YOUNG
52 CLERMISTON RD. NORTH
EDINBURGH 4
Tel: (031) 336 6710

SKIMP MK II

D. J. Rees

Introduction

SKIMP is a very simple language designed with the teaching of compilers in mind. Although simple, it possesses many of the structural features of full-scale languages, for example the subroutine and parameter structure, making it a worthwhile non-trivial vehicle for teaching. Many useful features of full-scale languages are missing but can be incorporated with very little difficulty. This is quite intentional. The reasons are firstly that SKIMP is a sufficiently rich language for the compiler to be written in SKIMP itself with little inconvenience (the SKIMP MKI compiler was written in SKIMP) and secondly that incorporation of the missing features provides a very suitable form of student exercise. Although it would be desirable for students to write a complete compiler of their own, this is a considerable task which they would probably not have the time to complete. The task of extending a simple language such as SKIMP for which a compiler already exists is much more feasible and still allows students to get to grips with the compiling process and so understand it more fully.

The name SKIMP arises from the language's origin. Essentially it is a much smaller version of the language IMP, an ALGOL-60 type language developed in Edinburgh for systems implementation and general use.

Since SKIMP is intended for teaching rather than production programming, it has not been necessary for the compiler to produce object code for a particular real machine. Instead, the opportunity has been taken to invent a machine whose order code does not have the inconsistencies and exceptional cases that all actual machines seem to have. The compiler produces symbolic code for this invented machine together with a certain amount of optional monitoring of the compilation.

This document describes the SKIMP language, the object machine and details of the method of compilation. Examples of compiler output and suggested extensions to SKIMP suitable for student exercises are given in appendices. To enable students to test their modified compilers an assembler/interpreter for the object code is available. The use of this is also described in an appendix.

Chapter 1

The SKIMP Language

A SKIMP program consists of a sequence of keywords, identifiers and constants together with arithmetical operators and various separator characters.

Keywords

A keyword consists of a sequence of upper or lower case letters preceded by the character '%'. For example:

%ROUTINE %END %THEN

The '%' character is regarded as a shift character in that all following letters up to the first non-letter are shifted to keyword mode. This implies that no spaces or other non-letter characters may be inserted into the middle of a keyword. The '%' can, however, be repeated after a space. Thus,

%INTEGERNAME and %INTEGER %NAME

are valid and equivalent, but

%INTEGER NAME

is invalid.

Identifiers

An identifier consists of a string of letters and digits of any length the first character of which must be a letter. For example,

SKIMP I J2 X1A

In descriptions of SKIMP syntax below, the phrase <NAME> is used to denote the presence of an identifier.

Constants

Constants appear as operands in arithmetic expressions and may be of two forms, decimal and character. Both represent integer valued quantities since SKIMP only has integer valued variables and performs only integer arithmetic. A decimal constant is represented by a sequence of digits. For example:

123 98765 4

No decimal points or powers of 10 are allowed.

Strings of up to four characters can be represented by surrounding the characters with quotes. For example:

'FRED' '123*' 'X'

The internal value is formed by packing the (ASCII) values of the characters right-justified in 8-bit fields of a word. A quote can become one of the characters of the value by repeating it. For example:

```
'JIM''S'        ''''
```

Input Lexical Processing

Spaces are deleted on input everywhere except when they appear between quotes. Before deletion, however, they are significant in terminating the keyword '%' shift mode. Text between quotes remains intact with the exception that a repeated quote is regarded as a single quote. Newlines can therefore appear between quotes. For example, to test for the value of a newline character:

```
%IF  I = '
'  %THEN  %RETURN
```

This is equivalent to:

```
%IF  I = 10  %THEN  %RETURN
```

since the ASCII value of a newline character is 10.

SKIMP statements are separated by either ';' or a newline. Thus several statements can appear on one line of text. For example:

```
X = 1  ;  Y = 2  ;  Z = 3
```

In order to represent a single statement on more than one line intermediate newline characters can be inserted when preceded by the continuation keyword %C. For example:

```
%IF  X = 1  %THEN  %C
A = B
```

This is equivalent to:

```
%IF  X = 1  %THEN  A = B
```

Program Structure

A SKIMP program starts with the statement:

```
%BEGIN
```

and ends with the statement:

```
%ENDOFPROGRAM
```

Inside the program, routines and functions may appear in sequence or nested as required. A routine takes the form:

```
%ROUTINE  RT
.....
%END
```

and a function takes the form:

```
%INTEGERFN  FN
.....
%END
```

The only difference between routines and functions is that functions produce a value as their result and routines do not. The structure of a program could take the following form, for example:

```
%BEGIN
   %ROUTINE  A
   .....
   %END
   %INTEGERFN  B
      %ROUTINE  C
      .....
      %END
   .....
   %END
.....
%ENDOFPROGRAM
```

Details of routines and functions and their parameters are given later.

Declarations

Names of variables and arrays denoting storage objects are declared at the start of a routine or function. Execution of declaration statements has the affect of setting aside storage for those variables or arrays. When the routine or function in which the declaration occurred is returned from, that storage is deleted so that it may be re-used for future declarations. When routines and functions are entered recursively new storage is set aside for the variables and arrays declared there without the storage for the old incarnation of the routine or functions being lost. This becomes accessible again when the recursive activation is left. A stack is used to accomplish this. The only objects SKIMP manipulates are integer valued.

All names must be declared before they can be used. Scalar declarations take the form:

%INTEGER <NAME>, <NAME>, <NAME>, . . .

Arrays are declared:

%INTEGERARRAY <NAME>, <NAME>, . . .(<EXPR> : <EXPR>)

where the phrase <EXPR> is used to denote the presence of an arithmetical expression. For example:

```
%INTEGER  I, J, K
```

sets aside three locations and names them 'I', 'J' and 'K'. The expressions in

the array declaration are the lower and upper bounds on the index of the arrays. They may be either constants, for example:

```
%INTEGERARRAY A ( 1 : 10 )
```

or expressions which are evaluated at run-time, for example:

```
%INTEGERARRAY B, C ( -1 : N+1 )
```

Only one-dimensional arrays are defined in SKIMP (but see possible extensions in Appendix 2).

Note that the first statement of a routine or function has the effect of declaring its name. Hence, this must appear before any calls on the names. This allows self-recursive calls but not cross-recursive calls between two routines at the same nesting level. (SKIMP does not have routine and function SPECs as in IMP).

Scope of Variables.

When a variable or array is declared, that name can be used anywhere in the routine or function in which the declaration appears, but not outside it. If the routine or function has nested routines or functions the name can also be used in these (as 'global' to them). For example:

```
%ROUTINE  A
%INTEGER I
   %ROUTINE B
   %INTEGER J
   ....
   %END
   %ROUTINE C
   %INTEGER K
      %ROUTINE D
      ....
      %END
   ....
   %END
....
%END
```

The name 'I' can be used anywhere in routines 'A', 'B', 'C' and 'D'. The name 'J' can be used only in routine 'B' and the name 'K' can be used in routines 'C' and 'D'.

Although, clearly, all the names of variables declared in a single routine or function must be distinct, the same names may be declared in different routines to refer to different storage objects. When a name is redeclared in a nested routine or function, the storage for the global object of the same name becomes inaccessible whilst within that routine or function. The name will always refer to the most 'local' declaration.

Arithmetic Expressions

An arithmetic expression consists of a sequence of operands separated by operators, possibly preceded by a unary operator (+, -, \) . Operands can be

array elements, constants, calls on functions or bracketted sub-expressions. The operators are:

+	:	addition
-	:	subtraction
*	:	multiplication
/	:	division (rounded)
**	:	exponentiation
<<	:	logical shift left
>>	:	logical shift right
&	:	logical AND
!	:	logical OR
!!	:	logical Exclusive OR
\	:	logical NOT

Their precedences are:

		\		:	highest
	**	<<	>>		
	*	/	&		
+	-	!	!!	:	lowest

The precedence is left to right for operators of equal precedence. Thus:

```
I / J * K
```

is equivalent to:

```
( I / J ) * K
```

Array element operands take the form:

```
<NAME> ( <EXPR> )
```

For example:

```
A ( I )
B ( 2 * A ( J ) )
```

Assignment

Variables and array elements are assigned new values using statements of the form :

```
<NAME> = <EXPR>
<NAME> ( <EXPR> ) = <EXPR>
```

For example:

```
I = 2
A ( I * J ) = ( K + L ) / 2
```

Routines and Functions

A routine is called by executing a statement consisting of the name of the routine. For example:

```
%ROUTINE  RT
....
%END
```

would be called by the statement:

```
RT
```

Similarly, the call of a function appears as an operand of an expression. For example:

```
%INTEGERFN  FN
....
%END
```

might be called by:

```
I = FN
```

Routines and functions can be called recursively to any depth. The exit from a routine is specified by the statement:

```
%RETURN
```

This is also implied by executing the %END statement of a routine. The exit from a function must give in addition a result for the function:

```
%RESULT = <EXPR>
```

For example:

```
%RESULT = I * J - K
```

Execution of the %END statement of a function is invalid. A routine or function can only be entered by an appropriate call as described above. If a routine or function is encountered in the normal flow of control of a program at run-time it is skipped around.

Routine and Function Parameters

Parameters of the following types can be passed to routines and functions:

```
%INTEGER
%INTEGERNAME
%INTEGERARRAYNAME
```

The first is a 'value' type parameter and the other two are 'reference' types. For the %INTEGER type, the value of an expression in the routine or function call is calculated and passed to the routine or function as the initial value of the parameter. The initialised parameter can be regarded thereafter as a local

variable. For the %INTEGERNAME type, a reference to some storage object outside the routine or function is passed and this object will be referred to whenever the parameter is referenced within the routine or function. Hence the only admissible forms of parameter in the call are either the name of a variable or the name of an array element. Similarly for the %INTEGERARRAYNAME type, a reference to an array is passed and the only admissible form of 'actual' parameter is the name of an array. For example:

```
%INTEGER A,B,C
%INTEGERARRAY M,N (1:10)
%ROUTINE JIM(%INTEGER I,%INTEGERNAME J,%INTEGERARRAYNAME  K)
....
%END
JIM (A+B,C,M)
JIM (A,M(B),N)
```

Jumps

SKIMP labels are numerical. For example:

123: 987:

To jump to these labels the instructions would be:

->123 ->987

Labels are local to a routine or function and jumps can only take place within the routine or function and not either to an outer textual level or to a nested inner level. Labels are not declared and jumps may precede or follow the occurrence of the corresponding label without restriction.

Conditional Statements

The general form of the conditional statement is:

%IF <COND> %THEN <INSTR> %ELSE <INSTR>

where <COND> represents the condition to be tested and <INSTR> represents a restricted class of instructions described below. If the condition is true the first of these instructions is executed, otherwise the second. The %ELSE clause can be omitted in which case if the condition is false execution proceeds to the following statement.

A <COND> is formed by one or more simple relations of the form:

<EXPR> <COMP> <EXPR>

where <COMP> represents one of the following comparators:

= # <= < >= >

For example:

I < J + K

These simple relations can be combined using the keywords %AND and %OR. For

example:

```
I = J  %AND  K = 0
```

No precedence is defined between %AND and %OR. Instead, brackets must be used. A single level of bracketting may only contain a sequence of simple comparisons connected with either %AND or %OR but not both. For example:

```
I = 0  %AND  ( J = 0  %OR  K = 0  %OR  L = 0 )
```

The relations are evaluated left to right but only up to the point where the truth or falsity of the whole condition can be determined.

An <INSTR> may be any of the following types of statement:

```
assignment
routine call
jump
%RETURN
%RESULT=<EXPR>
%STOP
```

For example:

```
%IF  I = J  %THEN  %RETURN
%IF  K > 0  %THEN  K = 0  %ELSE  K = -1
```

In addition, groups of statements can take the place of an <INSTR> and be made conditional. The grouping is made by preceding the first statement by the statement:

```
%START
```

and following the last one by the statement:

```
%FINISH
```

For example:

```
%IF J = K %THEN L = 0 %ELSE %START
  %IF J < K %THEN %START
    L = 1
    M = 2
  %FINISH %ELSE %START
    L = 2
    M = 1
  %FINISH
%FINISH
```

%STARTs and %FINISHes may be nested to any depth but must balance within any routine or function.

Comments

Comment statements start with the exclamation character '!'. Thereafter, all text is ignored up to the next statement separator i.e. semi-colon or newline. For example:

```
                    ! this is a comment
              I = 0     ;! so is this ;   J = 0
```

Input-Output

The standard IMP set of input-output routine names are built into SKIMP. These are:

```
%ROUTINE READ SYMBOL(%INTEGERNAME X)
  Read a single symbol from the current input stream into X.

%INTEGERFN NEXT SYMBOL
  Return as the result the next single symbol which will be read.

%ROUTINE SKIP SYMBOL
  Skip over the next single symbol in the input.

%ROUTINE PRINT SYMBOL(%INTEGER X)
  Output the single symbol X.

%ROUTINE SPACE
  Output a space symbol.

%ROUTINE SPACES(%INTEGER X)
  Output X spaces.

%ROUTINE NEWLINE
  Output a newline symbol.

%ROUTINE NEWLINES(%INTEGER X)
  Output X newlines.

%ROUTINE NEWPAGE
  Output a newpage symbol.

%ROUTINE READ(%INTEGERNAME X)
  Read a decimal number into X skipping preceding spaces and newlines.

%ROUTINE WRITE(%INTEGER X,Y)
  Output the decimal number X of Y digits.
```

Chapter 2

The Object Machine.

The SKIMP compiler produces object code output for an invented machine. The output takes the form of what would be this machine's assembly code rather than a binary form so that it can easily be inspected by students. Since the machine is only an invented one and does not exist, the object code cannot be run in the ordinary way, but an assembler/interpreter for this code is available which allows the output to be tested.

No attempt has been made to define the machine completely. Only those features necessary and relevant for the compiler are defined. Interrupt structure and input-output instructions, for example, are not defined.

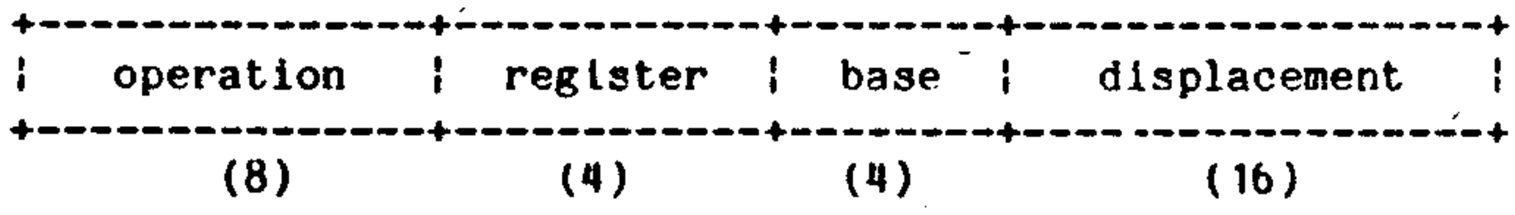
The machine has a number of registers which can either be used as accumulators or as index registers. In this respect, it is similar to machines such as IBM 370, DEC 10 etc.. It is not a true multi-accumulator machine, however, in that no register to register arithmetic operations are defined. The instruction format is shown below:

operation , register , base , displacement

For example:

LOAD, ACC, DR1, 2

Instructions operate between the register and an effective storage address formed from the sum of the contents of the base register and the displacement. The machine is word-addressed and assumed to have sufficient storage and registers for the purposes of the compiler. Although not strictly necessary, it is also useful to define a word size and the instruction word layout. A 32-bit word is envisaged with the layout:



This provides for 16 registers and a displacement of up to 65535. All sixteen registers, 0-15, can be used in the register field of the instruction, but only registers 1-15 in the base field allowing a zero in the base field to indicate no indexing in the effective address.

The compiler object code output does not in practice use register numbers. Instead, it uses register mnemonics for convenience of inspection. The mnemonics used by the compiler are:

ACC, STP, COT, WK, DR0, DR1, DR2, DR3,.....

Their individual use is explained in later chapters.

The operations defined for this machine are:

LOAD : Load the register with the contents of the word at the effective storage address.

LDA : Load the register with the effective storage address.

ADD : Add the contents of the word at the effective storage address to the register.

SUB : Subtract the contents of the word at the effective storage address from the register.

MLT : Multiply the register by the contents of the word at the effective storage address.

DIV : Divide the register by the contents of the word at the effective storage address.

EXP : Raise the register to the power given by the contents of the word at the effective storage address.

STR : Store the contents of the register in the location at the effective storage address.

NEG : Negate the contents of the register (the effective address is ignored).

NOT : Perform the logical NOT on the register (the effective address is ignored).

SHL : Shift the register logically left by the amount given by the contents of the word at the effective storage address (zeros shifted in from the right, bits shifted out to the left lost).

SHR : Shift the register logically right by the amount given by the contents of the word at the effective storage address (zeros shifted in from the right, bits shifted out to the right lost).

AND : Perform the logical AND between the register and the contents of the word at the effective storage address.

OR : Perform the logical OR between the register and the contents of the word at the effective storage address.

XOR : Perform the logical Exclusive OR between the register and the contents of the word at the effective storage address.

BAL : Unconditional branch to the effective address and the address of the next instruction stored in the specified register.

B : Unconditional branch to the effective address (any register present is ignored).

BZ : Branch to the effective address if the contents of the register are zero.

BNZ : Branch to the effective address if the contents of the register are not zero.

BG : Branch to the effective address if the contents of the register are greater than zero.

BNG : Branch to the effective address if the contents of the register are not greater than i.e. less than or equal to, zero.

BL : Branch to the effective address if the contents of the register are less than zero.

BNL : Branch to the effective address if the contents of the register are not less than i.e. greater than or equal to, zero.

STOP : Stop execution.

In addition to these instructions, the compiler also outputs an assembler directive 'FILL' in the same instruction format. This is used to direct the assembler to fill in previously assembled holes in the object code with appropriate values. This is intended primarily for dealing with forward jumps and is described in detail in later chapters.

