</u> type name may be used as a generic function with a single argument to convert the type of the argument to the named type. The sizes of values of the two types must be the same. (A routine is declared "machine dependent" by using the OPTION statement.)

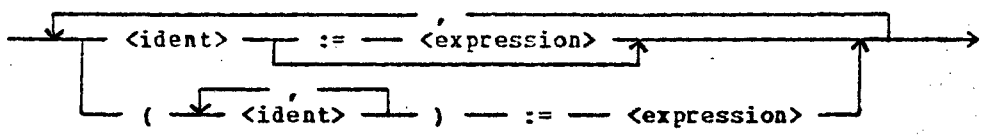
Standard functions exist to access various attributes of data types. The FIRST and LAST functions for scalar types have already been mentioned. The VALUE_SIZE function provides information about the space required to hold values of any type. It accepts a variable or type name and returns an integer, the number of storage units needed to hold a value of that type. The function TOTAL_SIZE is similar, except it returns the total number of storage units needed by a variable of that type. For example, TOTAL_SIZE gives the number of storage units of heap space that will be used if a variable of that type is created by the NEW procedure. The functions LO_BND and UP_BND accept two arguments, the name of an array variable and an integer expression, and return the lower and upper bounds of the appropriate dimension of the array respectively. Of course, the integer expression must have a value between one and the number of dimensions of the array (inclusive).

F.3. Variable

variable declaration :



var list :



Variables are declared via the VAR declaration. All variables used in a program must be declared before use. If a variable is of an assignable type, an initial value may be specified for it. Each time the variable is created (at a scope entry) the expression is evaluated and the value is assigned to the variable. Note that although pool pointer variables are not assignable, they are always initialized to NIL when created (by scope entry or use of the NEW procedure). The initial value expression refers only to the identifier (or list of identifiers on its immediate left. For example:

```
VAR i, j:=0, k : INTEGER;
```

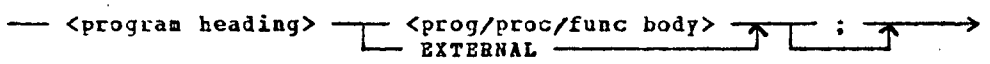
causes only j to be initialized, while:

```
VAR (i,j):=0, k : INTEGER;
```

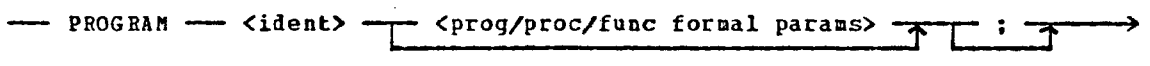
causes both i and j to be initialized.

F.4. Program

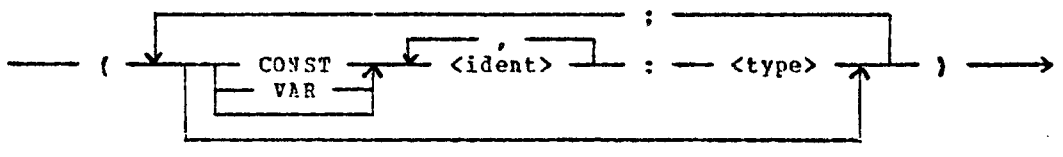
program declaration :



program heading :



prog/proc/func formal params :



prog/proc/func body :

```
---- <decls> -- BEGIN -- <ident> -- <statements> -- END -- <ident> ---->
```

Most of the features of programs have already been presented. All that really remains is to discuss formal parameters.

Programs (and procedures) may have three kinds of formal parameters, VAR, CONST and plain. For VAR parameters, the types of actual arguments must be <u>strongly</u> compatable with the corresponding formal parameters. For CONST and plain parameters, the actual arguments need only be <u>weakly</u> compatable with the corresponding formal parameters.

Var parameters are passed by reference, that is a use of the formal parameter in the program is really a use of the actual argument passed via the CREATE statement. All "name computation" (array subscripting, pointer dereferencing, etc.) for VAR parameters is done once, while arguments are being evaluated in the CREATE statement. The actual argument for a VAR parameter must be a variable name. The only VAR parameters allowed for programs are pool variables.

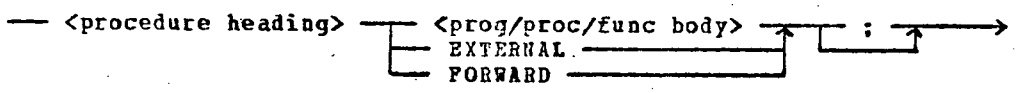
CONST parameters are treated as constants in the program. Their value is the value of the actual argument, which must be an expression (of which a literal or variable reference is a

trivial kind). Note that CONST parameters can only be of an assignable type.

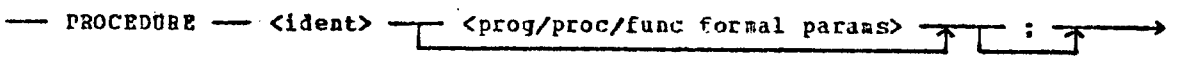
Plain parameters are those for which neither VAR or CONST is specified. They look like local variables which are initialized to the value of the actual argument (i.e. passed by value). The actual argument must be an expression.

F.5. Procedure

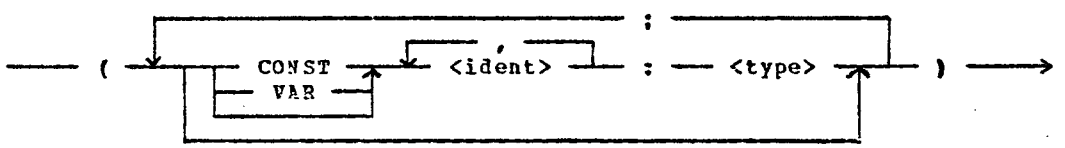
procedure declaration :



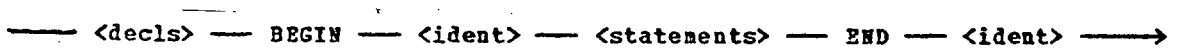
procedure heading :



prog/proc/func formal params :



prog/proc/func body :

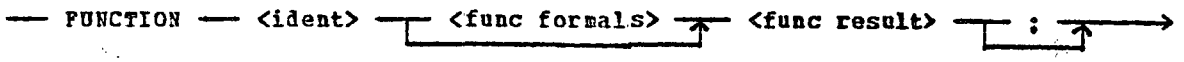


All procedures in WISCH are capable of being called recursively. They may be nested to any depth. Formal parameters for procedures are the same as for programs, except that procedures may have VAR parameters of any type. Name computation for VAR parameters is done only once, while arguments are being evaluated prior to the actual call.

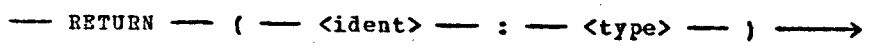
F.6. Function

function declaration :

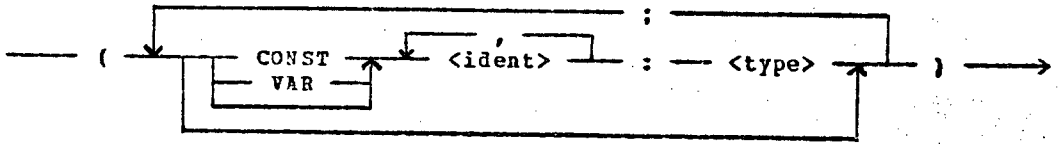
function heading :



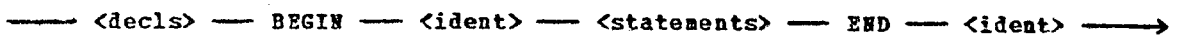
function result :



prog/proc/func formal params :



prog/proc/func body :

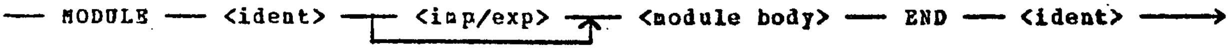


Like procedures, all functions are recursive and may be nested to any depth. Function formal parameters are the same as for programs and procedures. The function value is returned via

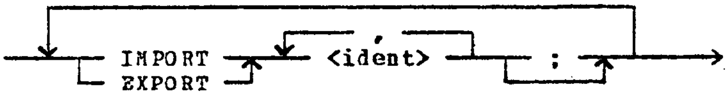
the RETURNS variable. Within the function looks like a local variable. When the function returns, its value becomes the value of the function.