Chapter 3

The Structure of the SKIMP Compiler

The SKIMP compiler consists of a set of external routines written in IMP. It is a 1-pass system in that the compiler makes one pass over the source text and generates the object code statement by statement. Each statement is processed separately and the processing consists of four phases. These are:

1 : line reconstruction
2 : lexical analysis
3 : syntax analysis
4 : code generation

The flow of control through these phases is shown in figure 3.1.

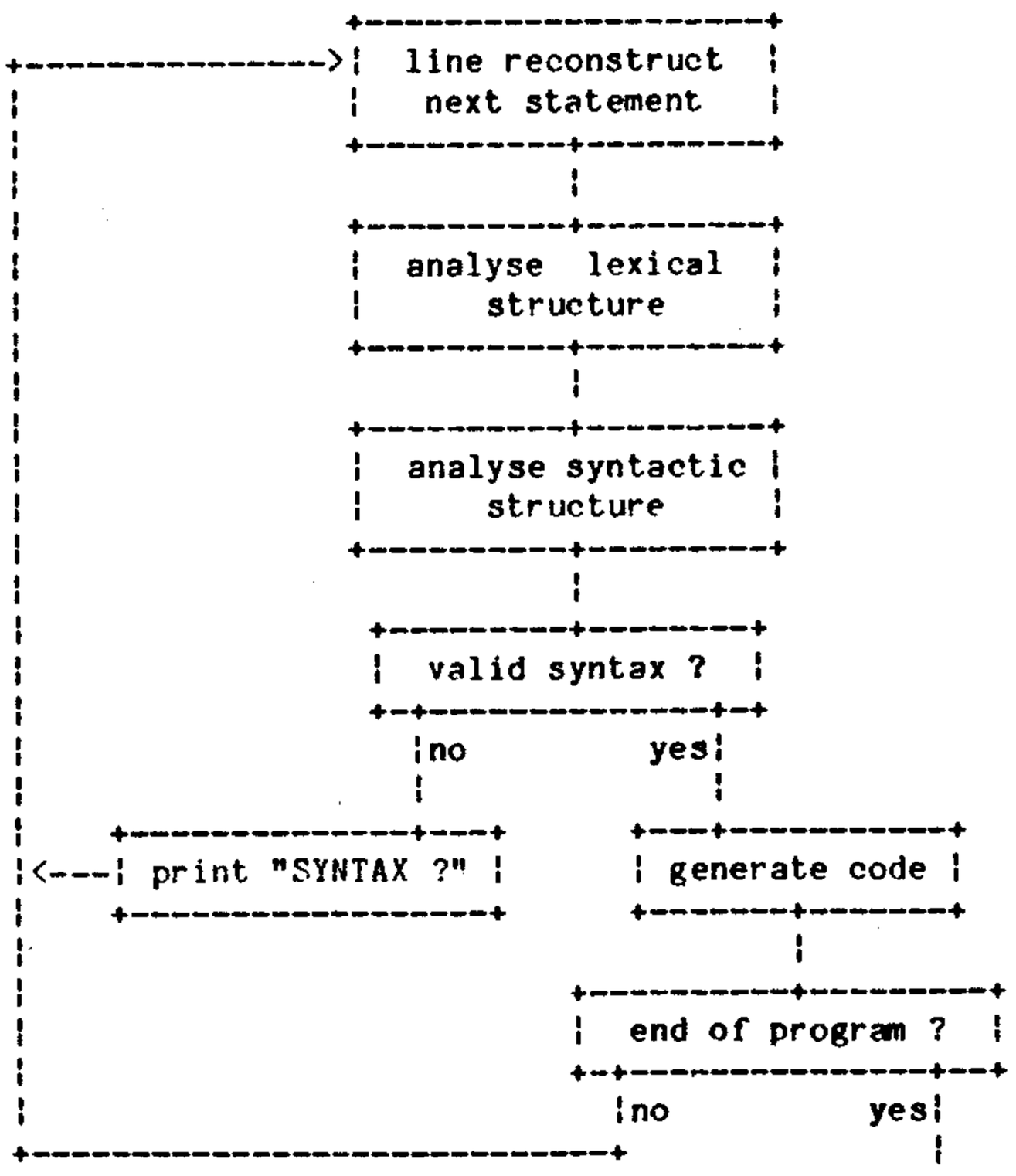


figure 3.1

Syntax Definitions

For convenience when continually modifying and developing the compiler, a description of the syntax of statements is read in from a file each time a program is compiled. The syntax takes the form of phrase structure definitions.

For example:

<DIGIT> = '0','1','2','3','4','5','6','7','8','9';

This would define a phrase <DIGIT> as any of the digits 0 to 9. Characters such as these digits are enclosed within single quotes as shown. Groups of characters can also appear between quotes if required. Commas separate the alternative forms the phrase may take and a semi-colon terminates the definition.

Keywords are enclosed within double quote characters. For example:

<PROC> = "ROUTINE" , "INTEGERFN" ;

Note that the '%' character can be omitted. When it is not clear what constitutes a single keyword, for instance %INTEGERARRAYNAME, it may be divided up as convenient. For example, the forms:

"INTEGERARRAYNAME"
"INTEGER" "ARRAY" "NAME"

are both valid and equivalent.

A decimal number might be defined by the phrase <N> as follows:

<N> = <DIGIT> <N> , <DIGIT> ;

That is, a number is either a digit followed by a number or is a single digit. This illustrates how phrases can be defined recursively.

It is also possible to have a null alternative i.e. one with no items. For example, if it was desired to define a decimal number optionally preceded by a plus or minus sign, a phrase <SIGN> could be defined:

<SIGN> = '+' , '-' , ;

Then the required definition could be:

<SN> = <SIGN> <N> ;

In other words, the third alternative form of <SIGN>, namely nothing, can precede the <N>. A phrase definition with only one alternative such as this is rather redundant but nevertheless quite valid. The particular syntax analysis scheme used in the SKIMP compiler imposes certain restrictions on the forms phrase definitions may take but discussion of these is postponed until the scheme itself has been described.

The syntax definitions for SKIMP are shown in figure 3.2. The types of statement in SKIMP are defined by the phrase <STATEMENT> and the remaining definitions follow from this. There are two phrases built into the compiler and which therefore do not need to be defined. These are <NAME> and <CONST>. They represent identifiers and constants respectively. The reason for them being built-in is that identifiers and constants are recognised at the lexical analysis stage and hence do not require definitions for use by the syntax analyser.

```
<STATEMENT>= <INSTR>,
             "IF"<COND>"THEN"<INSTR><ELSE>,
             <CONST>':'<STATEMENT>,
             "FINISH"<ELSE>,
             "INTEGER"<ARRAY>,
             <PROC><NAME><FORMAL>,
             "END"<OFPROG>,
             "BEGIN";
<INSTR>    = <NAME><ACTUAL><ASSIGN>,
             '->'<CONST>,
             "START",
             "RETURN",
             "RESULT"'='<EXPR>,
             "STOP";
<EXPR>     = <UNARY><OPERAND><EXPRREST>;
<UNARY>    = '-','\','+', ;
<OPERAND>  = <NAME><ACTUAL>,<CONST>,'('<EXPR>')';
<EXPRREST> = <OP><OPERAND><EXPRREST>, ;
<OP>       = '<<','>>','&','!!','!','**','/','*','+','-';
<COND>     = <TEST><CONDREST>;
<TEST>     = <EXPR><COMP><EXPR>,'('<COND>')';
<COMP>     = '=','#','<=','<','>=','>';
<CONDREST> = "AND"<TEST><ANDCOND>,"OR"<TEST><ORCOND>, ;
<ANDCOND>  = "AND"<TEST><ANDCOND>, ;
<ORCOND>   = "OR"<TEST><ORCOND>, ;
<ELSE>     = "ELSE"<INSTR>, ;
<ACTUAL>   = '('<EXPR><EXPRS>')', ;
<EXPRS>    = ','<EXPR><EXPRS>, ;
<ASSIGN>   = '='<EXPR>, ;
<ARRAY>    = "ARRAY"<NAME><NAMES>'('<EXPR>':'<EXPR>')',<NAME><NAMES>;
<NAMES>    = ','<NAME><NAMES>, ;
<PROC>     = "ROUTINE","INTEGERFN";
<FORMAL>   = '('"INTEGER"<FORM><NAME><NAMES><FORMALS>')', ;
<FORM>     = "ARRAYNAME","NAME", ;
<FORMALS>  = ','"INTEGER"<FORM><NAME><NAMES><FORMALS>, ;
<OFPROG>   = "OFPROGRAM", ;
```

figure 3.2

Syntax Reduction

When the syntax definitions are read in, they are reduced to a numerical form to suit the syntax analyser. The keywords are also picked out and built up into a dictionary for the benefit of the lexical analyser. This dictionary is a tree structure of cells which each represent an individual keyletter. When two or more keywords start with the same keyletters, the same cells are used for the letters in common. For example, the keywords %START and %STOP would be represented by the structure shown in figure 3.3.

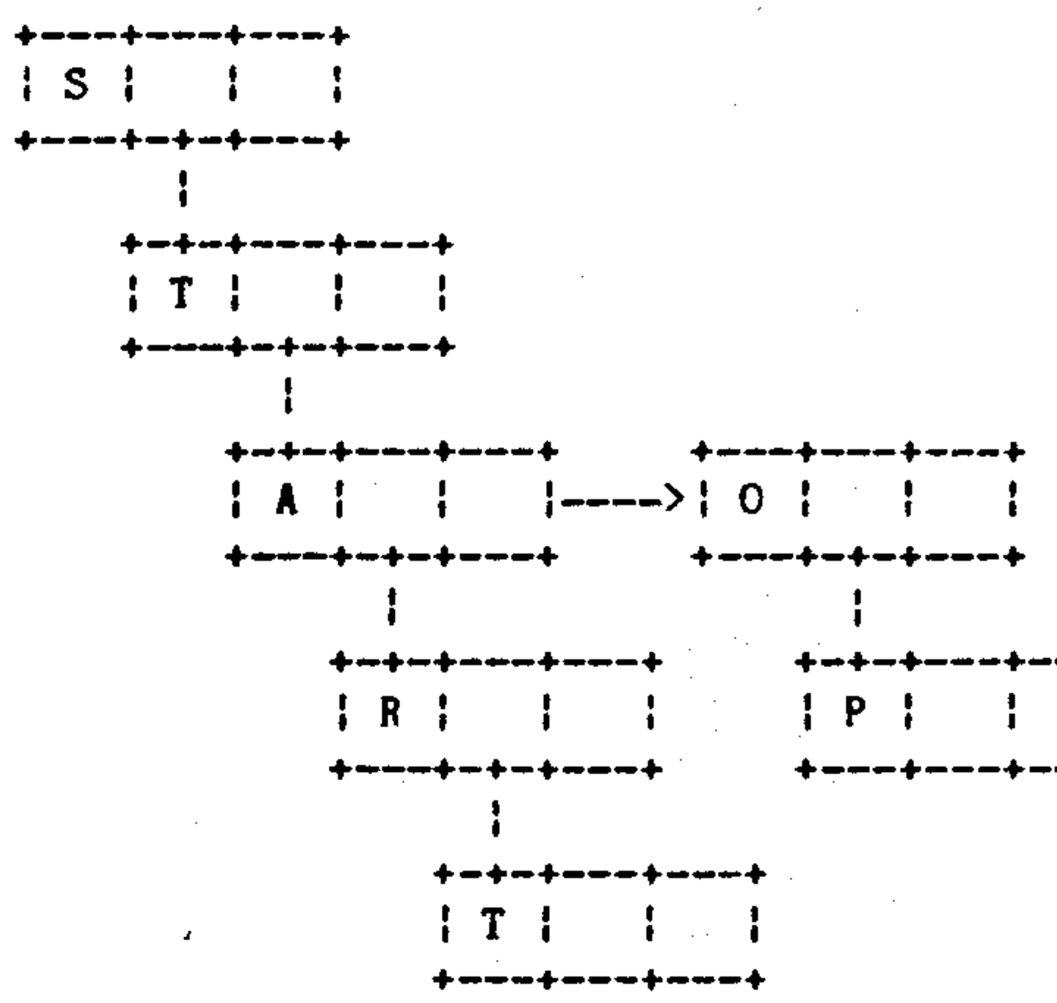


figure 3.3

As can be seen, the letters S and T share the same cells. Each cell has two links in addition. The first links together cells containing successive letters in keywords and the second links together cells containing alternative letters. Thus, the cell containing O is linked from the cell containing A through the second link since A and O are the alternative letters which may follow T. In addition to the three items already described, ther is a further item in the cells at the leaves of the tree which contains an identification number of the keyword which terminates there. These numbers are assigned as the dictionary is created and range from 128 upwards in order to distinguish them from normal characters at the lexical analysis stage. To improve dictionary searching efficiency, the first letter of each keyword is used to index into an array of 26 cells. Tree searching only takes place thereafter.

Just as keywords are assigned identification numbers, so are phrase names. Phrases <NAME> and <CONST> are assigned the numbers 256 and 257 respectively, and other phrases the numbers 258 upwards. These numbers are used in the reduced numerical form of the syntax definitions. In general, the reduced form of a definition is as shown in figure 3.4:

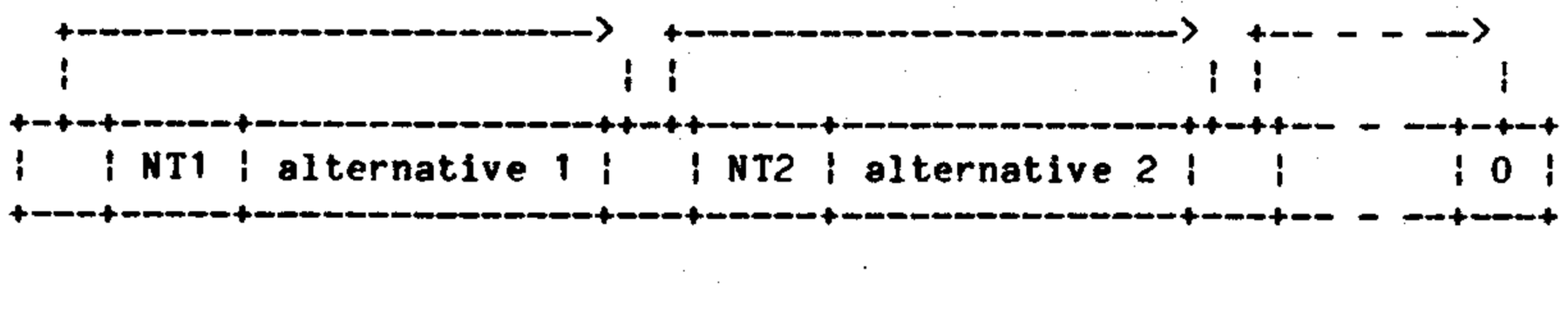


figure 3.4

The zero signifies the end of the definition. The contents of the alternatives fields in the diagram correspond one for one with the items in the definition read in. They are either the numerical equivalents of the literal characters or keyword or phrase identification numbers. NT1, NT2 etc. are the number of phrase items in that alternative. This is of relevance to the syntax analyser. The definitions reduced in this way are stored one after another in an array as shown in figure 3.5.

```
  1    4   1 259  11   3 135 260 145 259 261  16   2 257  58 258  20
 17    1 133 261  24   1 136 262  29   3 263 256 264  33   1 131 265
 33   36   0 130   0  42   3 256 266 267  47   1  45  62 257  50   0
 49  143  53   0 140  58   1 141  61 268  61   0 144   0  67   3 269
 65  270 271   0  71   0  45  74   0  92  77   0  43  79   0   0  84
 81    2 256 266  87   1 257  92   1  40 268  41   0  98   3 272 270
 97  271 100   0   0 105   0  60  60 109   0  62  62 112   0  38 116
113    0  33  33 119   0  33 123   0  42  42 126   0  47 129   0  42
129  132   0  43 135   0  45   0 140   2 273 274   0 146   3 268 275
145  268 151   1  40 260  41   0 155   0  61 158   0  35 162   0  60
161   61 165   0  60 169   0  62  61 172   0  62   0 178   2 128 273
177  276 183   2 138 273 277 185   0   0 191   2 128 273 276 193   0
193    0 199   2 138 273 277 201   0   0 206   1 132 259 208   0   0
209  215   2  40 268 278  41 217   0   0 223   2  44 268 278 225   0
225    0 230   1  61 268 232   0   0 243   4 129 256 279  40 268  58
241  268  41 247   2 256 279   0 253   2  44 256 279 255   0   0 259
257    0 142 263   0 136 134   0 273   4  40 136 280 256 279 281  41
273  275   0   0 280   0 129 137 283   0 137 285   0   0 294   4  44
289  136 280 256 279 281 296   0   0 300   0 139 302   0   0
```

figure 3.5

The identification numbers of the keywords and phrases corresponding to this are shown in figure 3.6. This figure also gives the starting index positions in the array of each of the phrase definitions.

Keywords		Phrases		
128	AND	256	0	NAME
129	ARRAY	257	0	CONST
130	BEGIN	258	1	STATEMENT
131	END	259	37	INSTR
132	ELSE	260	136	COND
133	FINISH	261	202	ELSE
134	FN	262	233	ARRAY
135	IF	263	256	PROC
136	INTEGER	264	264	FORMAL
137	NAME	265	297	OFPROG
138	OR	266	209	ACTUAL
139	OFPROGRAM	267	226	ASSIGN
140	RETURN	268	62	EXPR
141	RESULT	269	68	UNARY
142	ROUTINE	270	80	OPERAND
143	START	271	93	EXPRREST
144	STOP	272	101	OP
145	THEN	273	141	TEST
		274	173	CONDREST
		275	152	COMP
		276	186	ANDCOND
		277	194	ORCOND
		278	218	EXPRS
		279	248	NAMES
		280	276	FORM
		281	286	FORMALS

figure 3.6

If any faults are discovered in the syntax definitions by the reduction routine

the compiler will halt without attempting to compile the source.

Implementation Details

The routines which implement everything described in this chapter are contained in the file 'SKIMPA'. The routine which reads in the syntax definitions is named 'READ PS' and it takes the definitions from a file named 'SKIMPPS'. The resultant reduced form is stored in the array named 'PS'. The positions in this where phrase definitions start are given by the array 'PP' which is indexed by phrase identification number. The information given in figures 3.2, 3.5 and 3.6 is output to a file named 'SKIMPPSL'.

The keyword dictionary is formed from cells which are records of the record array named 'KD'. Their format is:

```
%RECORDFORMAT KDF(%INTEGER L,N,A,B)
```

'L' represents the key letter, 'N' the identification number at a leaf cell and 'A' and 'B' the two links used in the tree structuring. The creation of the dictionary is slightly involved since keywords in the dictionary have to be the 'lowest common denominator' of the keywords in the syntax description. This means that keywords may have to be split up if a new keyword is encountered which is a prefix of an existing one. A local string array 'KA' of keywords is used to aid this process.

The main control loop of the compiler which is illustrated in figure 3.1 is implemented in the main routine named 'SKIMP'. The box 'generate code' corresponds to routine 'STATEMENT' in file 'SKIMPB'. Subroutines used by this routine appear in the other files 'SKIMPC', 'SKIMPD' and 'SKIMPE'. The general strategy in processing statements is to break them down into component syntactic parts and process each of these simpler parts individually. At the highest level, each alternative form in the definition of phrase <STATEMENT> is processed separately by executing a switch on the alternative number in the analysis record of the statement (see next chapter). This is also done at the lower hierarchical levels.

Chapter 4

Line Reconstruction, Lexical and Syntax Analysis

Line Reconstruction

Line reconstruction is the first phase of processing of the raw source text and is a simple cleaning up operation. Its functions are:

1. to mark keyword letters
2. to remove spaces
3. to concatenate continuation lines
4. to discard comments

Since the use of '%' as a prefix can lead to many different ways of representing keywords (c.f. %INTEGERARRAYNAME above), this variability must be removed before they can conveniently be processed any further. This is done during line reconstruction by deleting the '%' and adding the value 128 to each keyletter.

The only significance of spaces is in terminating keywords. Hence, after keyletters have been marked spaces are discarded. All symbols within quotes (and this can include spaces and newlines) are left intact, however.

One statement at a time is read from the source file i.e. up to the first semi-colon or newline. Any number of lines may be concatenated using the '%C' at the end of a line subject only to a limitation of 256 characters in total. This is dealt with by looking at the previous character whenever a newline is encountered. If it is the value 'C'+128 i.e. a '%'-shifted 'C', then it and the newline are discarded and processing continues with the next line. If the first character of a statement is a '!' the statement is regarded as a comment and discarded.

Lexical Analysis

The cleaned up text received from the line reconstruct phase is next processed to find lexical objects. These are:

1. keywords
2. names
3. constants

The start of any lexical object is determined by inspecting each character in turn. A value greater than 128 indicates the start of a keyword. The adjacent characters also greater than 128 are concatenated into a string which is then looked up in the keyword dictionary described previously. The identification number thus produced (ranging from 128 upwards) is stored in the lexical output array in place of the keyletters. Potentially more than one identification number can result from a compound keyword. In this case the look up routine is called repeatedly until all the keyletters have been identified.

If a character value lies between 'A' and 'Z', it and the following letters and digits comprise a name which is then looked up in a dictionary. If the name

is already in the dictionary then its existing identification number is used. If it is not already in, it is inserted and a new identification number generated for it. The output to the lexical array consists of the number 256 (the phrase number of <NAME>) followed by the identification number of the name.

The identification numbers are the positions at which the names are stored in a 'hash table'. The hash table operates as follows. A calculation is performed on the characters of the name itself to produce a 'hash index'. If this position is not already occupied, the name is inserted at that point. If the position is already occupied the name stored there may or may not be the same as the new name being looked up. If it is the same then no further action is required. If it is not, a 'clash' has occurred which must be resolved. Numerous methods of doing this have been invented but the simplest possible one is used here. This is to scan forward cyclically inspecting each successive position in turn. If the new name is encountered in any of these positions, again no further action is required. If, however, an unoccupied position is encountered then the new name cannot be in already and so can be inserted there.

The requirements of the hash calculation are that it should be fast and that as far as possible different names should produce different indexes. This is, of course, unattainable in general since there are many more potential names than indexes but a fairly good attempt can be made. The scheme used in this compiler packs character values into a word, divides this word by a prime number just less than the size of the hash table and takes the remainder as the hash index.

Constants start with either a digit or a single quote character. The lexical action in this case is to evaluate the constant and to insert the number 257 (for phrase <CONST>) followed by the value into the lexical array. For character constants the value consists of the individual character values packed into 8-bit fields of a word right justified.

Characters which are not part of keywords, names or constants are passed through to the lexical array unchanged. Since characters have values less than 128, the type of a lexical item can be determined from the range it lies in.

Syntax Analysis

The compiler uses a top-down recursive descent method of syntax analysis which is directed by the table of reduced syntax definitions described earlier. This analyses the output of the lexical phase with respect to the phrase <STATEMENT>. The algorithm is implemented as a routine which considers just one phrase definition. Where literal characters or keywords appear as items in the definition they are compared directly with the lexical information and where a sub-phrase appears as an item, the routine is called recursively to match it.

The technique involved in matching a phrase is essentially a search in that each alternative in the phrase definition is considered in turn. Similarly, the items in an alternative are considered one after another. An alternative matches if all the items in it have been matched. If any item in the definition does not match, the whole alternative in which that item appears is abandoned and the algorithm moves on to try the next alternative. A flow diagram of the algorithm is given in figure 4.1. Figure 4.2 shows the stages in the analysis of the statement:

```
%INTEGER I,J
```

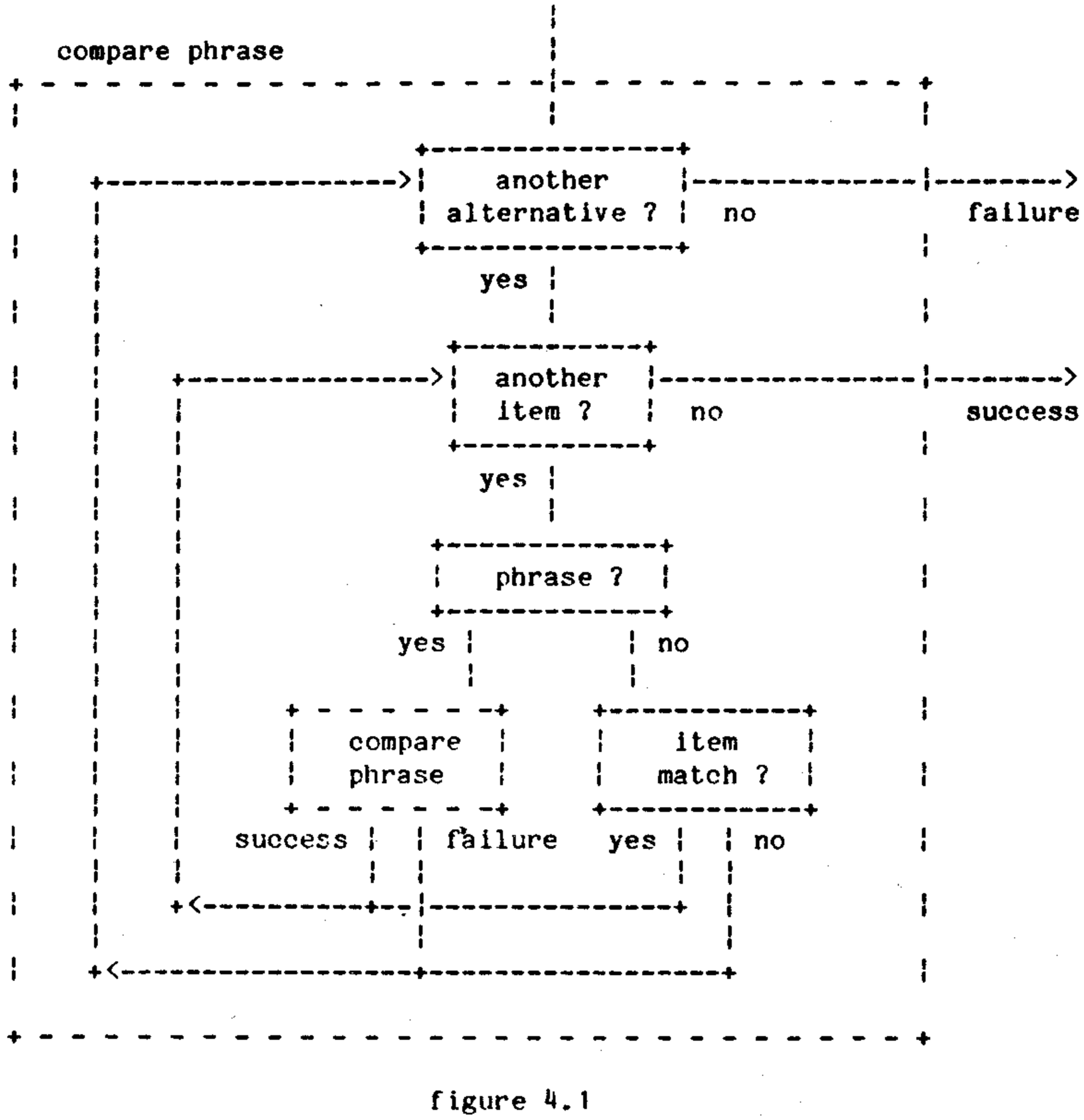


figure 4.1

The restrictions upon the form phrase definitions may take which were mentioned in the previous chapter arise from this comparison algorithm. Consider the definition:

<N> = <N><DIGIT> , <DIGIT> ;

If the analyser were to use this a recursive loop would result since <N> appears as the first item in the first alternative. Similarly, consider the definition:

<N> = <DIGIT> , <DIGIT><N> ;

In this case, the analyser would only be able to recognise single digit numbers since the first alternative would always prove successful and the second alternative would never be tried. Situations of these kinds must be avoided. Fortunately this usually presents no difficulties.

From the efficiency point of view, it is also desirable to avoid other forms of definition. An example might be:

<EXPR> = <OPERAND><OP><EXPR> , <OPERAND> ;

The inefficiency results from <OPERAND> being the first item in both alternatives. Clearly, whenever the second alternative is recognised (the last

```
compare <STATEMENT>
1st alternative
   1st item = <INSTR>

      compare <INSTR>
      1st alternative
         1st item = <NAME> / no match
      2nd alternative
         1st item = '-' / no match
      3rd alternative
         1st item = "START" / no match
      4th alternative
         1st item = "RETURN" / no match
      5th alternative
         1st item = "RESULT" / no match
      6th alternative
         1st item = "STOP" / no match
      failure to match <INSTR>

2nd alternative
   1st item = "IF" / no match
3rd alternative
   1st item = <CONST> / no match
4th alternative
   1st item = "FINISH" / no match
5th alternative
   1st item = "INTEGER" /  match
   2nd item = <ARRAY>

      compare <ARRAY>
      1st alternative
   1st item = "ARRAY" / no match
2nd alternative
   1st item = <NAME> /  match (I)
   2nd item = <NAMES>

      compare <NAMES>
      1st alternative
         1st item = ',' / match
         2nd item = <NAME> / match (J)
         3rd item = <NAMES>

            compare <NAMES>
            1st alternative
               1st item = ',' / no match
            2nd alternative
               null item / match
            successful match of <NAMES>

         no more items
      successful match of <NAMES>

   no more items
successful match of <ARRAY>

no more items
successful match of <STATEMENT>
```

figure 4.2

operand of an expression) phrase <OPERAND> will have been recognised twice. This could have been avoided by writing the definition of <EXPR> in the following way:

```
<EXPR> = <OPERAND><REST> ;
<REST> = <OP><OPERAND><REST> , ;
```

Definitions with only one alternative are, strictly, unnecessary and result in the inefficiency of an extra recursive call, but are occasionally used for the sake of clarity. <EXPR> and <COND> are examples in the SKIMP syntax.

Efficiency is also partly the reason for recognising names and constants at the lexical stage. If their recognition were left to this algorithm, two phrases such as:

```
<LETTER> = 'A' , 'B' , 'C' , . . . . 'Z' ;
<DIGIT> = '0' , '1' , '2' , . . . . '9' ;
```

would probably have to be introduced. Since the algorithm searches through the alternatives, the recognition of letters and digits would be very slow.

Also in view of the searching, it will clearly be beneficial to put the most commonly occurring forms of alternative first in the definition of a phrase. In particular, <STATEMENT> and <INSTR> are ordered in what is hopefully a sensible way with this consideration in mind.

Analysis Records

During the course of syntax analysis, a record is kept of which phrases have been recognised. This information is passed to the code generation phase of the compiler and directs its course of action. The record consists of a linearisation of the analysis tree corresponding to the statement recognised. For instance, the analysis tree corresponding to the example above would take the form shown in figure 4.3.

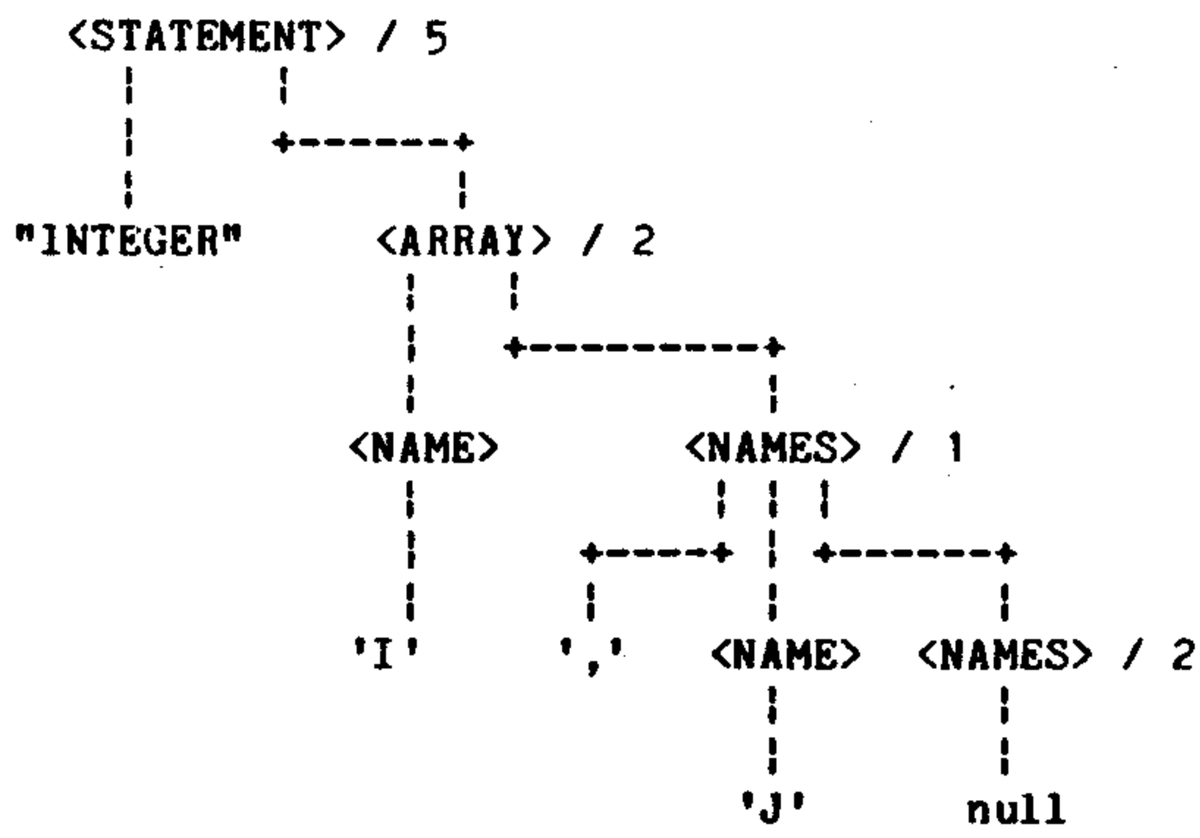


figure 4.3

The number after the each phrase name in the tree is the number of the

alternative in its definition which was matched. These numbers and the pointers emanating from each phrase comprise the information stored in the linearised form of analysis record. For each phrase, the record takes the form shown in figure 4.4:

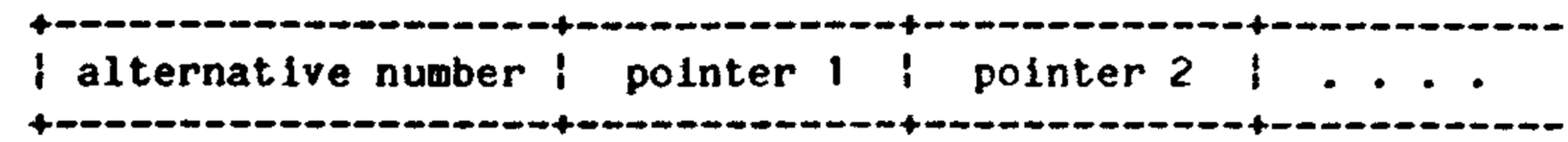


figure 4.4

The information is pruned to a certain extent, however. There are only pointers for the phrase items matched. All reference to keywords, literals and null alternatives is omitted since their presence can be inferred from the alternative numbers. Phrases <NAME> and <CONST> are treated specially since they were recognised at the lexical stage. To be consistent with other phrases, both are given an alternative number matched of 1. In addition, for <NAME>, the following word is set to contain the identification number of the name and for <CONST>, the following word is set to the value of the constant. The analysis record for the axample is therefore as shown in figure 4.5:

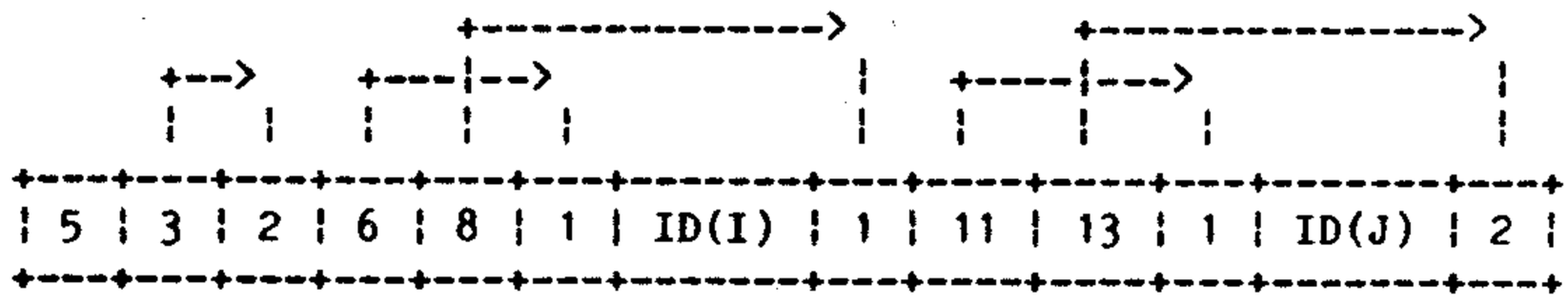


figure 4.5

The phrase names themselves also do not appear in the record as can be seen above. The pointers are just indexes into the array in which the record is stored. To aid clarity, when the analysis record is printed out by the compiler (the ANAL option) the positions at which phrases start are indicated by inserts such as:

(1/STATEMENT) (3/ARRAY) (8/NAMES)

Implementation Details

Line reconstruction and lexical analysis are implemented together in the one routine which is named 'READ STATEMENT' and which appears in file 'SKIMPA'. Subsidiary to this are routines which deal with each kind of lexical object. These are named 'KEYWORD', 'NAME' and 'CONST'. Reconstructed text is placed in an array named 'TT' and this is lexically reduced into an array named 'T'. The name dictionary which is created during lexical processing comprises an array named 'NAMEDLINK' and a byte array named 'NAMED'. The former is used as the hash table and the entries in it point to positions in the latter array where the characters of the names are stored as concatenated strings. For example, figure 4.6:

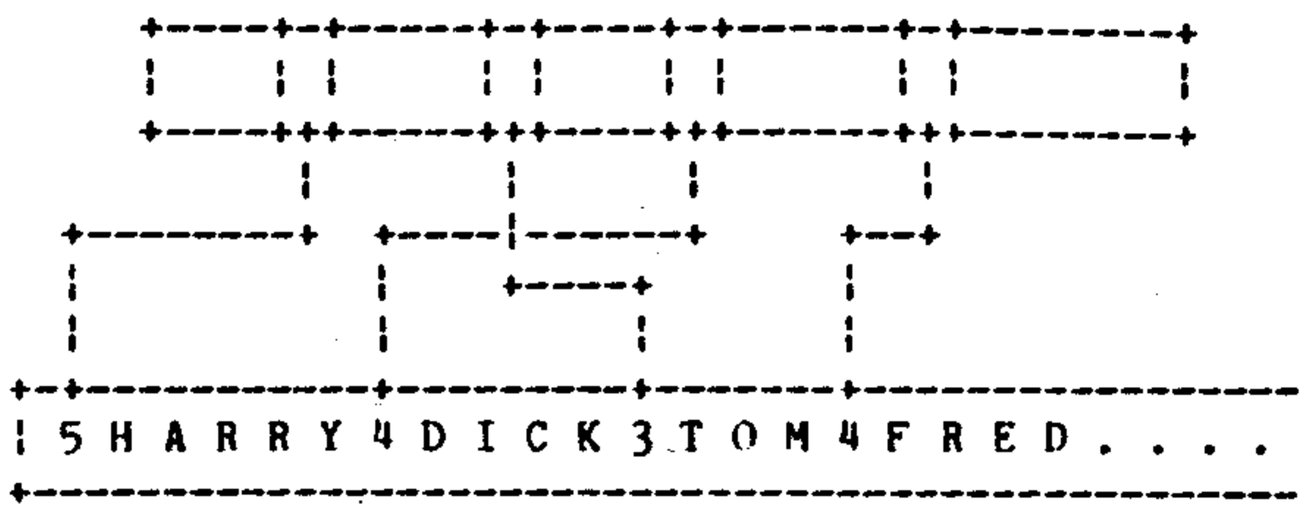


figure 4.6

The syntax analysis routine is named 'COMPARE', also in file 'SKIMPA', and is a function which returns the value 1 for success and 0 for failure. It takes as its single parameter the identification number of the phrase to be matched in this call. Hence, it is initially called with the number of phrase <STATEMENT>. Since <STATEMENT> is assumed always to be the first phrase definition to be read in, this number is 258. Phrase numbers 256 and 257, <NAME> and <CONST> respectively, are picked out on entry to the routine and treated as literals since they have already been recognised. The analysis record is placed in an array named 'A'.

Processing Analysis Records

The basis of the processing rests on knowing the starting position in the analysis record of each phrase which is of concern. The values in these positions indicate the alternative which was actually recognised by the comparison routine and it is this information which directs the code generation. The form of analysis record facilitates finding out where the phrases start. These are simply the values of the pointers which follow each alernative number. Consider, for example, the phrase structure associated with the declaration of variables:

```
<STATEMENT> = . . . . . ,"INTEGER"<ARRAY>, . . . .
<ARRAY> = "ARRAY"<NAME><NAMES>'('<EXPR>':'<EXPR>')',<NAME><NAMES>;
```

The analysis record for:

```
%INTEGER I, J, K
```

would be:

```
5  3  2  6  8  1  ID(I)  1  11  13  1  ID(J)  1  16  18  1  ID(K)  2
```

The compiler fairly consistently uses names for the positions of where phrases start in the analysis record which consist of the name of the phrase with 'P' (for 'pointer') appended. In the case of phrase <STATEMENT> the variable is 'STATEMENTP' and it is the value in the analysis record at that point which causes the main switch in routine 'STATEMENT'. For the positions of phrases <ARRAY>, <NAME> and <NAMES>, the variables are 'ARRAYP', 'NAMEP' and 'NAMESP' respectively. The following IMP code shows how the values for these are set up and the declaration of scalars is handled:

```
ARRAYP=A(STATEMENTP+1)
NAMEP=A(ARRAYP+1)
NAMESP=A(ARRAYP+2)
%IF A(ARRAYP)=2 %START
  %CYCLE    ;! for each name declared
    PUSHTAG(A(NAMEP+1), etc. )
    NEXTRAD(LEVEL)= etc.
    %IF A(NAMESP)=2 %THEN %EXIT
    NAMEP=A(NAMESP+1)
    NAMESP=A(NAMESP+2)
  %REPEAT
%FINISH %ELSE %START
  ! array declarations
```

. . . .

The routine call on 'PUSHTAG' and the following line deal with the declaration of each name. This is described in more detail in chapter 6. Each time round the loop the variables 'NAMEP' and 'NAMESP' are updated for the next name and the exit from the loop takes place when the null alternative of phrase <NAMES>, alternative 2, is encountered.

Chapter 5

Storage Organisation

Layout of Store

The compiler produces object code which assumes the layout of store shown in figure 5.1. The object code itself starts at address zero and the table of constants and run-time stack follow consecutively. If it was so desired, however, the code could easily be made relocatable and the two other areas could be located elsewhere, for instance in separate segments on a machine with such an architecture.

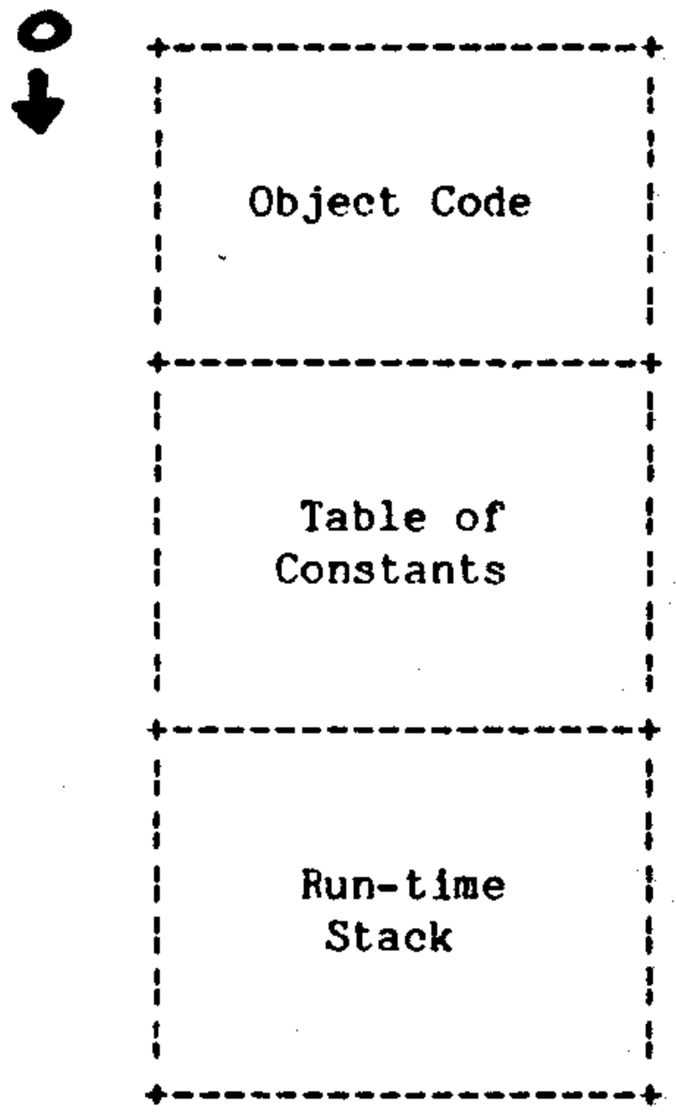


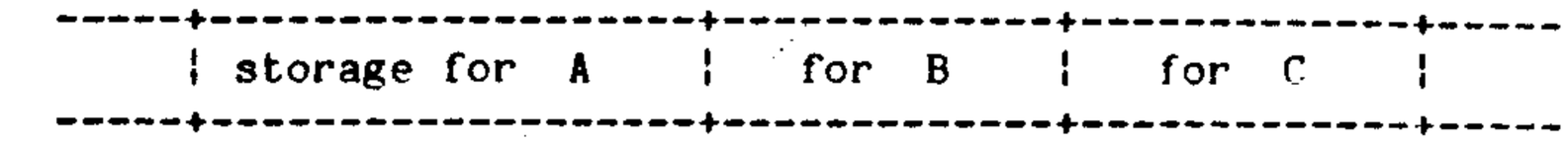
figure 5.1

The entry point to the object code is at address zero. The code dumped at this position sets the register 'COT' to the base of the table of contants and also sets the first display register, 'DR1', to the start of the stack (see below). Accesses to data are always made via index registers.

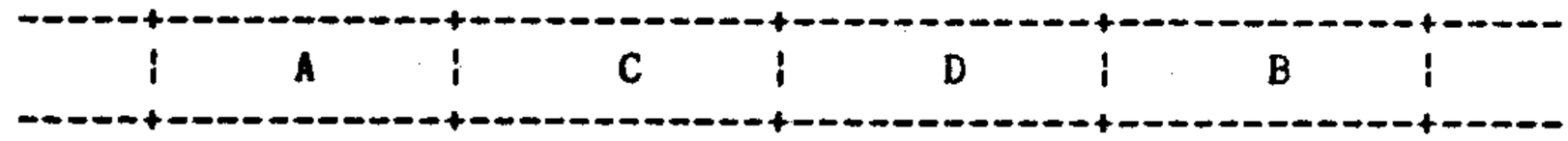
Run-time Storage Allocation and Addressing

The block structure of SKIMP leads naturally to the use of a stack for the allocation of storage. In particular, the possibly recursive calling of routines and functions is most conveniently dealt with using a stack. Storage for local variables and arrays is allocated when the routine or function is entered and deallocated when it is left. These areas are sometimes known as 'local name spaces'. Return addresses can also be preserved on the stack.

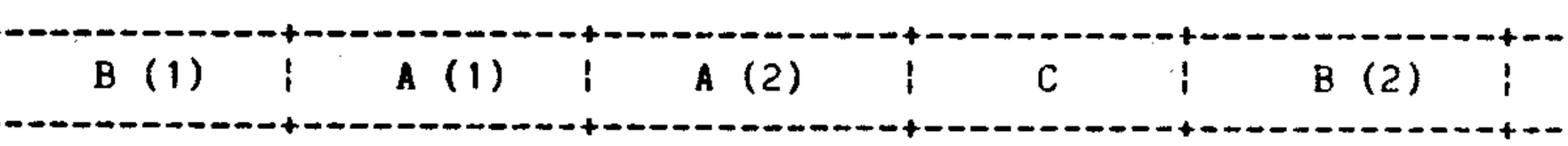
Consider a number of routines 'A', 'B', 'C' etc. and the storage allocation for variables and arrays in those routines. Suppose routine 'A' is called which then calls 'B' and which then calls 'C'. A picture of the stack would be:



For calls in a different order, say 'A' called 'C', 'C' called 'D' and 'D' called 'B', the picture would be:



Similarly recursive calls, say 'B' called 'A', 'A' called 'A', 'A' called 'C' and 'C' called 'B':



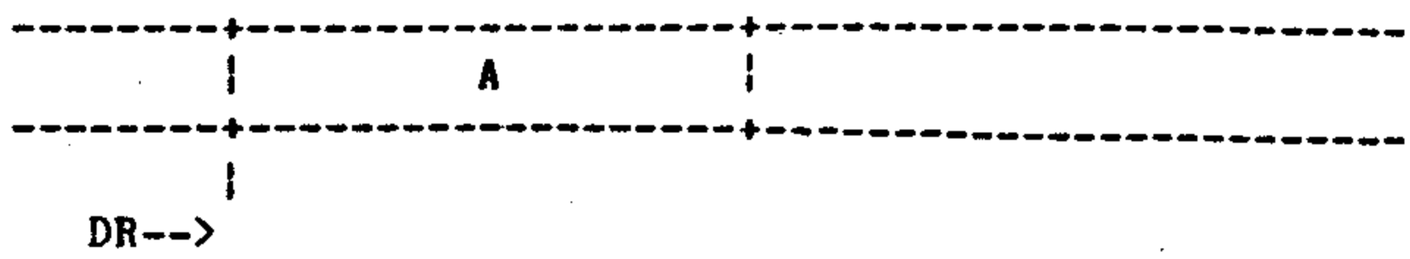
Note that completely new storage areas are allocated for recursive calls. A result of this 'allocate when called' scheme is that the storage for routine is liable to start anywhere on the stack and not in a fixed place. The compiler cannot therefore assign addresses to the variables as it comes across them in declarations at compile-time. However, it can assign them addresses relative to the start of the storage area for that routine and this it does. To access a variable at run-time therefore requires both this 'relative address' and the address of the start of the appropriate local name space. These start addresses can be kept in registers thus allowing the 'base register and displacement' form in a single object instruction to be used to access variables.

A register need not be allocated to each routine in the program, however. Only those which are currently active i.e. called, and which contain variables in scope at the current point of execution of the program, need have corresponding registers. Since only one routine can be in scope at each textual level i.e. routine nesting depth, only one register per textual level is needed. This set of registers is known as a 'Display'. For machines with no or only a few index registers, a display can be kept in store rather than registers but then extra overheads are involved whenever variables are accessed. A typical situation is that in which there are sufficient registers for one to be permanently allocated to the most recent local name space and perhaps another to be used for all global accesses. The loss of efficiency here is usually fairly acceptable. No such difficulties arise with the SKIMP machine.

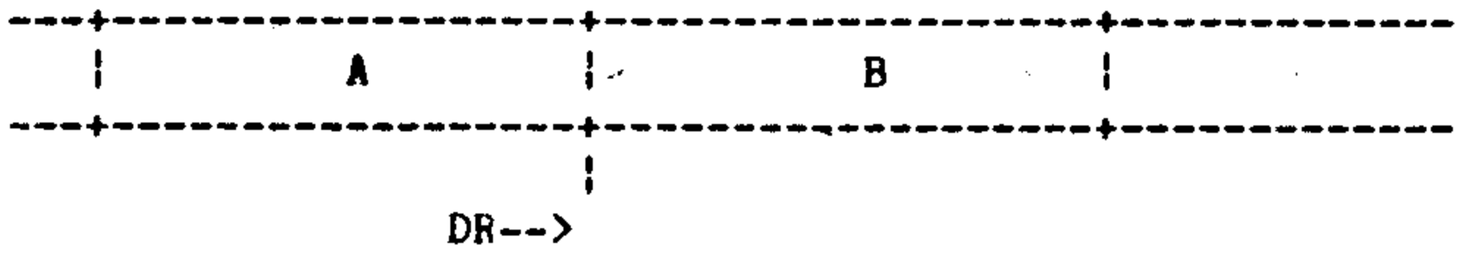
Clearly, the display will need to be updated on routine entry and exit. Consider the following section of program with two routines at the same textual level:

```
%ROUTINE A
. . .
%END
%ROUTINE B
. . .
%END
```

When 'A' is called, the stack situation will be:



where 'DR' is the register allocated to point at the start of 'A's local name space. When 'B' is called from 'A', the variables in 'A' are no longer in scope i.e. are inaccessible, so 'DR' can be re-used to point to 'B's local name space:



When 'B' returns, 'A's variables must again become accessible. Hence, the previous value of 'DR' must be preserved. This will be true for all the registers of the display. Their values must be preserved when they are updated so that they can eventually be restored. The stack can also be used for this purpose. The object code the SKIMP compiler produces puts preserved display values in the first location of each local name space followed by the return address to the calling routine. Storage for variables starts after that.