F.7. Module

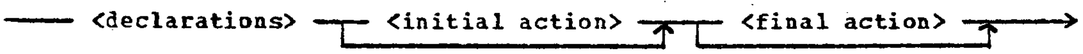
module declaration :



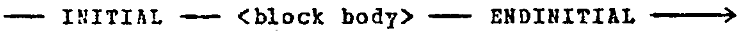
imp/exp :



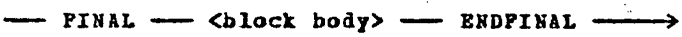
module body :



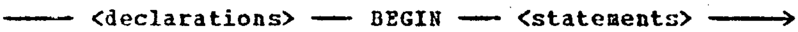
initial action :



final action :



block body :



Modules provide an means to encapsulate variables and routines. The modules of WISCH are derived from those of MODULA. They allow the implementation details of an abstract design to be hidden from the users of that design. Identifiers

defined inside a module are unknown outside of that module unless they are explicitly exported. Exported variables are read only outside of the module. Exported types are "opaque", variables of that type may be declared outside of the module but they may not be operated on externally since the structure of the type is unknown. They may only be operated on by routines defined within the module. Identifiers known in the scope surrounding the module are unknown within the module unless they are explicitly imported (or are PERVASIVE).

When a module is created (when its containing scope is entered), the optional INITIAL block is executed. This can be used for any necessary initialization of internal (or external) data structures. When the containing scope is exited the optional FINAL block is executed, providing a means to do any final cleanup, reporting, etc.

An example of a module is:

```
MODULE  symbol_table_manager
   IMPORT table_size, sym_type;
   EXPORT enter_sym, find_sym;

   VAR
      sym_tab : ARRAY (0..table_size-1) OF sym_type;
      num_entries, num_collisions : INTEGER;

   PROCEDURE enter_sym (new_sym : sym_type);
      {add new_sym to table if not there now};
   END enter_sym;

   PROCEDURE find_sym
            (sym:sym_type; VAR found:BOOLEAN);
      {look up sym (by name), if found then}
      {fill in the rest of the info to sym };
   END find_sym;

   INITIAL
      BEGIN
         (num_entries, num_collisions) := 0;
   ENDINITIAL

   FINAL
      BEGIN
         {report statistics on use, collisions};
   ENDFINAL

END symbol_table_manager;
```

F.8. Scope of names

The region of source text in which a particular name (and the object it names) is known is called the scope of that name. These nested, delimited regions are called scopes. The scope delimiters in WISCH are program declarations, procedure declarations, function declarations, module declarations, blocks, FOR statements and BIND statements. All references to a

name are to the innermost declaration of the name, working outward from the reference.

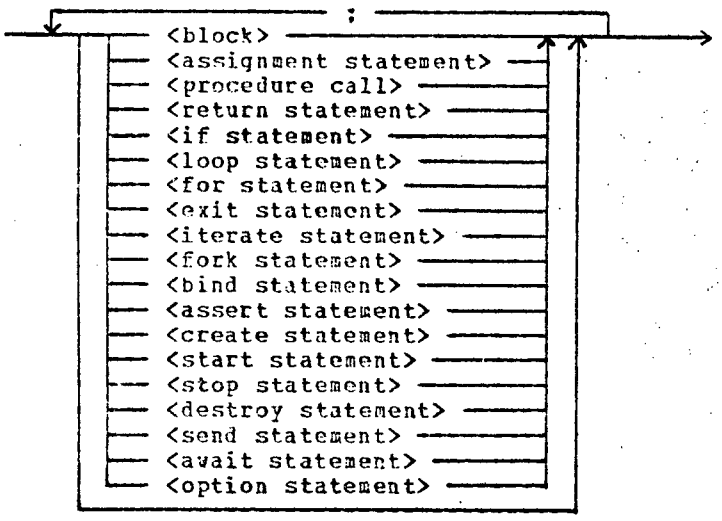
Normally, the only names known within a program are the formal parameter names and locally declared names. All names declared in outer scopes are unknown. This rule is excepted for PERVASIVE names, which are known in all inner scopes and cannot be redeclared. Any constant or type name may be PERVASIVE, but the only variables that may be are pool variables. Procedures, functions and blocks know the innermost declaration of all names declared in outer scopes. The only (non-PERVASIVE) external names known in a module are those which are explicitly imported. Names which are exported from a module are known as though declared in the immediately surrounding scope. The index variable of the FOR statement is known only within the statement body, where it is a constant. The binding identifier of the BIND statement is known only within the statement body, where it is a normal variable.

Once a name has been used in a scope (e.g., in an expression or as a type identifier), it may not be redeclared in that scope (but may be in an inner scope). For example, the following is illegal:

```
CONST max_size = 1000;
PROCEDURE foo;
   VAR index : 0..max_size;
   CONST max_size = 100;
```

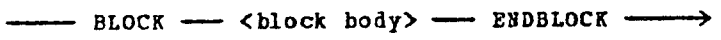
G. Statements

statements :

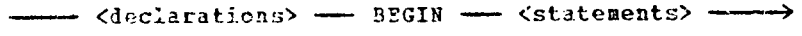


G.1. Block

block :



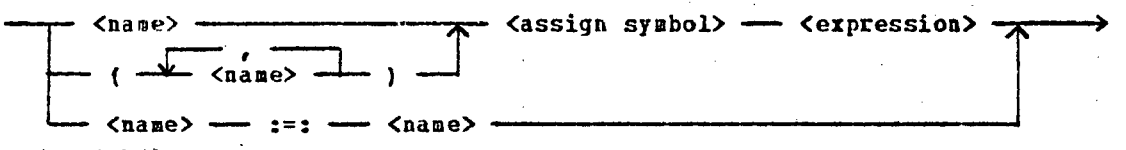
block body :



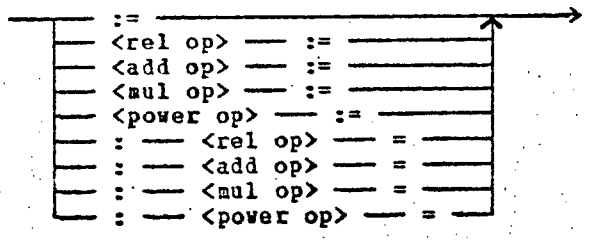
The block opens a new scope, allowing local declarations to be made. It is essentially an inline, parameterless procedure.

G.2. Assignment

assignment statement :



assign symbol : (no enbedded blanks)



WISCH has a large variety of assignment symbols. The standard assignment uses := to assign the value of the expression on the right to the storage specified by the name on the left. The name is evaluated <u>after</u> the expression to be assigned. The type of the name must be weakly compatable with the type of the expression. If it is strongly compatable, no run-time checks are needed; otherwise a check must be made before the assignment to ensure type sequrity. The symbol :=: performs a storage swap. For simple values, a :=: b is equivalent to (but perhaps more efficient than):

```
BLOCK
   VAR t : type_of_a_and_b;
   BEGIN
      t := a;    a := b;    b := t;
ENDBLOCK;
```

For overlapping values (e.g., two substrings of the same variable) the result of a :=: b is implementation defined.

The other assignment symbols are all very similar to one another, being shorthand for "update" operations on variables. They may be divided into two classes by form (and function). The first class is of the form "op:=", the second is ":op=". For the first, a op:= b is equivalent to a := a op b, except that the name of "a" is evaluated only once in the former case. Similarly, a :op= b is equivalent to a := b op a. For example:

```
i +:= 1
```

increments the integer variable i by 1, and

```
s :%= "value of s: "
```

concatenates "value of s: " onto the <u>front</u> of string variable s. There is one of these symbols (in each form) for each binary operator (for which it makes sense).

Multiple assignments are allowed for all assignment operators except the swap operator by writing a parenthesized list of variable names to be assigned to. For example:

```
(a,b,c) +:= expr;
```

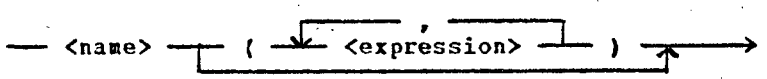
is equivalent to:

```
a +:= expr; b +:= expr; c +:= expr;
```

except expr is evaluated only <u>once</u> in the former case, before any of the names.

G.3. Procedure call

procedure call :



Procedures are called by simply writing the procedure name and supplying a list of actual arguments. The arguments are evaluated from left to right, and must be strongly compatable with their corresponding formal parameters.