This discussion leads to the following scheme for updating the display. The display is represented by the registers 'DR1', 'DR2', 'DR3', etc. and the register which is used to point to the start of unallocated storage on the stack is named 'STP', standing for STack Pointer. The call of a routine or function is made by means of an instruction of the form:

```
BAL,   WK,   ,   (start address)
```

which leaves the return address in a 'work' register named 'WK'. The code at the point of entry to a routine is:

```
STR,   DR(level of routine),   STP,   0
LDA,   DR(level of routine),   STP,   0
STR,   WK,   STP,   1
ADD,   STP,   ,     (storage area size)
```

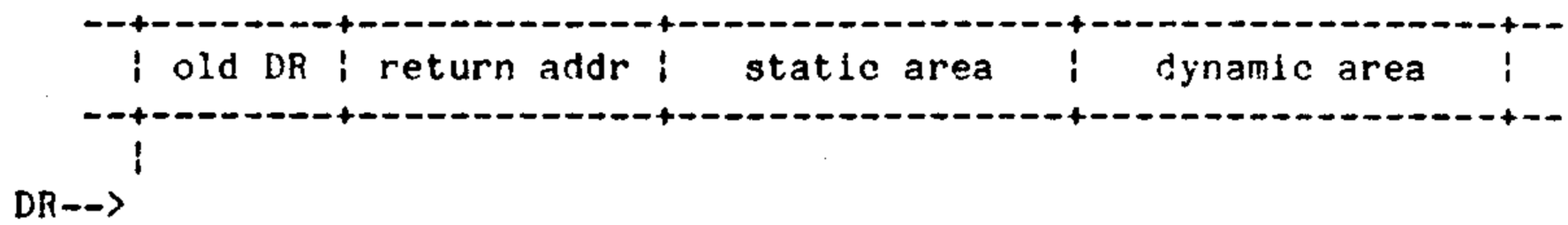
At a point of exit i.e. %RETURN, %RESULT=, or %END, the code is:

```
LDA,   STP,   DR(level of routine), 0
LOAD,  DR(level of routine),   STP, 0
LOAD,  WK,   STP,   1
B,     ,     WK,   0
```

Array Addressing

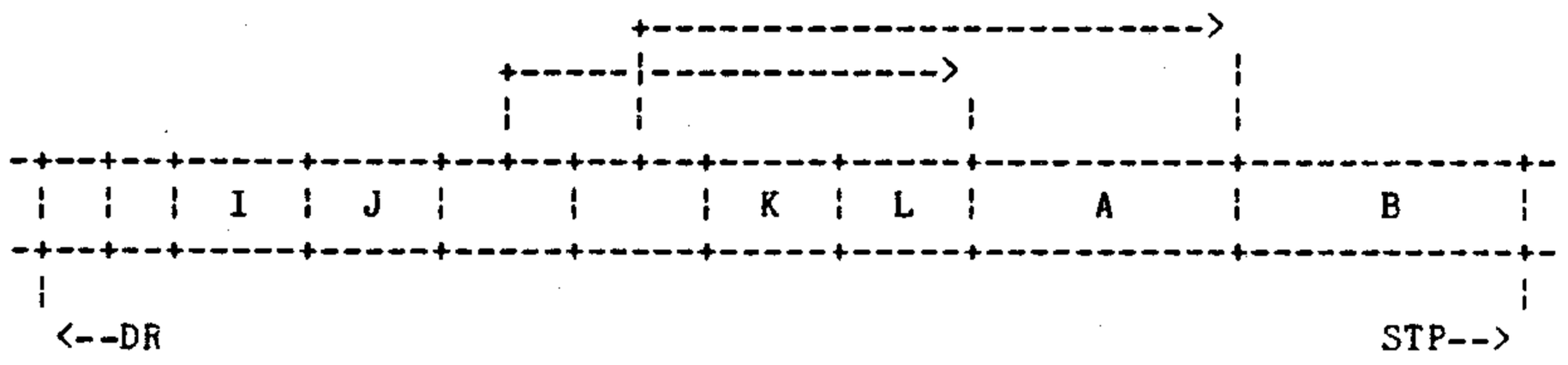
As mentioned previously, variables are allocated locations relative to the start of a local name space as they are declared. A problem arises for arrays since they may have bounds which are evaluated at run-time which means that the amount of space they need is not known at compile-time. If the space for arrays and variables were allocated in the order of declaration, variables declared after an array would not have a fixed relative address. The solution is to

allocate the space for arrays after that for the variables as shown in the diagram:



Each array is now allocated a single location in the static area and this points to the place in the dynamic area where the storage for the array is located. For example:

```
%INTEGER   I, J
%INTEGERARRAY   A, B( ? : ? )
%INTEGER  K, L
```



When register 'STP' is incremented at routine entry, it is just incremented past the static storage area. The object code output for an array declaration allocates space beyond 'STP' and moves 'STP' up as appropriate for the size of the array. Variables in the static area have an address:

DR(textual level of declaration) + relative address of variable

which allows them to be accessed in one instruction. Array elements on the other hand can only be accessed indirectly via the pointer. If this points to the position of the possibly hypothetical zero element of the array rather than the first actual location, the address of an element such as:

A(expression)

will be:

[DR(level) + rel. addr. of pointer] + value of expression

where the square brackets indicate 'contents of'.

In addition to incrementing 'STP' the object code for an array declaration must also fill in this zero address. For the general case of an array declaration:

%INTEGERARRAY A,B,....(lower : upper)

the form of the object code is:

```
code to evaluate 'lower' in 'ACC'
STR,   ACC,   DR(level),   temp1
code to evaluate 'upper' in 'ACC'
LDA,   ACC,   ACC,         1
STR,   ACC,   DR(level),   temp2
SUB,   STP,   DR(level),   temp1
STR,   STP,   DR(level),   rel. addr. of 'A's pointer
ADD,   STP,   DR(level),   temp2
SUB,   STP,   DR(level),   temp1
STR,   STP,   DR(level),   rel. addr. of 'B's pointer
ADD,   STP,   DR(level),   temp2
. . . . .
```

'temp1' and 'temp2' are temporary working storage locations set aside in the local name space to hold the values of 'lower' and 'upper+1' respectively. (Similar working storage is also used for partial results when evaluating expressions.) For each array declared the value 'STP-lower' before 'STP' is incremented gives the zero element address and that plus 'upper+1' gives the final value of 'STP' having allowed space of length 'upper-lower+1' for the array.

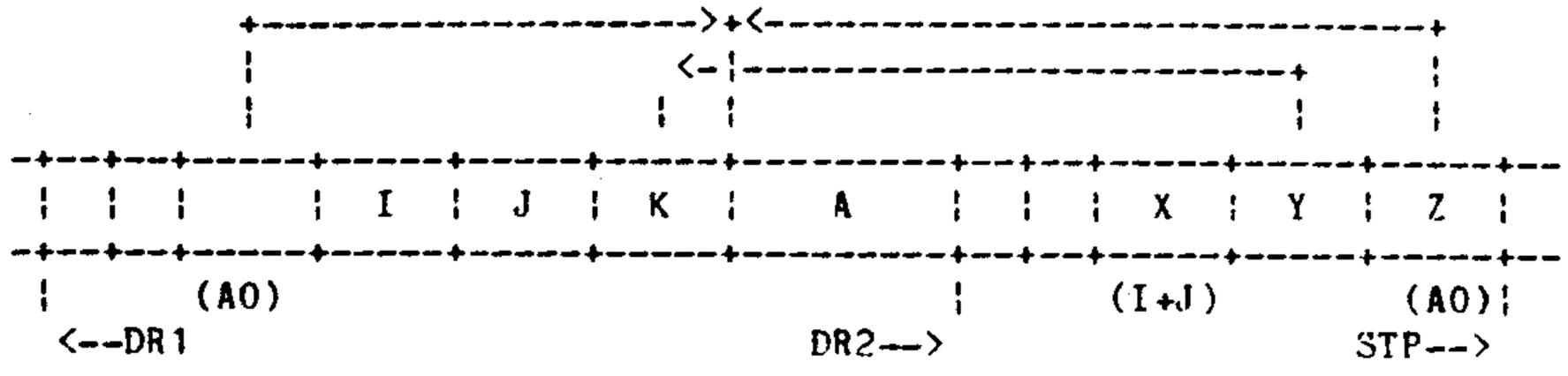
Parameter Passing

Parameters are regarded as initialised local variables of routines in SKIMP. Passing parameters for a routine call therefore consists of setting these initial values. This is quite straightforward since the the parameters will have the first relative addresses to be allocated in the static area of the routine's local name space. Before the entry code has been executed, the start of the future local name space will pointed to by the current value of 'STP'. This allows the values of the parameters to be stored using addresses relative to 'STP' before the 'BAL' is executed.

The three types of parameter are each treated differently. For %INTEGER value type parameters, the expression in the call is evaluated in register 'ACC' and then stored in the appropriate place. For %INTEGERNAME type parameters, the address of the variable or array element is evaluated in register 'ACC' and then stored. Thereafter, the parameter has to be accessed indirectly via this address. For %INTEGERARRAYNAME type parameters, the word containing the zero element address of the array in the call is copied to the parameter location. This allows accesses to array name parameter elements to be handled by the compiler in exactly the same way as those to ordinary array elements.

To illustrate the scheme, consider the stack when routine 'R' in the following program is called:

```
%BEGIN
%ROUTINE R(%INTEGER X, %INTEGERNAME Y,%INTEGERARRAYNAME Z)
. . . . .
%END
%INTEGERARRAY A ( 0 : ? )
%INTEGER I, J, K
R(I+J, K, A)
. . . . .
```



The object code for such a call would be:

```
LOAD,   ACC,   DR1,   3   ]- i+j
ADD,    ACC,   DR1,   4   ]
STR,    ACC,   STP,   2 ->
LDA,    ACC,   DR1,   5   )- name
STR,    ACC,   STP,   3   )
LOAD,   ACC,   DR1,   2   )- array name
STR,    ACC,   STP,   4   )
BAL,    WK,    ,      (start address of R)
```

If a variable of %INTEGERNAME type i.e. a parameter, is passed as the actual parameter for another %INTEGERNAME parameter, the word containing the reference to the original actual parameter is copied to the position for the new parameter thereby maintaining just one level of indirection to reach the required address.

Implementation Details

Declarations of both scalar variables and arrays are processed as alternative 5 of phrase <STATEMENT> in file 'SKIMPB'. The routine 'PUSHTAG' which records details of declarations for future use is described in the next chapter. Relative addresses in the static area of any local name space are allocated sequentially. However, routines may be nested and may appear in the middle of the variable declarations of the enclosing routine. Hence, the next relative address to be allocated must be retained for each textual level up to the current level to allow for this. These values are stored in the array 'NEXTRAD' which has an element for each possible textual level. Since the compiler variable 'LEVEL' always contains the value of the current textual level, the assignment of relative addresses always uses 'NEXTRAD(LEVEL)'.

Routine 'GET WORK' allocates locations in the local name space for temporary storage of partial results, in this instance in connection with array declarations. They are handled in exactly the same way as scalar declarations.

Routine entry and exit sequences are dumped by the routines 'ENTER' and 'DUMP RETURN' which appear in file 'SKIMPD'. Note that 'ENTER' also handles the '%BEGIN' statement since this is similar in many respects.

Chapter 6

Identifier Tag Handling

All identifiers in SKIMP either have to be declared or have to be specified before they can be used. This is to enable the compiler to generate the appropriate code when the identifier is later encountered. In practice this means that the compiler must preserve information about each identifier as it is declared or specified and this will be referred back to each time the identifier is used. This information on each identifier is often called a 'tag'. There are several different items of information which are packed together to form the tag which are:

1. Form of identifier
2. Type of identifier
3. Number of parameters (or dimensions)
4. Textual level of declaration
5. Relative address or start address

1. The form of an identifier is intended to differentiate between scalar variables and arrays, value and reference types of parameter, and data and program objects.

2. The type of an identifier specifies whether it is %INTEGER in nature or not. (In extensions to SKIMP the type would distinguish between %INTEGER, %REAL, %COMPLEX, %STRING etc.)

The various forms and types in SKIMP as at present implemented are given the code values shown below:

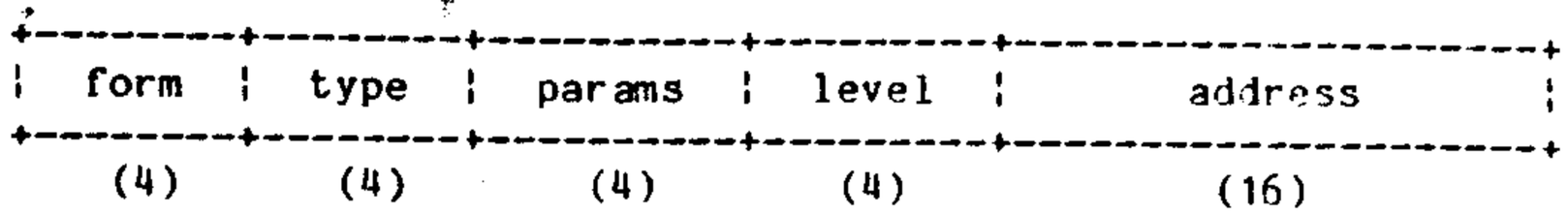
	form	type
%INTEGER	0	1
%INTEGERNAME	1	1
%INTEGERARRAY	2	1
%INTEGERARRAYNAME	3	1
%ROUTINE	4	0
%INTEGERFN	4	1

3. If SKIMP implemented arrays of more than one dimension, the number of dimensions would have to be saved. As this is not the case this information is redundant. The corresponding information for routines and functions is needed, however. This is the number of parameters they take. Information about the parameters occupies further tag words using a scheme described later.

4. The textual level of declaration of each name must be saved since this is used to select the appropriate display base register for addressing purposes.

5. To complete the information needed for addressing variables and arrays, the relative addresses assigned must be saved. For routines and functions the address of their entry points must be stored so that the correct address can be output for the BAL instructions.

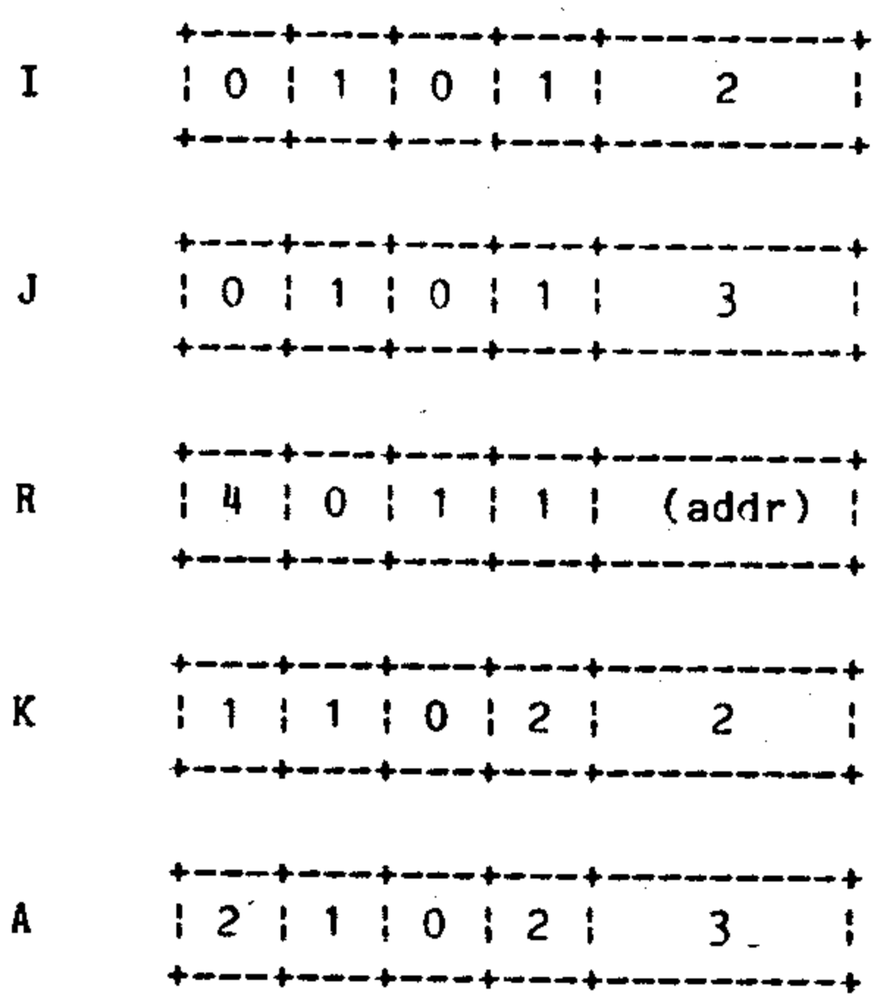
These items are packed into a single word in the following format:



As an example, consider the program:

```
%BEGIN
%INTEGER I,J
    %ROUTINE R(%INTEGERNAME K)
    %INTEGERARRAY   A(1:10)
    .......
    %END
.......
%ENDOFPROGRAM
```

The tag words for these names would be:



Values of tags can be inspected in the output from the compiler by using the option '/TAGS' when calling the compiler. They are printed out (in hexadecimal) at the %END of every routine and function and at %ENDOFPROGRAM for the names that were locally declared. An example of this can be found in appendix 1.

The packing of the tag word places some limits on the programs the compiler can compile, for instance only up to fifteen routine parameters and only up to fifteen textual levels, but these are not serious limits for this compiler. (In fact, less than fifteen textual levels are permitted due to lack of registers.) This word of information has to be made available whenever the name it corresponds to is used. Since names are identified by a dictionary position i.e. hash position, a further array is used in parallel. This contains pointers to the list cells which contain the tag information. This is be illustrated in figure 6.1 for the preceding example again.