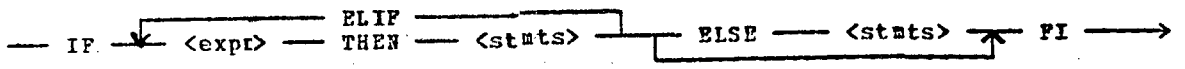
G.4. Return

return statement :

The return statement is simply the word RETURN. It causes an immediate return from the current procedure or function. It is illegal to attempt to return from the outer level of a program. (To stop a program use the STOP statement.)

G.5. If

if statement :



The IF statement allows arbitrary sequences of statements for either branch. The ELIF clause allows concise coding of the common IF-THEN-ELSE-IF sequence. For example:

```
IF b1 THEN
   s1;
   s2;
ELSE
   IF b2 THEN
      s3;
      s4;
   FI;
FI;
```

can be compressed to:

```
IF b1 THEN
   s1;
   s2;
ELIF b2 THEN
   s3;
   s4;
FI;
```

G.6. Loop

loop statement :

```
── LOOP ── <statements> ── NEXT ──►
```

The indefinite looping structure of WISCH is actually defined via two statements, LOOP and EXIT. The LOOP statement delimits a range of statements to be repeated until terminated by executing an EXIT statement. This provides a single structure to handle all common forms of looping such as the "while" and "repeat" statements of PASCAL.

The traditional "WHILE b DO s" form can be written as:

```
LOOP
   EXIT UNLESS b;
   s;
NEXT;
```

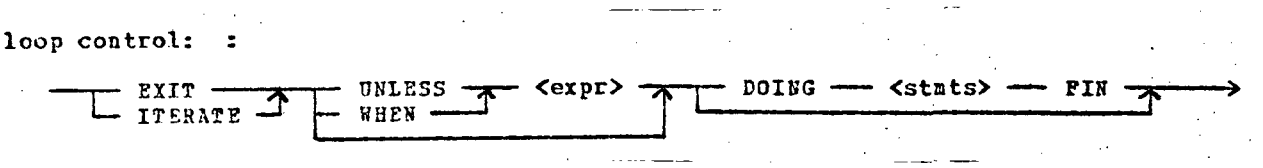
The "REPEAT s UNTIL b" form can be written as:

```
LOOP
   s;
   EXIT WHEN b;
NEXT;
```

The "n+1/2" times loop can be written as:

```
LOOP
   s1;
   EXIT WHEN b;
   s2;
NEXT;
```

G.7. Exit



The EXIT statement is used to immediately exit from the closest enclosing LOOP or FOR within the current program, procedure or function body. Use of the EXIT statement when not inside a loop (in the current program, etc.) is illegal. In other words, the EXIT must have something to exit from, and cannot cause execution to leave the current program, procedure or function. It is legal though to EXIT from a local BLOCK. The exit will occur unconditionally if no WHEN clause is present, otherwise it will occur if the boolean expression is true. The DOING clause allows specification of a "postlude" action, useful for multi-level exits and "side effects" for specific exit conditions.

For example, consider the following code to do disk track allocation:

```
new_track := -1;
FOR cylinder := 0 TO last_cylinder LOOP
   FOR track := 0 TO tracks_per_cylinder-1 LOOP
      EXIT WHEN ¬reserved(cylinder,track) DOING
         new_track := cylinder*tracks_per_cylinder + track;
         EXIT;  {from outer FOR also}
      FIN;
   NEXT;
NEXT;
IF new_track ¬= -1 THEN . . .
```

When a free track is found, both loops are exited and new_track

is assigned the value of the track. If no tracks are free, the outer loop will finish leaving new_track as -1. This loop construct (and example) are due to M. Yasumura [YAS 77].

G.8. For

for statement :

```
-- FOR -- <for control> -- LOOP -- <statements> -- NEXT -->
```

for control :

```
-- <ident> -- := -- <expr> --+-- TO -- <expr> --+-- BY -- <expr> --+-->
                             |                |                   |
                             +-- BY -- <expr> -+-- TO -- <expr> --+
```

The FOR statement is very similar to those found in other ALGOL-like languages. The index variable is local to the loop and is a constant within it. The ":=" and "TO" control expressions may be of any scalar type (but must be of the same type) and the "BY" expression must have an integer value. The semantics are roughly to step the index variable from the := value to the TO value, using every ABS(BY)'th value inbetween. The sign of the BY expression determines whether the loop is ascending (positive) or descending (negative). The BY expression must not be zero. If the initial value is beyond the final at the start, the loop body isn't entered. All three control expressions are evaluated only once, at the beginning of the loop. The FOR statement may also be exited early via an EXIT statement.

The precise semantics of the FOR statement are illustrated by the following WISCH code:

```
FOR index := start TO final BY step LOOP statements NEXT
```

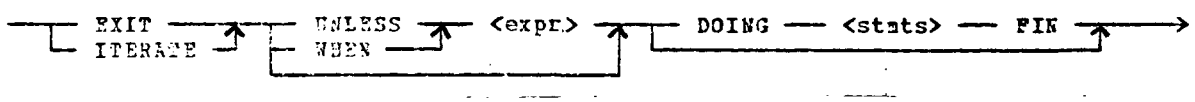
is equivalent to:

```
BLOCK
   TYPE
      int = INTEGER;
      index_type = {type of start and final};
   CONST
      t1 = final;
      t2 = step;
   VAR
      t3:=start : index_type;
   BEGIN
      LOOP
         EXIT WHEN (int(t3)*SIGN(t2)) > (int(t1)*SIGN(t2));
         BLOCK
            CONST
               index = t3;
            BEGIN
               statements;
         ENDBLOCK;
         t3 := index_type(int(t3)+t2);
      NEXT;
ENDBLOCK;
```

where t1, t2 and t3 are not otherwise used in the program.

G.9. Iterate

loop control: :

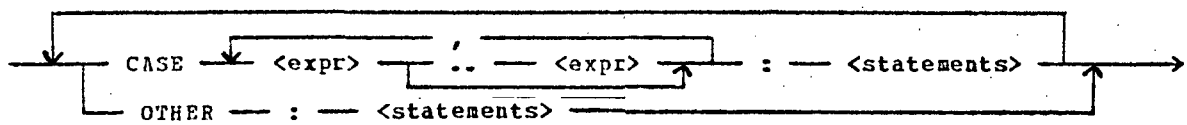


The ITERATE statement causes an immediate jump to the next iteration of a LOOP or FOR statement. For a LOOP statement, this means to the start of the loop body and for a FOR statement to the "increment and test" portion. Like the EXIT statement, it may only be used inside a LOOP or FOR statement in the current program, procedure or function.

G.10. Fork

fork statement :

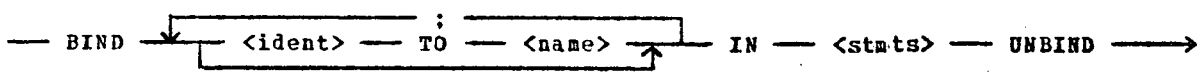
case stmt list :



The FORK statement is WISCH's version of what is known in some languages as a "case" statement. The branch expression must have a scalar value, which selects one of the labeled case lists for execution. The case labels must be literal expressions agreeing in type with the branch expression. A particular case may be labeled by any number of values. The "subrange" form of case label is short for the list of all values in the range. The OTHER case is selected if none of the explicit labels match the expression value. It is an error if no OTHER case is present and none of the explicit labels match.

G.11. Bind

bind statement :



The BIND statement creates a scope in which a (long and complex) variable name is bound to some (short and simple) identifier (the bound identifier). On entry to the statement, the variable name is "evaluated", possibly involving such actions as array subscripting, record field selection and/or pointer dereferencing. The bound identifier is then made to refer to the selected storage location and has the same type as the name. Subsequent changes to the "name" do not change the location referenced by the bound identifier. Pool pointers may only be dereferenced when used as part of the variable name in a BIND statement. A pool pointer may not be altered (by a SEND or AWAIT) while it is dereferenced in a BIND statement.

G.12. Assert

assert statement :

The ASSERT statement evaluates the boolean expression, and if it is not TRUE, causes the process to stop with a fatal error.

G.13. Create

create statement :

```
-- CREATE -- <name> -- FROM -- <name> -- <create options> -->
```

create options : (must be in order from top to bottom)

```
--+--------------------------------------------+--+-->
  |  +-- ERROR -- <name> ----------------------+  |
  |  +-- PRIORITY -- <expression> -------------+  |
  |  +-- STACK -- <expression> ----------------+  |
  |  +-- HEAP -- <expression> -----------------+  |
  |  +-- TIME -- <expression> -----------------+  |
  +-----------------------------------------------+
```

The CREATE statement creates a new process modeled after the specified program. The process variable must not already refer to an existing process.

G.14. Start

start statement :

```
-- START -- <name> -->
```

The START statement removes a process from the stopped state. Where it goes depends on its state when it was stopped. If it was waiting, it will go back to a waiting state, otherwise must have been in a started state and will join the list of processes ready to be executed. It will then be given processor time in some implementation defined manner (the only guarantee is that it will be given processor time eventually).

G.15. Stop

stop statement :

```
— STOP ─┬─ <name> ──────→
        └─ MYSELF ──┘
```

The STOP statement removes a process from the started or waiting states and puts it in the stopped state. As long as it is stopped, it will not get any processor time, nor will it be given any shared objects if it was waiting.

G.16. Destroy

destroy statement :

```
— DESTROY — <name> ──→
```

The DESTROY statement destroys the process, which must be in a stopped state. All pooled objects to which the object has references are returned to their owner pools.

G.17. Await

```
await statement :
   ── AWAIT ── <name> ── FROM ── <name> ──►
```

The AWAIT statement requests that an object be allocated from a pool and the reference to it assigned to a pool pointer. The pointer must have the value NIL before the assignment. If an object is available the process is given it and allowed to continue. If the pool is empty, the process is put in the waiting state and attached to the pool.

G.18. Send

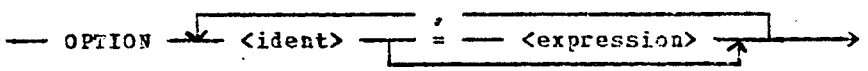
```
send statement :
   ── SEND ── <name> ── TO ┬─ <name> ──┬── RESPOND ── <name> ──►
                          └─ SENDER ──┘
```

The SEND statement sends an object pointed to by a pool pointer to a pool, allowing the sending process to continue. If the pool is empty and there are any processes in the waiting

state, one of them is given access to the object and placed in the started state. If OWNER is specified in the TO clause, the object is sent to its owner pool. If SENDER is specified, it is sent to the response pool given in the <u>previous</u> SEND on that object. The RESPOND clause specifies the pool to be used in future "returns to sender". If it is omitted, the field is left unchanged.

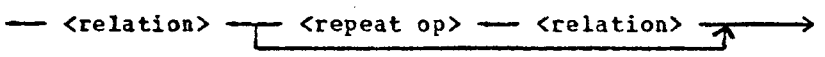
G.19. Option

option statement :

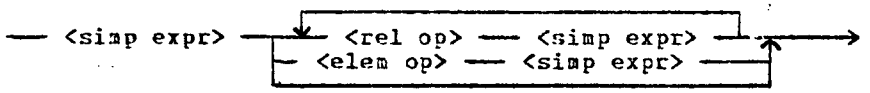


H. Expressions

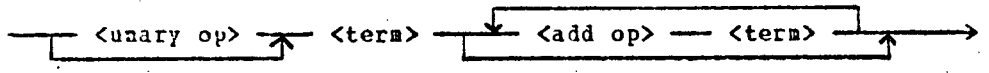
expression :



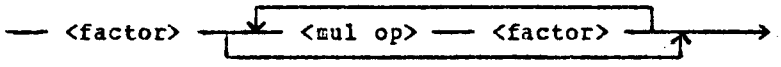
relation :



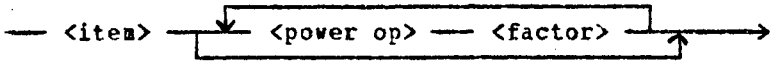
simp expr :



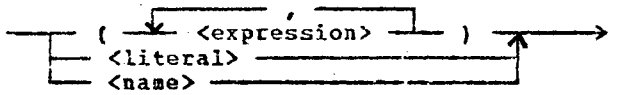
term :



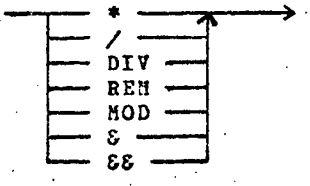
factor :



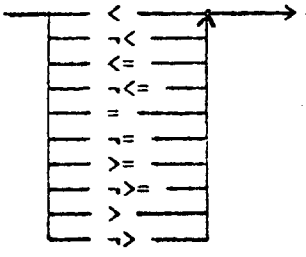
item :



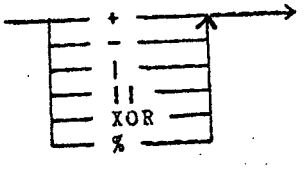
mul op :



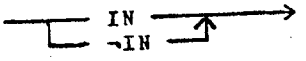
rel op :



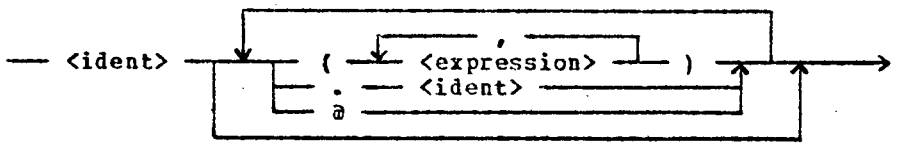
add op :



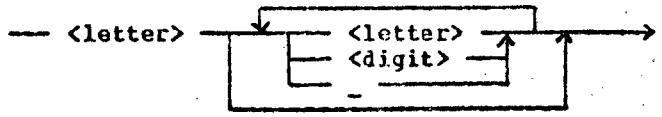
elem op :



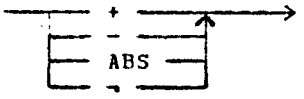
name :



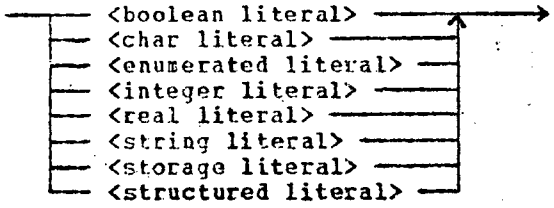
ident : (no embedded blanks, last char cannot be _)



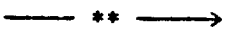
unary op :



literal :



power op :



repeat op :

While this report discusses expressions in terms of "typed" expressions, ie. integer, boolean, etc., and programmers usually think of them in these terms, it is very important to realize that the analysis of expressions in WISCH is governed solely by the precedence of operators and not by the types of the operands. This is not to say that types are ignored, on the contrary, WISCH is a strongly typed language and all operands must have suitable types for their operators. What this does say is that the determination of which operators have which operands is done on the basis of operator precedence only. This is especially noticeable in expressions which contain boolean

operators (&, |, and ¬) and relational operators. Since the boolean operators are treated like other scalar operators (such as * or + for integers), they have higher priority than the relational operators. Thus common expressions such as:

```
(p ¬= NIL) & (p@.value < a)
```

must be parenthesized for proper interpretation.

The following table summarises the operators available in WISCH:

operator	prec.	operands	description
**	3	integer,real	exponentiation
*	2	integer,real	multiplication
*	2	set	set intersection
/	2	real	real division
DIV	2	integer	integer division
REM	2	integer	integer remainder
MOD	2	integer	mathematical modulus
&	2	boolean	logical and
&&	2	storage	bitwise and
+,-	1	integer,real	addition, subtraction
+,-	1	set	set union, set difference
|	1	boolean	logical or
||,XOR	1	storage	bitwise or, bitwise xor
%	1	string	string concatenation
+,-	1	integer,real	unary plus, unary minus
ABS	1	integer,real	unary absolute value
-	1	set	unary set complement
¬	1	boolean	unary logical negation
¬	1	storage	unary bitwise complement
<,¬<,<=	0	any <1>	general inequality test
¬<=,>=,	0	"	" " "
¬>=,>,¬>	0	"	" " "
=,¬=	0	any,pointer	" equality "
IN,¬IN	0	scalar & set	set element test

<1> Where "any" means any scalar, real, string or set type.

I. I/O facilities

The WISCH programmer has a need for two distinct forms of I/O facitities. First, in order to write an operating system, facilities must be provided to control and communicate with any device at the level of a device handler. Second, higher level mechanisms are needed to perform the I/O required by the higher level protions of the system, e.g., to read from and write to a "console" terminal or to access files on backing store.