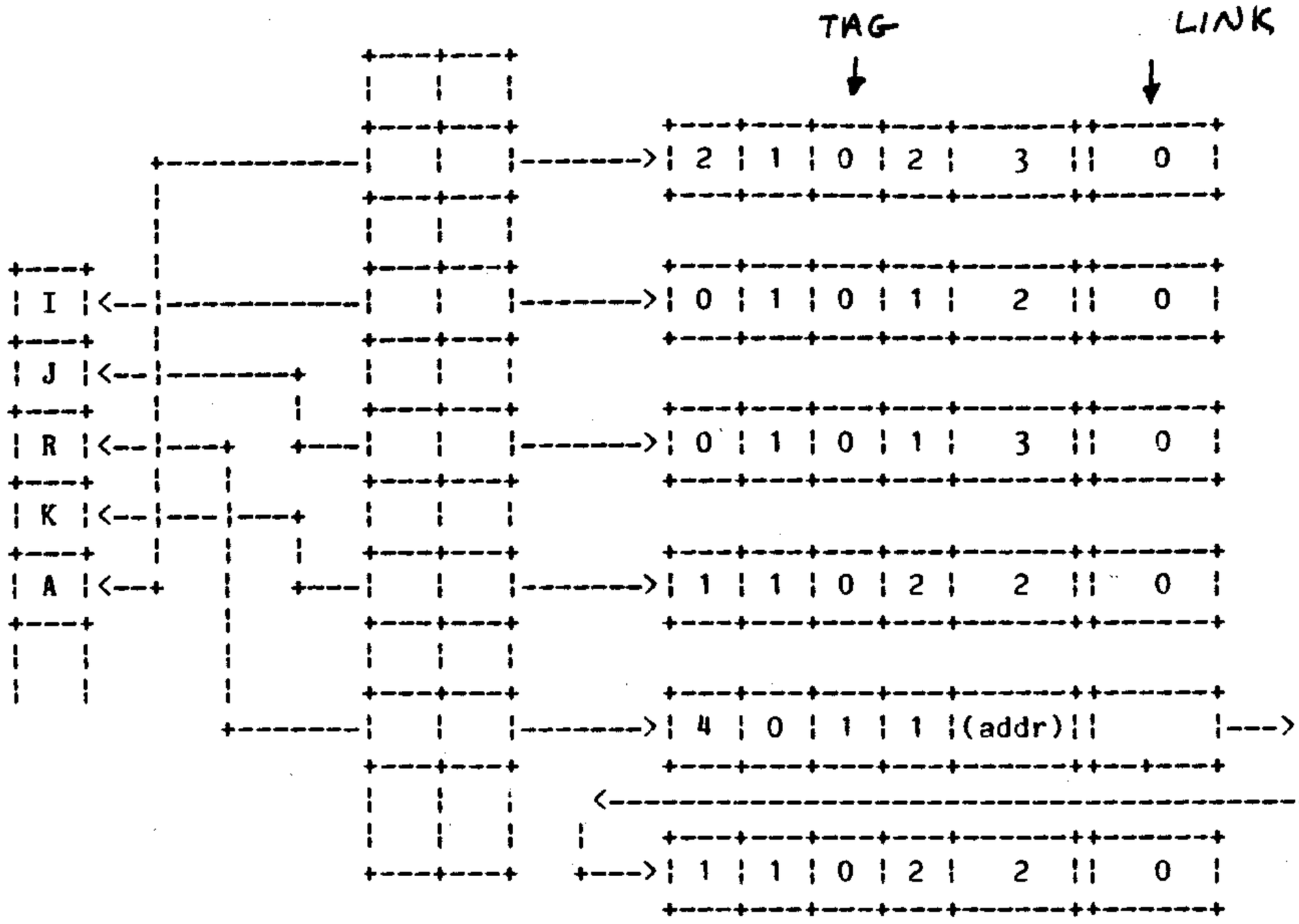


figure 6.1

The diagram also shows how the information on parameters is stored. This is in further list cells beyond that for the name of the routine. In general, the structure is as shown in figure 6.2. Note that there is now a duplication of the tag information on the parameter, firstly, in the chain of cells after the tag for the name of the routine and secondly in the dictionary position for the parameter name. However, this second occurrence is only present during compilation of the routine i.e. when the parameter name is in scope, whereas the first occurrence is present whenever the routine name is in scope i.e. whenever parameter information is needed to compile the routine calls.

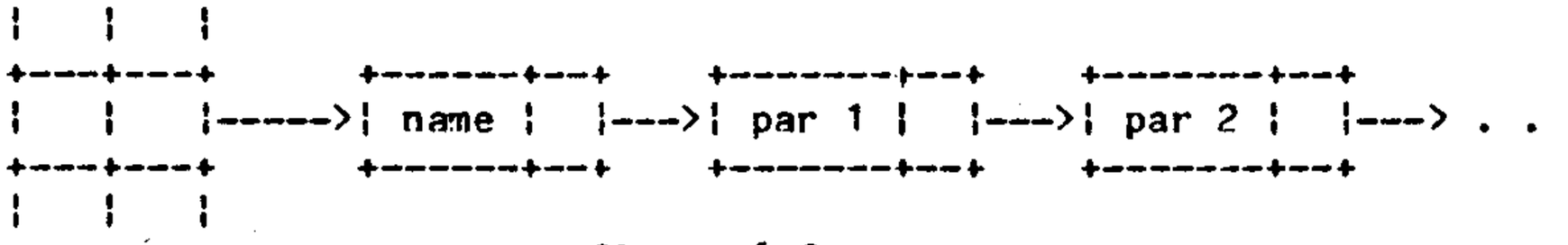


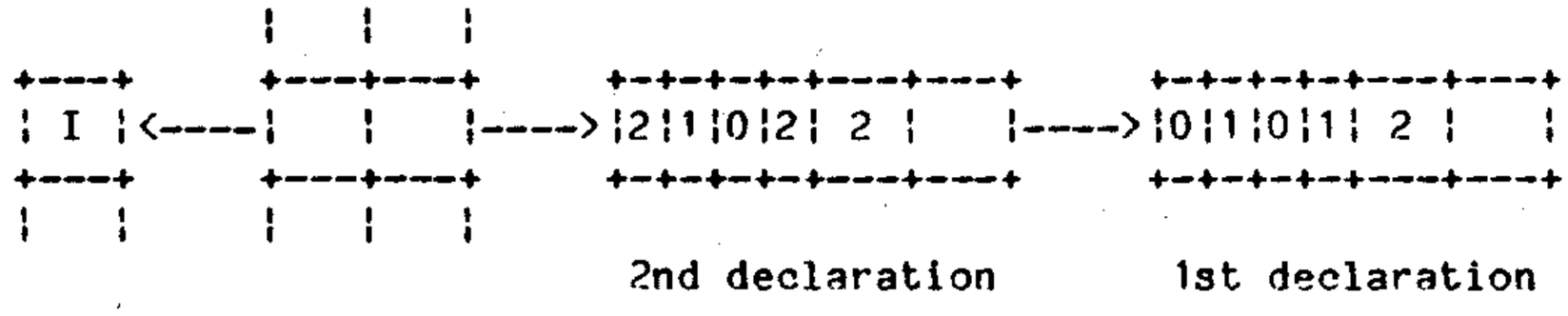
figure 6.2

Only those names which are in scope at the position in the program the compiler is working should have a tag cell. A tag cell is therefore pushed down whenever a name is declared and popped up again (together with all the parameter tag cells for routine names) when it is 'undeclared' i.e. at the end of the routine or function in which it was declared. This also allows for the redeclaration of the same name at an inner nested routine level. The tag at the head of the pushdown list is thus always the one for the latest declaration. If the redeclaration is a routine name, of course, the original tag is beyond all the parameter information cells in the list.

Consider the following program:

```
%BEGIN
%INTEGER I
   %ROUTINE R
   %INTEGERARRAY I(1:10)
   ......
   %END
......
%ENDOFPROGRAM
```

The tags for this whilst the compiler is compiling routine 'R' will be:



If when a call is popped up, nothing remains on the list i.e. this was not a redeclaration, then the space in the name dictionary for that name can also be recovered. This is the case since the dictionary works on a last-in-first-out basis, a consequence of the nested structure of SKIMP programs. The question arises as to how the compiler knows which tags to pop up when an %END statement is reached. An obvious method would be to search through the dictionary for tags having the appropriate textual level in the level field. A more efficient method, however, which avoids searching, is to remember in a list the ones which have to be popped up. Each cell in the list contains the dictionary position of a name, so that this list can be scanned down at the %END statement and a 'popup' performed for every position mentioned in it (plus extra 'popups' for routine parameter tag cells). Because of the nesting of routines in SKIMP a separate list has to be maintained for each textual level. An array with one position per textual level contains pointers to the heads of these lists as shown in figure 6.3:

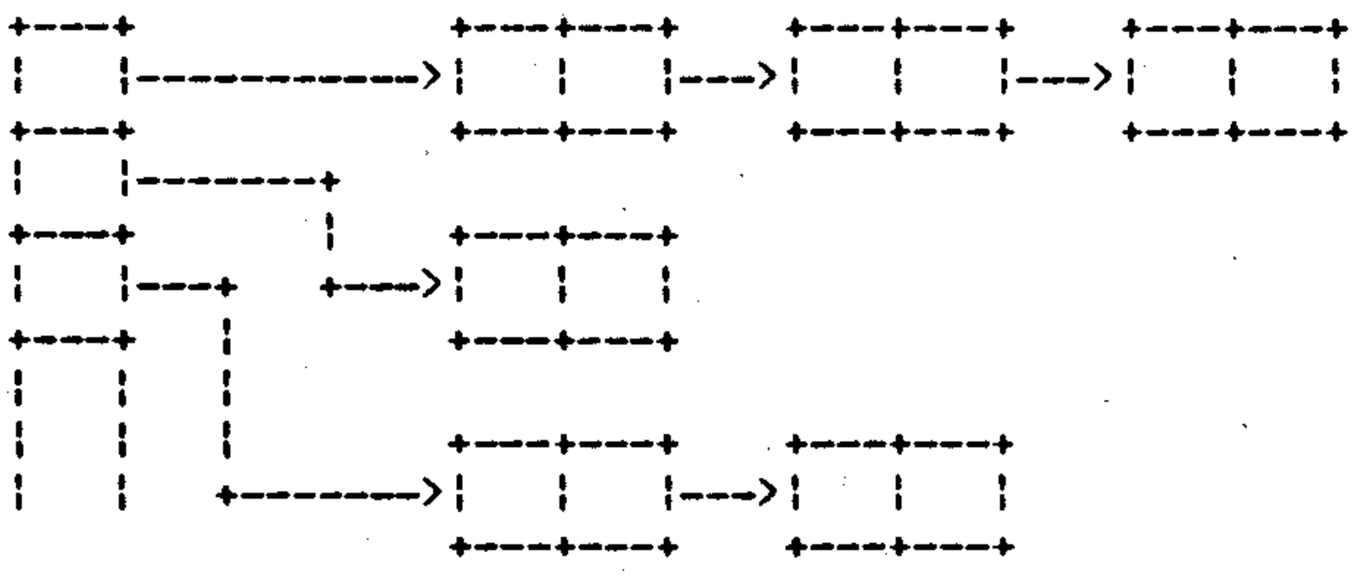
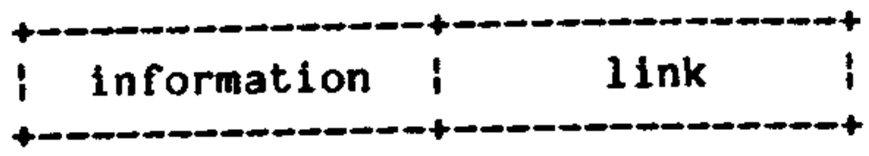


figure 6.3

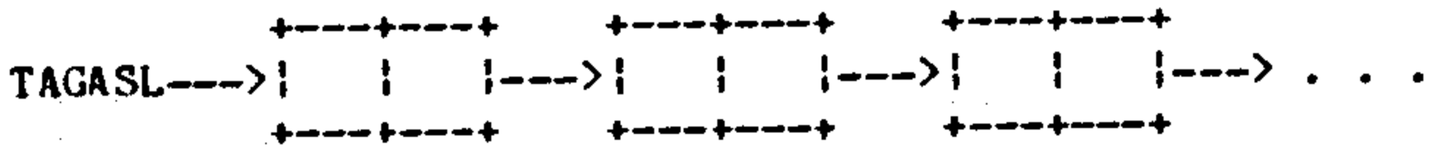
Implementation Details

Lists of calls are frequently used by the compiler to maintain current status information during compilation. Some are used as pushdown popup stacks but not

all. A list cell in this implementation consists of two words, one an information word and the other a link word:



Information words are taken from an array named 'TAG' and link words from an array named 'LINK'. Initially, all these pairs of words are formed into an 'Available Space List' or 'Free List' and thereafter cells are taken from this list as required and returned to it when no longer needed. The variable 'TAGASL' always points to the head of the available space list:



By convention, the link of the last call in any list is set to zero. Links to other list calls take the form of the index of the words in the 'TAG' and 'LINK' arrays. These are therefore declared from 1 upwards to avoid conflict. Both of the arrays are initialised in file 'SKIMPA' to set up the tags and links for built-in routine names such as 'READ SYMBOL' etc. (The name dictionary is also initialised for the same reason.)

A function 'NEWTAG', in file 'SKIMPD', takes the cell indicated by 'TAGASL' from the available space list whenever a cell is required. 'TAGASL' is then moved down the list to the next one. If ever 'TAGASL' is zero, the compiler stops with a suitable monitor. The result of the 'NEWTAG' function is the index of the cell in 'TAG' and 'LINK'. A common sequence of instructions might take the form:

```
CELL = NEWTAG
TAG (CELL) = ....
LINK (CELL) = ....
```

A function named 'RETURNTAG',also in file 'SKIMPD', has the reverse effect. It takes a single parameter which is the index of the cell to be returned to the available space list. The result of the function is the value in the 'LINK' word of the cell returned. This is purely a matter of programming convenience.

The two parallel arrays which form the hash dictionary of names and pointers to tags are named 'NAMEDLINK' and 'TAGLINK' and point to the 'NAMED' and 'TAG/LINK' arrays respectively.

Routines 'PUSHTAG' and 'POPTAGS', in file 'SKIMPD', push and pop tags from 'TAGLINK' as required. 'PUSHTAG' is called with the identification number of the name whose tag is being pushed together with it's form, type, etc. as parameters. A subsidiary function of this routine is to check whether the same name has already been declared at this level and monitor the fault if so. 'POPTAGS' is called at the %END of each routine or function to pop all the tags for names declared at that level, monitoring them if the '/TAGS' option was set. The array which contains pointers to lists of names declared at each level is named 'NAMELIST'.

Chapter 7

Compilation of Expressions

Although all the registers in the object machine can be used as accumulators the machine is regarded, for the sake of simplicity, as a single-accummulator machine and all expressions are evaluated in the one register 'ACC'. This avoids the necessity of having register to register operations which would be needed for a true multi-accumulator machine and avoids the extra complexity this introduces for expression compilation.

A 'tree' algorithm is used in the SKIMP compiler. Many other methods are possible but this one produces reasonably good object code without too many redundant 'LOAD' and 'STR' operations. It is also straightforward to comprehend. Expressions are regarded as trees with operators at the nodes and operands as leaves. For example, the expression:

```
I * J - K / (L + M)
```

has the tree form shown in figure 7.1.

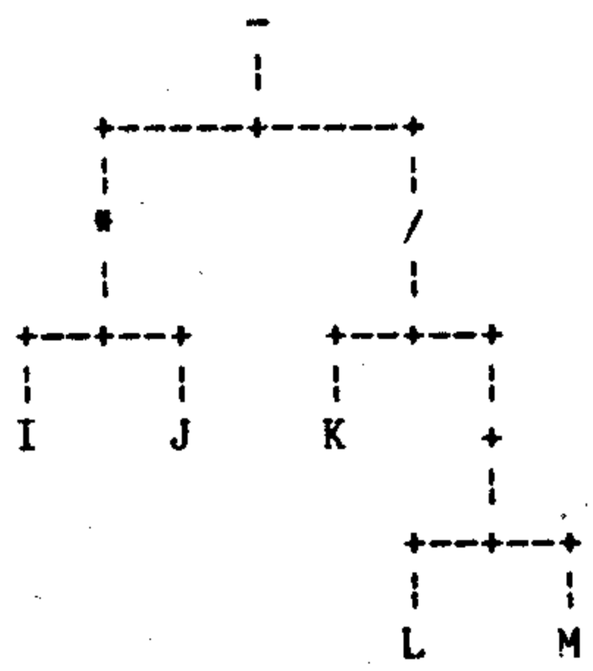


figure 7.1

The algorithm examines the 'shape' of the tree in order to decide the best order of evaluation of the expression. The commutativity of the operators, that is, whether 'X?Y' is the same as 'Y?X' or not, is taken into account and also whether the sub-trees of operator nodes are leaves or not. When the sub-tree is a leaf, that operand is capable of being immediately loaded into 'ACC' or can operate immediately on 'ACC'.

When the sub-tree is not a leaf i.e. a further tree, the algorithm works recursively to compile it. The algorithm can be tabulated for the various combinations possible, the whole thing being regarded as 'evaluate tree' whose object is to dump code which puts the result in 'ACC'. The table is shown in figure 7.2.

This algorithm would not be acceptable if the order of evaluation was defined as part of the language, for instance strictly left to right. As an example, the object code is improved by evaluating the second operand first for non-commutative operators. To illustrate the working of the algorithm, consider the expression which was used above:

```
I * J - K / (L + M)
```

left branch	operator	right branch	actions
leaf 1	all	leaf 2	LOAD,ACC,leaf 1 (operation),ACC,leaf 2
tree	all	leaf	evaluate tree (operation),ACC,leaf
leaf	comm.	tree	evaluate tree (operation),ACC,leaf
leaf	non-comm.	tree	evaluate tree STR,ACC,(work) LOAD,ACC,leaf (operation),ACC,(work)
tree 1	all	tree 2	evaluate tree 2 STR,ACC,(work) evaluate tree 1 (operation),ACC,(work)

figure 7.2

The steps, with indentation for recursive depth are shown in below. The 'work' locations used for partial results are in the local name space for the current routine as was described in chapter 5 in connection with array declarations.

```
evaluate tree [ I*J-K/(L+M) ]
   tree [ I*J ]   all [ - ]   tree [  K/(L+M) ]
   evaluate tree [ K/(L+M) ]
      leaf [ K ]   non-comm [ / ]   tree [ L+M ]
      evaluate tree [ L+M ]
         leaf [ L ]   all [ + ]   leaf [ M ]
         LOAD,ACC, L
         ADD,ACC, M
         STR,ACC, work 1
         LOAD,ACC, K
         DIV,ACC, work 1
      STR,ACC, work 1
      evaluate tree [ I*J ]
         leaf [ I ]   all [ * ]   leaf [ J ]
         LOAD,ACC, I
         MLT,ACC, J
      SUB,ACC, work 2
```

The case of array elements and function calls as operands must be considered in more detail. These cannot be regarded as leaves of the tree since they involve further computation which needs register 'ACC'. They are therefore treated as trees from the point of view of the algorithm. Array element accessing puts the value of the element in 'ACC' and the result of functions is also left in 'ACC'. This satisfies the requirements of 'evaluate tree'.