The low level facilities are virtually impossible to define in a machine or even configuration independant manner. Different machines have vastly different mechanisms for controlling peripheral equipment. A single machine may have quite different means of communicating with different devices depending on the device type, use of direct memory access, etc. Because of these difficulties, WISCH does not define a single approach to I/O. Instead, an implementation must provide the necessary low level drivers as standard routines that may only be called from a machine dependant program, procedure or function.

These drivers will provide only very low level support, for example seeking a disk arm to a cylinder, reading/writing a sector from the current cylinder, requesting status information, reading/writing a single character/line to/from a terminal, etc. To the process requesting it, the I/O transfer appears as a

synchronous, indivisible operation. The WISCH programmer has no need to be at all concerned about any underlying interrupt structure. If a process initiates an I/O operation that is interrupt driven, it will become blocked until completion of the operation. Of course for operations that are not interrupt driven (e.g. the reading of a real time clock), the process need not block. Ideally, this blocking mechanism would utilitize the existing message passing facilities.

Since the structure of the high level I/O mechanisms is exactly what one is writing in WISCH, they must also be left undefined.

J. Loading and running programs dynamically

The design of a "complete" language suitable for writing "real" operating systems is complicated by a difficult issue not found in "normal" language design. An operating system must be able to dynamically load an arbitrary program into main memory and then execute that program. The new program must be able to communicate with the system, preferably in a well-controlled, secure manner. Unfortunately, this issue is only partly resolved in WISCH.

It is suggested that an implementation provide a standard procedure, LOAD, which accepts two main arguments: a VAR parameter which is a pointer to a program and a pointer to a procedure. LOAD will acquire a suitable amount of memory (the amount may be another parameter) and will call the routine passed to it to do the actual loading. (This procedure should have a large ARRAY OF STORAGE as a VAR parameter.) LOAD will set up any necessary descriptor information and make the program pointer refer to the newly loaded program. Processes may then be created from this program just like any other.

K. Implementation of WISCH

A major aspect of this research has been the actual implementation of most of the language. Due to the size of the project, the implementation effort has so far been restricted to the sequential parts of the language. Except for the multi-programming features, the only major unimplemented items are array and record literals, dynamic subranges and the OPTION statement. Several other areas are incompletely implemented as of this writing, notably many of the standard routines, string operations, the FORK statement, the BIND statement, modules, most run-time checking and "code" pointers.

The bulk of the implementation is the compiler, which produces object modules for the ASP machine (described in PART II of this thesis). The compiler is written in PASCAL and runs under MTS on the university's AMDAHL 470. The ASP linker is used to link the load modules and translate them into a form suitable for final loading on the Hewlett-Packard 21MX minicomputer owned by the department of Computer Science. The WISCH program may then be run using the ASP machine under the RTE-2 operating system on the 21MX. Thus, WISCH is currently implemented as a cross-compiled, sequential high-level language.

The compiler consists of three passes and is somewhat modelled after the Concurrent PASCAL and Sequential PASCAL compilers [HAR 77]. The first pass performs lexical analysis

and parsing, using a hand-coded recursive descent parser. It generates an intermediate code file containing a postfix representation of the parse tree. Pass two is the heart of the compiler. It builds the block structured symbol table, performs declaration analysis, evaluates literal expressions and generates code for statements. It is organized as a push-down automaton, transforming the postfix output from pass one into a "tokenized" assembly language for pass three. Pass three is relatively small; it is essentially assembler accepting the tokenized output from pass two and generating load modules for the ASP linker.

Each routine is output as a separate load module. These modules contain not just the name of the routine, but also its path name in the program tree. The compiler allows separate compilation of individual routines and the replacement of a load module is a simple task. (This must currently be done by manual editing, but a program to do it would be quite simple.) External routines must be identified by name and location in the program tree. The linker builds a tree-structured load map as it links things together. Two routines with the same name are no problem at all (provided they are not siblings).

L. Future work with WISCH

Although it is intended to be a useable tool, WISCH is still a research vehicle. There are several interesting areas arising out of the language yet to be investigated. The compiler is well structured and reasonably well documented so completing the implementation or modifying the language is a feasible task. Much of the language works at present, making evaluation of its current design also feasible. Some of these possible areas of future development are:

1. Completion of the implementation of the concurrency features. There are several interesting questions involved here, particularly involving how to organize the process control and message passing facilities. There are many chioces of how to partition physical memory, how to provided the shared memory for pools, etc.

2. Investigate the actual usefulness of the language, particularly in applications that have not traditionally used multiprocessing. Evaluation of the sequential aspects of the language, especially the control structures and modules, could be attempted at this time. Evaluation of the concurrent parts depends on completion of them, but could be an exciting area.

3. Examine solutions to some problems faced by all "systems programming" languages. One such issue is how to provide an ablilty to call procedures with varying numbers of parameters without abandoning the "protectionist" philosophy of the language. Another is how to handle type escapes; WISCH

currently defines one approach but there are certainly others. The combination of these two is also interesting, it is the problem faced when one tries to write routines such as PASCAL's READ and WRITE without having them built into the compiler.

4. Use the language as a base for further language development. The current design and implementation could provide a good basis for other experiments in language design. It is hoped that people will feel free to modify the language at will.

M. Introduction to ASP

The primary motivation for the design and implementation of the programming language WISCH was to provide a suitable high-level language for systems programming on the department of Computer Science's Hewlett-Packard 21MX microprogramable minicomputer. Since the architecture of the 21MX is less than pleasant, especially for the implementation of Algol-like languages, it was decided that WISCH would be implemented on a totally different architecture, ignoring the standard 21MX instruction set. The ASP machine (A Stack Processor) is the result of this design effort.

Although designed as a vehicle for WISCH, ASP is a suitable machine for many modern languages. It has been microprogrammed to run on an HP 21MX, E-series computer. It is not a virtual or abstract machine, it is every bit as real as the standard 21MX or any other microprogrammed computer. Because of this commitment to reality, performance constraints and the nature of the underlying hardware have forced compromises into the design. ASP is not claimed to be an "optimal" machine in any sense. It is however a viable architecture, offering easy to generate code, very compact code and reasonable speed.

N. Basic structure of ASP

The machine has a fairly normal stack oriented architecture. Conceptually, memory is divided into three distinct areas, the program, the stack, and the heap. The program area contains the re-entrant program code plus large constants. At any instant, the stack contains a stack frame for each currently activated routine. Stack allocation of variables and recursive routines are the normal mode of operation. The heap is used for dynamic allocation of space under explicit program control.

All memory is composed of 16 bit words and is word addressed. The rightmost (least significant) bit of a word is bit 0. An early version of the machine used byte addressing, but this caused serious performance problems (because of the hardware of 21MX). Some of the terminology used in the rest of this document is a carry over from this version and deals with (8 bit) bytes. But unless otherwise mentioned, all sizes, offsets and addresses in the sequel are in words. Integer arithmetic is two's complement. The exact format of reals is currently undefined, but they are tentatively planned to use 6 bytes of memory. Strings are stored as a one word (current) length followed by "max-length" bytes for the characters in the string. Sets are simply a sequence of words, 16 elements per word. Arrays are just a sequence of elements stored in lexicographic order. The array subscripting instruction

requires a descriptor, but this may be located anywhere in memory and need not be linked to the array.

Physical memory is organized as follows:

```
(high address)

heap (grows down)

                    <-- stack limit

                    <-- workspace top

workspace (grows up)

variables
parameters
linkage area
                    <-- current stack frame
function value (if appropriate)

earlier stack frames

global stack frame
                    <-- stack base

program area

(low address)
```

The code for a routine consists of a "prologue" area followed by the actual body of the routine. The prologue contains:

offset	contents
0	size of parameter and linkage area
1	size of variable area
2	maximum amount of workspace needed
3	size of "constant" area
4-n	"constant" area (large literals)

The first four words must always be present. The constant area may be empty, but by convention the first part of it should be a string literal giving the name of the routine. The linkage area

for each stack frame is a six word area composed of:

offset	contents
0	static link
1	entry point of current routine
2	dynamic link
3	return address
4	index register save
5	address of last stack word needed (end of variables and maximum workspace)

The global (bottom-most) stack frame is a special case. Its must begin with a nine word area organized as follows:

offset	contents
0	address of first word in the stack (a pointer to itself)
1	main entry point of the program
2	address of HP save area
3	address of ASP save area
4	entry point of interrupt interface
5	address of last stack word needed by main program
6	address of highest word available for stack or heap
7	error code mask
8	instruction that caused the error

The use of these words is explained in the section dealing with the actual implementation of ASP.

The machine has eight internal registers. Normally, only the index register is dealt with directly, the others are used implicitly. There are instructions, however, to load or store any of them directly (intended to be used only by debugging and error handling routines). The registers are:

PC - program counter
PS - program status
SB - stack base pointer
SF - stack frame pointer
WP - workspace top pointer
SL - stack limit pointer
IR - index register
CI - current instruction

The program status register is as organized as:

xxxx xxxx EEEE SCCC The "x" bits are currently unused. The "E" bits are the machine error status. Normally they are zero, but if a fatal error occurs, they are set to a code to identify the error. The "S" bit is for a planned single-step mode. The "C" bits are the comparison condition code. They are set by compare instructions and tested by conditional branch instructions. The program counter contains the address of the next instruction to be executed. The stack base register contains the address of the first word in the global stack frame. It allows rapid access to global variables. The stack frame register contains the address of the first word of the current stack frame. The workspace top register contains the address of the top word currently in the workspace. The stack limit register contains the address of the last word available for the stack. It defines the boundary between the heap and the stack, so it changes as the heap grows and shrinks. The index register may be used in operand addressing and loop control. The current instruction register contains the first word of the current instruction. In the event of a fatal error, its contents are stored at offset 8 in the global stack frame. The machine currently recognizes 11 error states. These states and their associated codes are:

0 - no error
1 - illegal instruction
2 - stack overflow
3 - heap overflow
4 - integer arithmetic error
5 - real arithmetic error
6 - (unused)

7 - subscript out of range
8 - range check failure
9 - (unused)
10 - string operation error
11 - assertion failure (used with the "suicide" instruction)
12 - FORK jump error (used with the "suicide" instruction)

Almost all expression evaluation is done using only the workspace for temporary storage. The exceptions are the string and set operations which must use stack or program space set aside for temporaries. All workspace operands are 1, 2, 3 or 4 words long. Commonly used dyadic operators have a one address form. The first operand is taken from the top of the workspace, the second is the addressed operand and the result is placed back on the workspace. (Comparison operations do not return a value to the workspace, they simply set the comparison condition code.) A few common monadic operations are also provided in a one address form. Less commonly used operations are provided in a zero address form with all operands taken from the workspace.

O. Design rationale

The stack orientation with expression evaluation on the workspace was chosen over a register orientation for simplicity of code generation and code compaction. Register machines are more flexible and potentially faster, but are considerably more difficult to generate code for and the resulting code almost certainly will be larger. The mixture of one address and zero address operations was selected over a pure zero address machine for space and time savings. Consider the evaluation of an expression with n variables (n-1 operators). The zero address machine will require n LOAD instructions and n-1 zero address operations. The one address machine will require at most n/2 LOAD's, n/2 one address operations and (n/2)-1 zero address operations (many intermediate results). (The expressions A+B+C+D and (A+B)*(C+D) illustrate these two extremes.) To provide some numbers, assume that LOAD's and one address operations are 16 bits long and zero address operations are 8. Tabulating the above results:

	zero addr.	best one addr.	worst one addr.
# of instr.	2n - 1	n	1.5n - 1
# of bits	24n - 8	16n	20n - 8

For various values of n the following ratios of zero address to one address are obtained:

	# of instr.		# of bits.	
n	best	worst	best	worst
2	1.50	1.50	1.25	1.25
3	1.67	1.43	1.33	1.23
4	1.75	1.40	1.38	1.22
5	1.80	1.38	1.40	1.22
limit	2.00	1.33	1.50	1.20

Further improvements may be realized through an encoding scheme as suggested by Tannenbaum [TAN 78]. However, such schemes require cheap byte addressing to perform well, which is definitely not the case for the hardware that ASP must deal with. All ASP instruction lengths are multiples of 16 bits (so the worst case size above becomes 24n-16). An "optimal" machine should also be designed after an analysis of actual programs in the languages that the machine is to support, clearly a difficult task with WISCH.

P. Instruction repertoire

In the instruction descriptions that follow, classes of instructions are defined and described. The names attached to these classes are simply labels, they are not meant to be definitions. For example, the "Program Relative Instructions" are really a form of one address instruction, but are different in structure from the "One Address Instructions". Following each class description is a list of the instructions in that class. This list gives a short description of the instruction and the basic hexadecimal value of the first word of it (ie. showing the opcode).

P.1. One address instructions

These all have both one and two word forms. The first byte of the instruction is always:

CCCC CCSG

where "CCCC CC" is the "opcode", "S" gives the instruction size (0 is one word) and "G" selects addressing relative to the current or global stack frame (0 is current). For the one word form, byte 2 is:

IX00 0000

where "I" selects indirect addressing (0 is direct), "X" selects indexed addressing (0 is non-indexed) and "00 0000" is the offset (a value from 0 to 63). For the two word form, byte two is:

IX-- LLLL

where "I" and "X" are as above and "LLLL" selects the relative static level (0 is local, 1 is the "father", 2 the "grandfather", etc.). If the "G" bit of byte 1 is set the "LLLL" bits are ignored. The second word in the two word form is the offset as a 16 bit signed integer (ie. it can be negative). Indirection is single level and occurs before indexing.

```
0000  load 2 bytes to top of the workspace
1000   "   4   "   "   "   "   "      "
2000   "   6   "   "   "   "   "      "
3000   "   8   "   "   "   "   "      "
4000   "   the upper byte of a word (as a value from 0 to 255)
5000   "    "  lower  "   "  "  "    "  "   "    "   " "   "
6000   "    "  index register
7000   "    "  effective address to top of the workspace
8000  store 2 bytes from the top of the workspace (pop work)
9000   "   4   "    "    "   "  "  "    "      "       "   "
A000   "   6   "    "    "   "  "  "    "      "       "   "
B000   "   8   "    "    "   "  "  "    "      "       "   "
C000   "   to the upper byte of a word              "   "
D000   "   "   "  lower  "   "  "  "                "   "
E000   "   the index register
F000  zero       a word in the stack
0400  increment "  "  "   "    "
1400  decrement "  "  "   "    "
2400  compare stack operand with zero  (opnd rel zero)
3400  compare index register with stack operand
4400  integer add stack operand to workspace top
5400     "    subtract stack operand from workspace top
6400     "    multiply workspace top by stack operand
7400     "    divide       "      "  "    "      "
8400     "    compare workspace top with stack operand
9400  bitwise AND stack operand to workspace top
A400     "    OR    "      "    "      "      "
B400  real add stack operand to workspace top
C400   "   subtract stack operand from workspace top
D400   "   multiply workspace top by stack operand
E400   "   divide       "      "  "    "      "
F400   "   compare workspace top with stack operand
0800  save 2 bytes from workspace to stack (no pop)
1800   "   4   "    "       "     "    "    "   "
2800   "   6   "    "       "     "    "    "   "
3800   "   8   "    "       "     "    "    "   "
4800   "   to the upper byte of a word
```

5800 " " " lower " " " "

P.2. Immediate operand instructions

These also have one and two word forms. The first byte is always:

CCCC 10S-

where "CCCC" is the opcode and "S" gives the instruction size as before. For the 1 word form the lower 9 bits of the word are the operand, treated as a 9 bit signed integer (-256 to +255). For the two word form the second word contains the value as a 16 bit signed integer while the lower 9 bits of the first word are ignored.

```
6800  load immediate to top of workspace
7800   "       "     "  index register
8800  add      "     "    "      "
9800   "       "     "  top of workspace
A800  multiply top of workspace by immediate
B800  divide    "  "      "     "      "
C800  bitwise AND immediate to top of workspace
D800     "    OR      "     "   "  "      "
E800  compare index register with immediate
F800  compare top of workspace with immediate
```

P.3. Jump instructions

The jump instructions all use addressing relative to the first word past the jump itself. They have both one and two word forms. The first byte is always:

SXXX 110-

where "XXX" is the opcode and "S" is the instruction size. For the one word form the lower 9 bits are treated as a 9 bit signed integer (-256 to +255). For the two word form the lower 9 bits of the first word are ignored; the second word contains the jump offset as a 16 bit signed integer. Thus, for a one word jump an offset of -1 is an infinite loop, while for a two word jump an offset of -2 gives an infinite loop.

```
0C00  jump unconditionally
1C00   "   on equal
2C00   "   "  greater
3C00   "   "  greater or equal
4C00   "   "  less
5C00   "   "  less or equal
6C00   "   "  not equal  (less or greater)
7C00   "   via FORK table
```

The FORK jump needs further explanation. It adds the jump offset to the top of the workspace (which is then popped) and then to the program counter. This value at the location thus referenced is then added in to give the final jump address. In higher level terms, the jump offset is the offset to a branch table. The top of the workspace contains an index into this table. The table entries are offsets from themselves to a final destination.

P.4. Program relative instructions

These instructions use addressing relative to the entry point of a routine instead of relative to a stack frame. Their primary intent is to provide a simple access mechanism to large constants stored in the program area. They have both one and two word forms. The first byte is always:

SCCC 1110

where "CCC" is the opcode and "S" gives the instruction size. The rest of the instruction is identical to the one address instructions.

0E00	load	2 bytes from the program area
1E00	"	6 " " " " "
2E00	"	the effective address
3E00	real	add program operand to top of workspace
4E00	"	subtract program operand from top of workspace
5E00	"	multiply top of workspace by program operand
6E00	"	divide " " " " " "
7E00	"	compare " " " to " "

P.5. Memory block instructions

These are instructions which deal with contiguous "chunks" of memory. Except for the workspace pointer increment, they all require two absolute addresses on the top of the workspace. All of them have both one and two word forms. The first byte is always:

CCCS 1111

where "CCC" is the opcode and "S" is the instruction size. For

the one word case, the second byte is the operand length in words as a value from 0 to 255. For the two word case the bottom bit of the first word selects a "dynamic" mode. If it is clear, the second word contains the operand length as a signed 16 bit integer. Obviously, negative values are nonsense for all but the workspace pointer increment instruction. If bit 0 is set, the second word of the instruction is omitted and the length is taken from the top of the workspace (above the addresses).

0F00 increment the workspace pointer register by the operand length; no addresses are used
2F00 compare words as integers; the top address is the right comparand, the bottom address is the left one; the comparison stops at the first pair of differing words
4F00 move a block of words; the top address is the "to" address and the bottom is the "from" address; the move goes from low address to high address
6F00 swap two blocks of words; the swap goes from low address to high address

P.6. Zero address load, store & save

These instructions are zero address analogues of the one address instructions described above. They are all one word long. The bottom two bits of the word are "I" and "X" bits as for the one address instructions (bit 1 is "I" and bit 0 is "X").

8F00 load 2 bytes to top of the workspace
8F10 " 4 " " " " " "
8F20 " 6 " " " " " "
8F30 " 8 " " " " " "
8F40 " the upper byte of a word (as a value from 0 to 255)
8F50 " " lower " " " " " " " " " " " "

```
8F60  store 2 bytes from the top of the workspace (pop work)
8F70   "   4   "    "    "   "  "   "      "       "   "
8F80   "   6   "    "    "   "  "   "      "       "   "
8F90   "   8   "    "    "   "  "   "      "       "   "
8FA0   "   to the upper byte of a word                "   "
8FB0   "   "   "  lower  "   "  "  "                  "   "
8FC0  save 2 bytes from workspace to stack (no pop)
8FD0   "   4   "    "       "     "    "    "   "
8FE0   "   6   "    "       "     "    "    "   "
8FF0   "   8   "    "       "     "    "    "   "
9F00   "   to the upper byte of a word
9F10   "   "   "  lower  "   "  "  "
9F20   "    "  effective address to top of the workspace
```

P.7. Machine register access

This group of instructions allows the workspace top to be loaded from or stored to any of the eight machine registers. To keep the terminology consistent with the other instruction descriptions, note that the word "load" means to transfer a value from a register to the top of the workspace and the word store means to transfer a value from the top of the workspace to a register. Except for the index register transfers, these instructions are provided for test and debugging purposes. They are not intended for general use. All are one word long.

```
9F30  store to the status register
9F31   "   "   "  program counter
9F32   "   "   "  stack base pointer
9F33   "   "   "  stack frame pointer
9F34   "   "   "  workspace top pointer
9F35   "   "   "  stack limit pointer
9F36   "   "   "  index register
9F37   "   "   "  current instruction register
9F38  load from the status register
9F39   "   "    "  program counter
9F3A   "   "    "  stack base pointer
9F3B   "   "    "  stack frame pointer
9F3C   "   "    "  workspace pointer
```

9F3D	"	"	"	stack limit pointer
9F3E	"	"	"	index register
9F3F	"	"	"	current instruction register

P.8. Shift instructions

These three insructions operate on the workspace top and have the shift count and direction encoded in the instruction. The direction is in bit 4, 0 is left and 1 is right. The bottom four bits have the shift count as a value from 0 to 15.

9F40	logical shift (end-off, zero-fill)
9F60	arithmetic shift
9F80	rotation

P.9. Word operations

These instructions are the zero address operations for single word operands. They are all one word long. The shift instructions presented here have the count on the workspace above the value to be shifted. A positive count indicates a left shift, a negative count is a right shift. If the absolute value of the count is larger than 16 it is assumed that 16 is meant.

9FA0	integer	addition
9FA1	"	subtraction
9FA2	"	multiplication
9FA3	"	division
9FA4	"	modulus (see the WISCH report for the)
9FA5	"	remainder (difference between these two)
9FA6	"	absolute value

9FA7	"	negation
9FA8	"	comparison (pop both operands)
9FA9	"	comparison (pop only the left operand)
9FAA	bitwise	AND
9FAB	"	inclusive OR
9FA	"	exclusive OR
9FAD	"	complement
9FAE	"	logical shift
9FAF	"	arithmetic shift
9FB0	"	rotation

P.10. Real operations

Although their format is undefined, opcodes a have been allocated for zero address instructions to operate on real values. They are all one word long.

9FC0	real	addition
9FC	"	subtraction
9FC	"	multiplication
9FC	"	division
9FC	"	absolute value
9FC	"	negation
9FC	"	compare
9FC	"	convert integer to real
9FC	"	" real to integer

P.11. Other instructions

There are many other miscellaneous instructions in the ASP machine repertoire. They are collected together here and given individual descriptions below.

9FF0 array subscripting

This instruction is one word long. The bottom four bits

are "CDDD", where "C" is a checking bit and "DDD" is the number of dimensions in the array less one. If the "C" bit is set each index is checked to make sure that it is within its proper range, no checking at all is done if it is clear. This instruction requires n+2 arguments on the workspace for an n dimensional array. The top of the workspace must be a pointer to a descriptor block. This block contains 2n+1 words. The first n pairs are the lower bound and range (upper-lower+1) for each dimension. The lower bound is the first word in the pair. The last word in the descriptor is the size of a single array element. The next n elements of the workspace must be the array indices, with the n'th on top. The bottom argument is the address of the start of the array. The instruction leaves the address of the selected element on the top of the workspace.

AF00 normal routine call

This instruction is two words long. The bottom four bits of the first word give the relative static level of the called routine, 0 is a son of the caller, 1 is a brother, 15 is a "global" level routine. The second word of the instruction is the absolute address of the routine entry. This instruction assumes that all parameters (including the 6 word linkage area) are on the workspace. It uses the word at offset 0 of the routine prologue to decide how far the start of the new stack frame should be below the current top of the workspace. The workspace pointer is

then incremented by the size of the variable area and a check is made to make sure that there is enough workspace for the routine.

AF10 routine return

This instruction is one word long. It simply causes a return from the current routine.

AF20 dynamic routine call

This instruction is one word long. Its bottom four bits contain the relative level of the called routine as defined above. The entry point of the called routine is on the top of the workspace instead of inline in the program space.

AF30 duplicate top of workspace

This one word instruction duplicates values on the top of the workspace. Its bottom two bits contain the number of words to duplicate less one.

AF80 range check

This instruction checks that the top word of the workspace (treated as a signed integer) is within a specified range, causing a fatal error if it isn't. It does not pop the value being cheked. The instruction has one, two and three word forms. Its bottom four bits are "xxCC", where "SS" selects the size. Opcode AF80 is a two word instruction, the second word containing the range bounds. Both bounds must be between 0 and 255. The upper byte of the second word is the upper bound, the lower byte is the lower bound. Opcode AF81 is a three word instruction.

The second word is the lower bound and the third is the upper. Both bounds are treated as 16 bit signed integers. Opcode AF83 is a one word instruction, the bound are on the workspace. The top value is the upper bound, the next is the lower.

AF40 full exit from ASP

This one word instruction requires two arguments on the workspace. The top must be a pointer to an 8 word save area for the ASP internal registers. The next must be a pointer to a 4 word save area for the HP registers P, S, X, and Y. First the ASP registers PC, PS, SB, SF, WP, SL, IR and CI are stored in the ASP save area. Next the HP registers are loaded from their save area. Finally the HP A register is loaded with the pointer to the ASP save area, the B with the HP pointer and control is returned to the HP instruction set.

AF50 inline exit from ASP

This instruction is one word long. It allows ASP and HP instructions to be intermingled in a single routine. It pushes the ASP registers SL and IR on the workspace (IR on top), puts the ASP PS into the HP A, the ASP WP into the HP B (actually B points to SL, ie. it is WP-1), the ASP SB into the HP Y and the ASP SF into the HP X. It then simply exits to the HP instruction interpreter so that it will begin execution of the next instruction in sequence.

AF60 suicide exit from ASP

The one word suicide instruction is intended to be used

for debugging and to programably cause fatal errors. Its bottom four bits are placed in the "EEEE" bits of the PS register and an error exit from ASP is made.

AF70 interrupt exit from ASP

This is a one word instruction, provided to allow testing of the ASP-HP interrupt interface. The function of this interface is described below in the section on the implementation of ASP.

Q. Implementation of ASP

As mentioned previously, the ASP machine has been implemented as a microprogram for a Hewlett-Packard 21MX E-series computer. It allows ASP programs to be run under the RTE-2 operating system.

To start an ASP program the machine must be loaded into the HP's WCS memory, the base of the global stack frame must be set up and the ASP registers must be given initial values. Words 0 and 1 of the global stack frame are self explanatory. Words 2 and 3 are left alone. Word 4 points to the start of an interrupt interface routine. The structure of this routine is described below. Word 5 has the same purpose as word 5 of a normal routine frame; it is used for checking for stack overflow. Word 6 is also self explanatory, the initial value of the SL register should be the same. Word 7 is a bit mask for fatal errors that are to be ignored. Bit i masks out error code i. Word 8 is left alone. An 8 word save area must be set up with the initial values for the ASP registers PC, PS, SB, SF, WP, SL, IR, and CI. The HP A register is loaded with the address of this save area and the HP B register with the address of a 4 word HP save area. This HP save area will get the values of the HP registers P, S, X and Y when ASP is started. Execution of the HP instruction with octal code 101611 will then cause the ASP machine to be started. The save area pointers will be stored in global locations 2 and 3 automatically. If a

fatal error is deteced by ASP, a full exit is done automatically using these save areas.

Since ASP has no I/O instructions it is necessary to be able to exit to the HP instruction set to perform I/O and then to re-enter ASP. Control must also be automatically surrendered to the HP instruction set whenever an interrupt is detected. The former capability is provided by the inline exit and entry instructions. The ASP inline exit lets routines be schizophrenic and is described above. The HP instruction with octal code 101612 accomplishes the inverse operation. When an interrupt request is detected while ASP is running, a special quick exit is made to the HP instruction set to allow it to be handled. The PS, SL and IR registers are pushed onto the workspace (IR on top), the PC is put into the HP A register and WP into the HP B register (B actually points to the PS value, ie. it is WP-2). The ASP SP and SB registers happen to be the same hardware registers as the HP X and Y, so they are left alone. The HP P register is then loaded with the interrupt entry point and control is given to the HP instruction set. Typically, the interface routine is just a few HP NOP instructions (to allow the interrupt to be caught) followed by an interrupt return to ASP, HP instruction 101613.

R. Future work with ASP

Like WISCH, ASP is both a working tool and a research vehicle. There are two main areas of future work with ASP that seem well suited to M.Sc. research.

An excellent project would be to take the ASP machine and revise and polish it into a better kernel machine for a variety of languages, especially languages of local interest to UBC such as BCPL, C, PASCAL and ZED. Properly designed, it could even allow inter-lingual routine calls with no overhead. This project could benefit from the large body of existing programs in these languages to analyze what language features are actually used. This new machine could also be extended to contain instructions forI/O and memory management, making it a "real" computer in the fullest sense. Such a project would place the design of computer architecture where it really belongs, subservient to the acutal needs of programmers. The machine produced would likely be of considerable practical value.

Another project, which fits well with the first, would be to study how to best provide more sophisticated, language specific instructions. For WISCH this might include instructions for string manipulation, set operations, heap maintainance, message passing and process control. These instructions could be designed as optional extentions to the

kernel machine mentioned above.

1. Atwood, J. W., et. al. Project SUE status report. Technical report CSRG-11, Computer Systems Research Group, University of Toronto (April 1972).

2. Brinch Hansen, P. Structured multiprogramming. Communications of the ACM, Vol. 15, No. 7 (July 1972), pp. 574-578.

3. Brinch Hansen, P. The programming language Concurrent Pascal. IEEE Transactions on Software Engineering, Vol. 1, No. 2 (June 1975), pp. 199-207.

4. Brinch Hansen, P. Experience with modular concurrent programming. IEEE Transactions on Software Engineering, Vol. SE-3, No. 2 (March 1977), pp. 156-159. Corbato, F. J. PL/1 as a tool for system programming. Datamation, Vol. 15, No. 5 (May 1969), pp. 68-76.

5. Hartmann, A. A Concurrent Pascal Compiler for Minicomputers. Springer-Verlag, Berlin, (1977).

6. Hoare, C. A. R. Towards a theory of parallel programming. Operating Systems Techniques, ed. C. A. R. Hoare and R. H. Perrot. Academic Press, New York (1972).

7. Hoare, C. A. R. Monitors: An operating system structuring concept. Communications of the ACM, Vol. 17, No. 10 (October 1974), pp. 549-557.

8. Hoare, C. A. R. Parallel Programming: An axiomatic approach. Computer Languages, Vol. 1 (1975), pp. 151-160.

9. Jensen, K. and Wirth, N. PASCAL User Manual and Report. Springer-Verlag, Berlin (1974).

10. Kahn, K. A small-scall operating system foundation for microprocessor applications. Proceedings of the IEEE, Vol. 66, No. 2 (February 1978), pp. 209-215.

11. Lampson, B. W., et. al. Report on the programming language EUCLID. Sigplan Notices, Vol. 12, No. 2 (February 1977).

12. Liskov, B. The design of the Venus operating system. Communications of the ACM, Vol. 15, No. 3 (March 1972), pp. 144-149.

13. Randell, B., ed. The Origins of Digital Computers. Springer-Verlag, Berlin (1973).

14. Richard, F. and Ledgard, H. F. A reminder for language designers. Sigplan Notices, Vol. 12, No. 12 (December 1977), pp. 73-82.

15. Silberschatz, A., et. al. Extending Concurrent Pascal to allow dynamic resource management. IEEE Transactions on Software Engineering, Vol. SE-3, No. 3 (May 1977), pp. 210-217.

16. Sorensen, S. M. and Stanstrup, J. PLATON reference manual. Report R-75-58, Regional EDP Center, University of Aarhus (July 1975).

17. Sorensen, S. M. and Stanstrup, J. PLATON - Design and implementation on RC 3500. Report RECAU-77-80-R, Regional EDP Center, University of Aarhus (May 1977).

18. Stanstrup, J., and Sorensen, S. M. PLATON - A high level language for systems programming. Report R-75-59, Regional EDP Center, University of Aarhus (May 1975).

19. Stoy, J. E. and Strachey, C. OS6 - An experimental operating system for a small computer. Part 1: General principles and structure. Computer Journal, Vol. 15, No.2 (April 1972), pp. 117-124.

20. Tannenbaum, A. Implications of structured programming for machine architecture. Communications of the ACM, Vol. 21, No. 3 (March 1978), pp. 237-246.

21. Whitaker, W. A. The U. S. Department of Defense common high order language effort. Sigplan Notices, Vol. 13, No. 2 (February 1978), pp. 19-29.

22. Wirth, N. MODULA: A language for modular multiprogramming. Eidgenossische Technische Hochschule, Zurich (March 1976).

23. Wirth, N. The use of MODULA and Design and implementation of MODULA. Eidgenossische Technische Hochschule, Zurich (July 1976).

24. Yasumura, M. Evolution of loop statements. Sigplan Notices, Vol. 12, No. 9 (September 1977), pp. 124-129.

The word WISCH comes from the name WIhelm SCHickard, a German astromomer who built a calculating machine in the early 1620's. Unfortunately the machine was destroyed by fire not long after it was finished. Schickard died during the Plague and no copies of his machine were ever made. Evidence of its existence comes from a description and drawings sent to Johannes Kepler in 1624 (at which time Blaise Pascal was one year old).