Variables of type %INTEGERNAME i.e. parameters, can still be regarded as leaves by making use of another index register, 'WK', for the indirect

reference. Constants are leaves of trees and, in general, are accessed from the table of constants following the object code using register 'COT' as the base. The use of a location in the constant table is avoided if a 'LOAD' of the constant is required and the value is less than 65536. This is because the 'LDA' instruction can be used with the index register set to zero. For example,an instruction to 'LOAD' the value 1234 would be:

```
LDA, ACC, , 1234
```

The tree representation is that of Reverse Polish form together with pointers. In Reverse Polish notation the operator follows the operands. In general,

```
operand 1  operator  operand 2
```

becomes:

```
operand 1  operand 2  operator
```

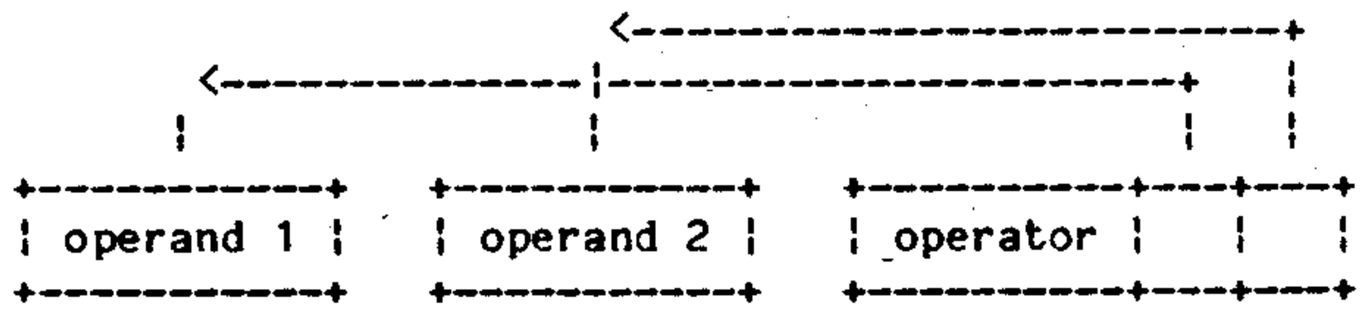
and the same transformation takes place for the operands when they are expressions. For example, the infix expression:

```
I * (J + K) /  L - M
```

becomes:

```
I J K + * L /  M -
```

The extra pointers that were mentioned come after each operator and correspond to the branches of the tree. In general the form is:



When an operand is a further sub-expression, the pointer points to the operator in it's repesentation. Otherwise, the pointer points to the first word of a pair of words which describe the operand. The first of the pair is a type number and the second is consequent on the value of the first. The type values are:

↑ operators

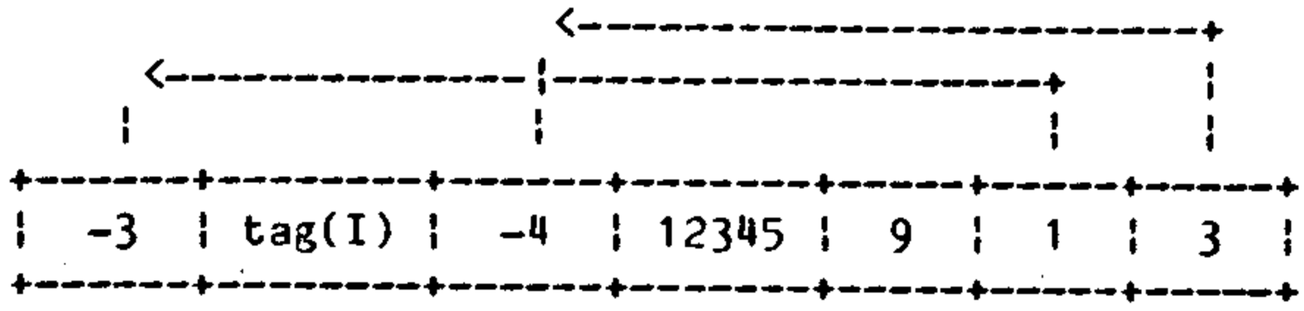
-1	:	function call
-2	:	array element
leaves — -3	:	scalar variable
-4	:	constant

They are negative to distinguish them from values which denote operators and hence sub-trees. The values for operators are their alternative numbers from phrase <OP> (with two additional values beyond for the unary operators '-' and '\') and are thus always positive. Values greater than or equal to -2 cause the algorithm to be called recursively, a requirement which was mentioned above. For types -1 and -2, the second word of the pair contains an analysis record pointer which will allow the function call or array element accession object code to be generated when it is required. For type -3, the second word is the

tag of the variable and for type -4, the second word is the value of the constant. As an example, consider the expression:

I + 12345

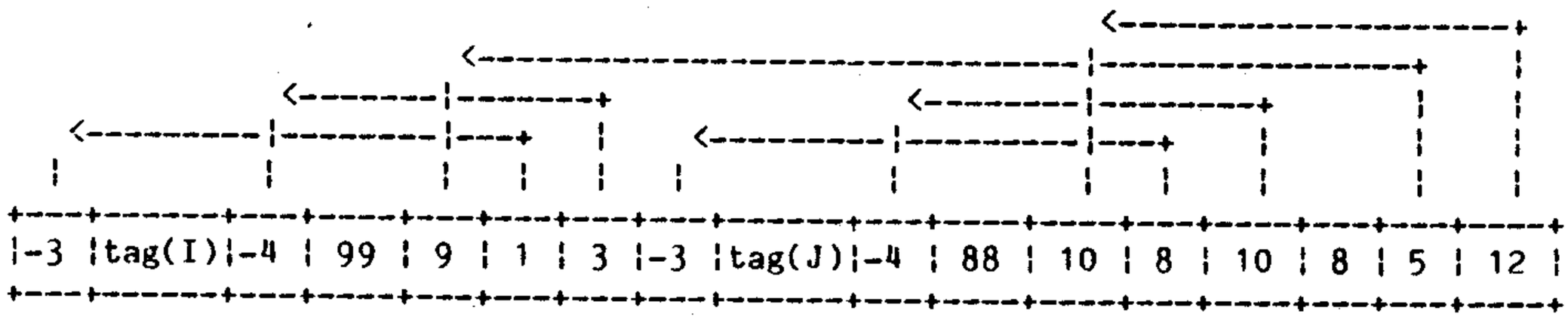
The representation of this is:



An example to show sub-trees might be:

(I + 99) * (J - 88)

The representation of this is:



The technique used to convert infix expressions to this form combines a method of converting them to Reverse Polish with a scheme of pseudo-evaluation to calculate the values of the pointers. The conversion to Reverse Polish is shown diagrammatically in figure 7.3. A pushdown stack is used for operators to take account of their precedence. This delays their storage until the appropriate place in the output. A new stack is declared at each recursive level to deal with bracketted sub-expressions. The conversion of the expression:

I + J * (K - L) / M

would proceed as shown in figure 7.4. The dotted box indicates the recursive call to deal with '(K-L)'.

When operators and operands are 'stored' according to the flow diagram, a pseudo-evaluation takes place. To explain what is meant by 'pseudo'-evaluation, first consider an 'evaluation' of a reverse Polish expression. A pushdown stack is used to retain operand values and partial results during the evaluation. Proceeding from left to right, whenever an operand is encountered it's value is pushed down. Whenever a (binary) operator is encountered, the top two items on the stack are popped and the operation between them is performed. The result is then pushed back onto the stack. (For unary operators, just one item is popped from the stack). Proceeding in this way, when the whole of the reverse Polish string has been dealt with, the value of the whole expression is what remains in the stack. For example:

10 + 8 * (6 - 4) / 2

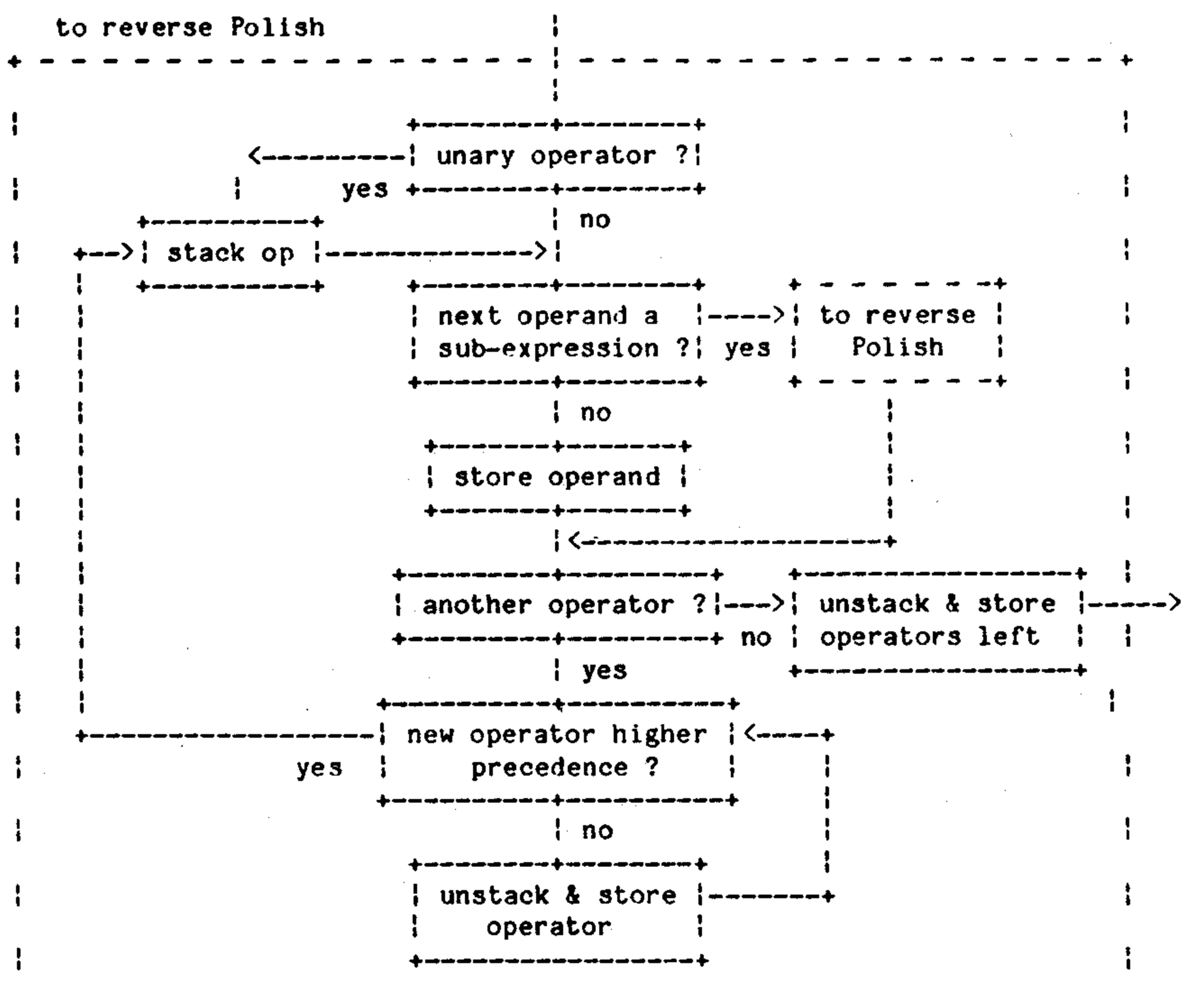


figure 7.3

input	op stack	action	cumulative output
I	empty	store I	I
+	+	stack +	I
J	+	store J	IJ
*	+*	stack *	IJ
K	empty	store K	IJK
-	-	stack -	IJK
L	-	store L	IJKL
	empty	unstack & store -	IJKL-
/	+	unstack & store *	IJKL-*
	+/	stack /	
M	+	store M	IJKL-*M
	+	unstack & store /	IJKL-*M/
	empty	unstack & store +	IJKL-*M/+

figure 7.4

which in reverse Polish is:

10 8 6 4 - * 2 / +

The stages of evaluation in a pushdown stack are:

				4					
		6	6	2		2			
	8	8	8	8	16	16	8		
10	10	10	10	10	10	10	10	18	

In a 'pseudo'-evaluation, it is not the values of operands that are pushed down but some other property. In this case, it is positions in the representation of the tree. In place of evaluation when operators occur, the positions of operands are popped up and it is these values which are appended to the tree representation. The position of the operator is then pushed down in their place. Consider the example used above:

(I + 99) * (J - 88)

which in reverse Polish is:

I 99 + J 88 - *

The successive contents of the pseudo-evaluation stack will be:

				10		
	3		8	8	12	
1	1	5	5	5	5	15

The representations of the trees corresponding to all the expressions in a program can be monitored by using the parameter '/EXPR' when calling the compiler. Examples can be found in appendix 1.

Implementation Details

The routine concerned with the compilation of expressions is named 'EXPR' and can be found in file 'SKIMPE'. Within 'EXPR', subroutine 'TO TREE' produces the linear tree representation using a further subroutine 'PS EVAL' for the pseudo-evaluation part. It also has the subsidiary function of checking that all the names used in the expression have been declared and are of an appropriate type, for instance, not routine names.

The tree code generating algorithm is implemented by routine 'EVALUATE' which makes use of a subroutine 'OPN' to dump primitive operations i.e. routine calls, array and variable accesses and constant handling.

A minor complication of routine 'EXPR' deals with the expressions involved with simple comparisons in conditional statements. This is because a comparison such as:

expr1 <COMP> expr2

is compiled as an evaluation of 'expr1 - expr2' and then the value tested against zero. An external flag named 'CONDFLAG' is set in these circumstances and its effect is form a tree with branches representing each expression the topmost node being a '-'. The algorithm then proceeds as normal.

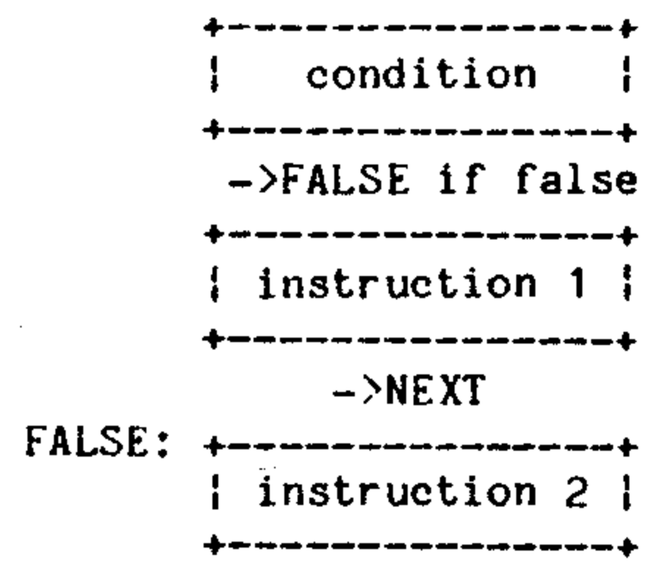
Chapter 8

Conditional Statements

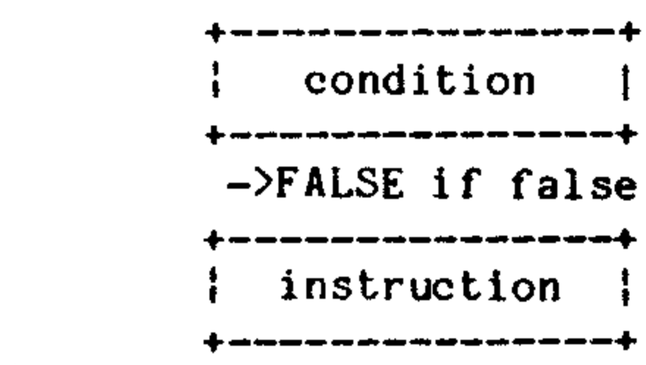
The general form of conditional statement is:

%IF condition %THEN instruction 1 %ELSE instruction 2

where the 'instructions' include the case of a group of statements surrounded by %START and %FINISH. In compiling this, the aim is to produce object code of the form:



where the box containing 'condition' implies object code instructions which evaluate the truth or falsity of the condition for subsequent testing. When the %ELSE clause is absent, this simplifies to the form:



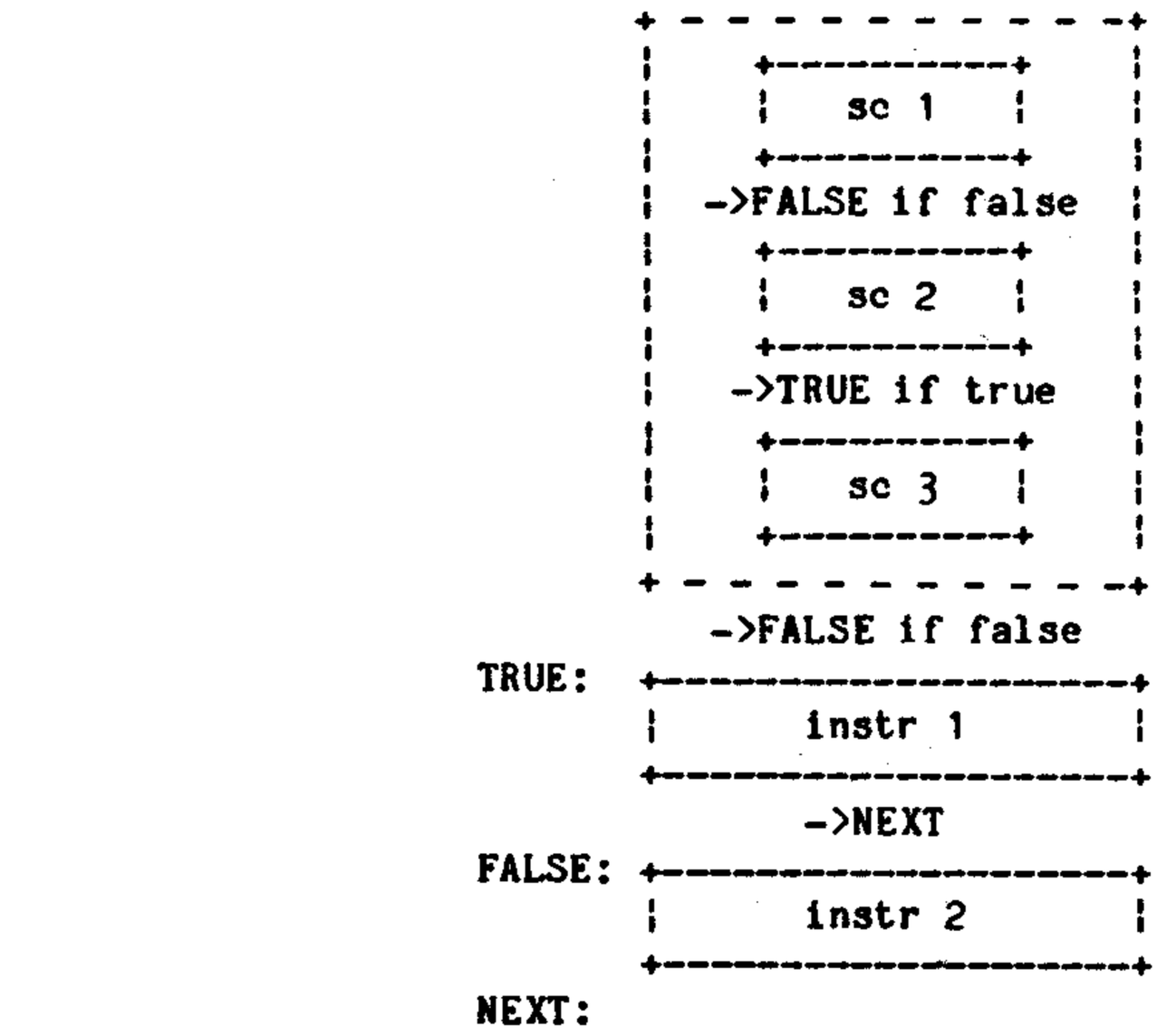
In addition, since the truth or falsity of the whole condition can sometimes be determined without evaluating all the simple comparisons, there may be intermediate jumps out from the middle of the condition to the consequent instructions. For example, consider:

%IF sc1 %AND (sc2 %OR sc3) %THEN instr 1 %ELSE instr 2

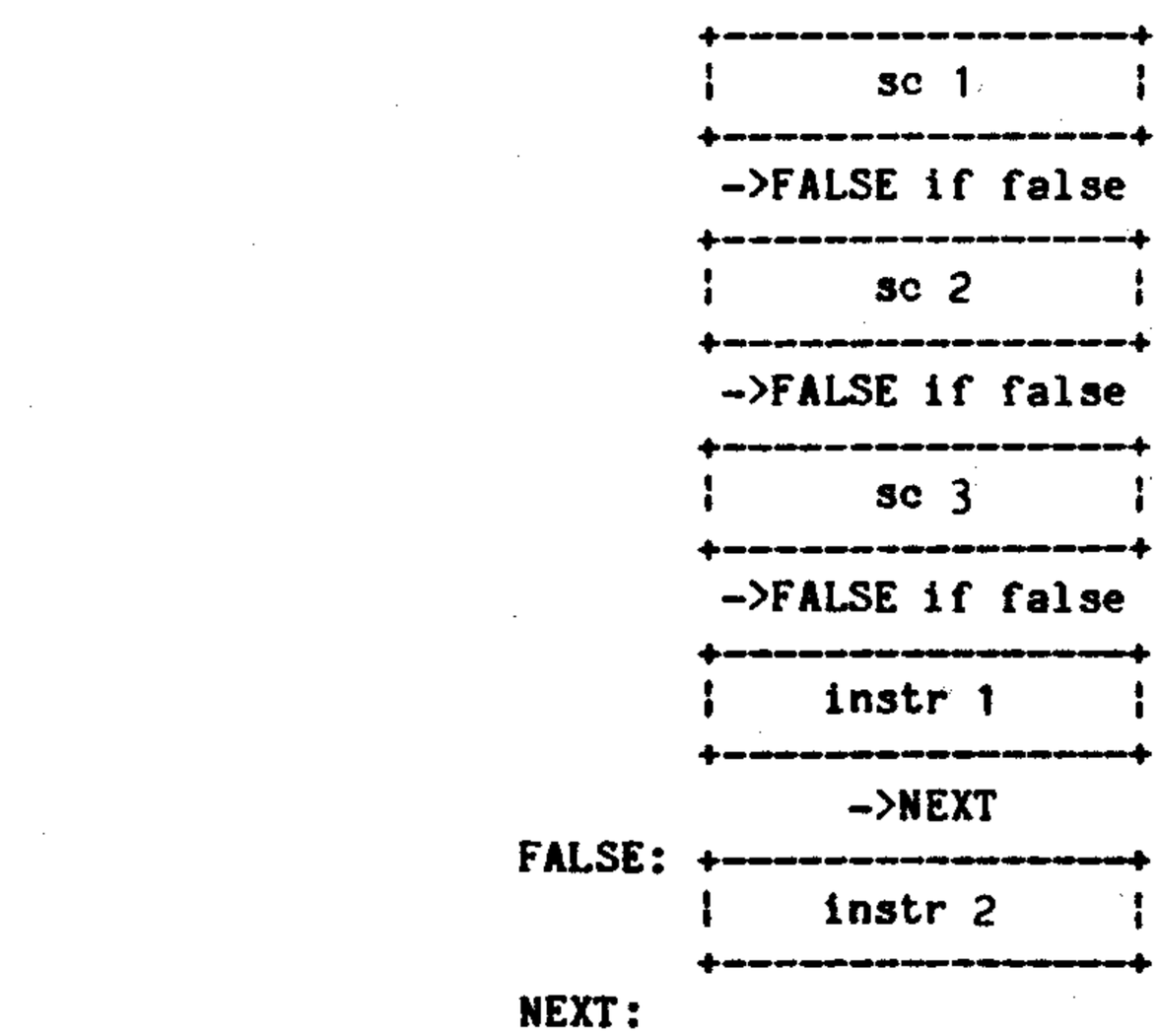
This would produce code of the form shown in the next diagram. Analagously with arithmetic expressions, however, the language may define conditions to be evaluated 'in toto' as Boolean expressions which would preclude jumping out. Fortunately, SKIMP does not require this.

A number of observations can be made with a view to producing a straightforward simple compilation algorithm. The first concerns whether the jump following each simple comparison in the condition should be on true or false. Consider the case:

%IF sc1 %AND sc2 %AND sc3 %THEN instr 1 %ELSE instr 2



The code in this second case should be of the form:



Next consider:

%IF sc1 %OR sc2 %OR sc3 %THEN instr 1 %ELSE instr 2

This should produce the code in the diagram below. It will be apparent that if an %AND follows the simple comparison, the jump is on false and if an %OR follows, the jump is on true. To make this rule consistent throughout, since the jump following the last simple comparison is always on false we can imagine an %AND following the last simple comparison:

%IF (condition) %AND %THEN instr 1 %ELSE instr 2

```
                +---------------+
                |      sc 1     |
                +---------------+
                  ->TRUE if true
                +---------------+
                |      sc 2     |
                +---------------+
                  ->TRUE if true
                +---------------+
                |      sc 3     |
                +---------------+
                 ->FALSE if false
TRUE:           +---------------+
                |     instr 1   |
                +---------------+
                      ->NEXT
FALSE:          +---------------+
                |     instr 2   |
                +---------------+
NEXT:
```

The next observation concerns the destination of the jump. This is not always to one of the unconditional instructions. It can be to a later simple comparison. For example:

```
%IF ((sc1 %AND sc2) %OR sc3) %AND sc4 %THEN instr 1 %ELSE instr 2
```

This should produce code of the form:

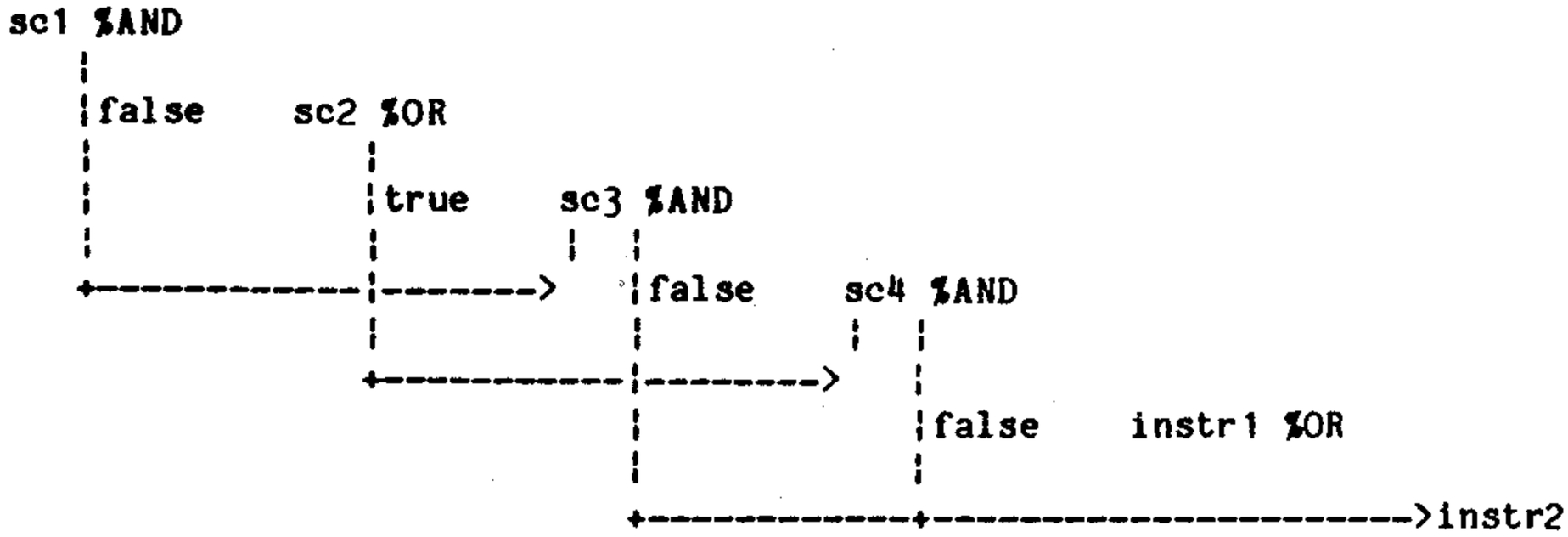
```
                +--------------+
                |     sc 1     |
                +--------------+
                ->FALSE1 if false
                +--------------+
                |     sc 2     |
                +--------------+
                 ->TRUE if true
FALSE1:         +--------------+
                |     sc 3     |
                +--------------+
                ->FALSE2 if false
TRUE:           +--------------+
                |     sc 4     |
                +--------------+
                ->FALSE2 if false
                +--------------+
                |    instr 1   |
                +--------------+
                     ->NEXT
FALSE2:         +--------------+
                |    instr 2   |
                +--------------+
NEXT:
```

For a simple comparison followed by an %AND, the jump is to the simple comparison following the next %OR at an outer bracketting level. For a simple comparison followed by an %OR, the jump is to the simple comparison following

the next %AND at an outer bracketting level. This can again be made consistent if the conditional statement is regarded as of the form:

%IF ((condition) %AND %THEN instr 1) %OR %ELSE instr 2

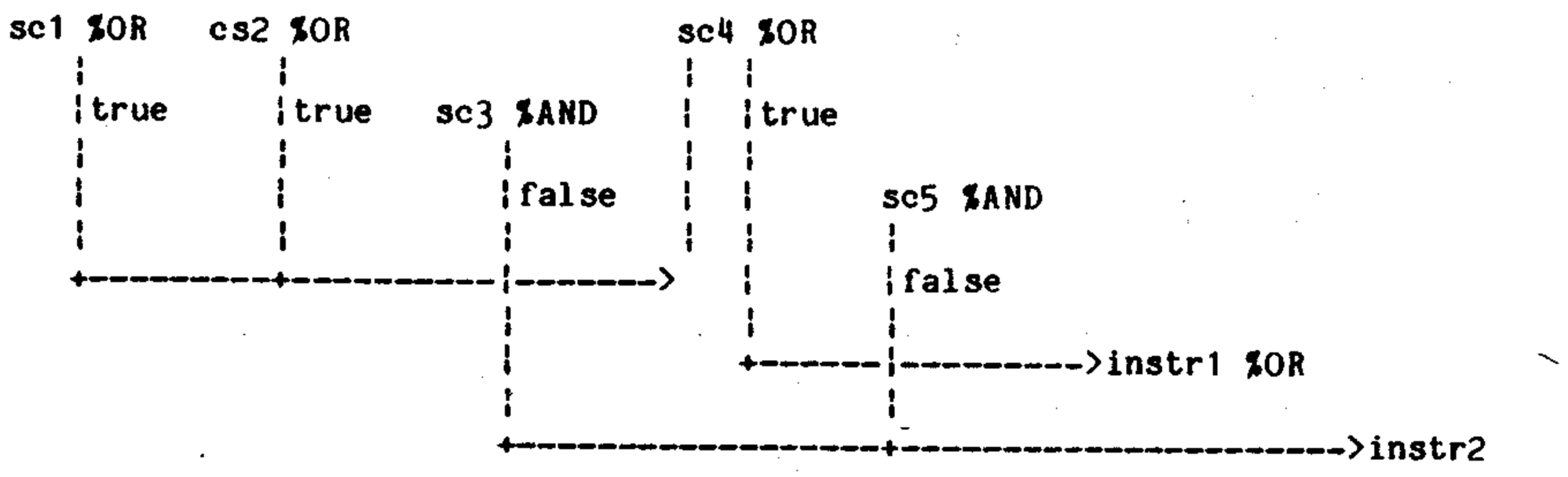
To demonstrate this, we could redraw the example above showing bracketting levels with each simple comparison (or instruction) at the level of the following %AND or %OR:



A further example might be:

%IF (sc1 %OR sc2 %OR sc3) %AND (sc4 %OR sc5) %THEN instr1 %ELSE instr2

The organisation required is:



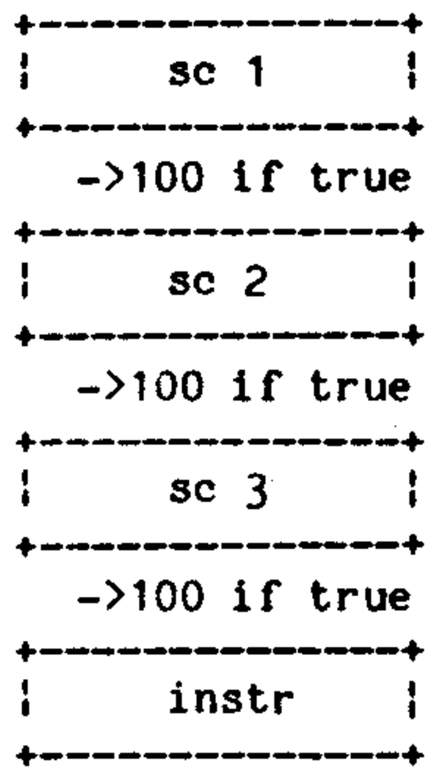
In compiling conditional statements, the plan of action is to make several passes over the condition, working out the structure required step by step and finally, in a last pass, generating the object code. Five parallel arrays are used, each with one position per simple comparison. The first pass traverses the analysis record and locates the positions of all the simple comparisons. Simultaneously, the nesting levels and truth or falsity of the branches is determined and also stored. The next pass goes over this information to determine the destinations of the branches using the rules of thumb described above. For those simple comparisons which are the destination of branches, the next pass allocates 'private' labels to which branches can be compiled using the normal methods used elsewhere in the compiler (see chapter 9). Private labels are just values allocated upwards from 10000, which allows them to be distinguished from ordinary labels. For the sake of consistency, the instructions made conditional are also allocated positions in the arrays and are regarded as being at nesting levels -1 and -2. The rules can then be applied uniformly throughout.

When the instruction made conditional is itself a branch, the object is optimised slightly to allow branches to be made direct to the required place

from within the condition instead of having a branch to a branch. For example:

%IF sc1 %OR sc2 %OR sc3 %THEN ->100 %ELSE instr

The code produced takes the form:

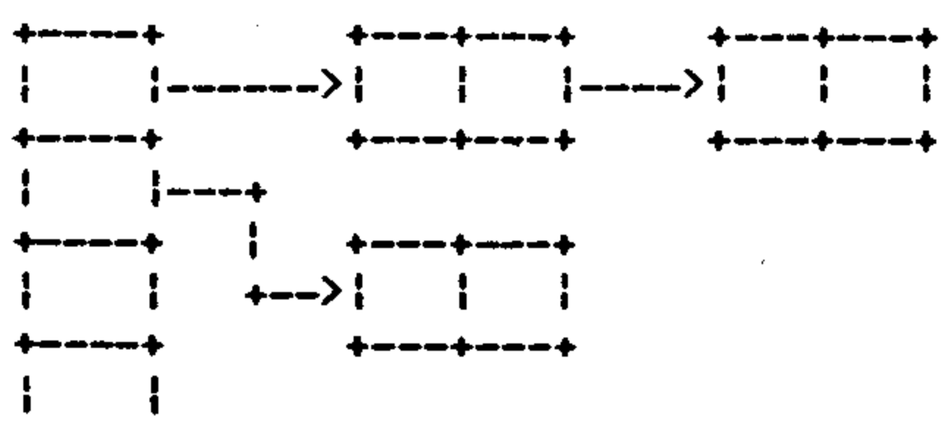


This is easily achieved by using the required label instead of a private label to label the conditional branch instruction in the branch array. The only other modification is to ensure that the branch on the previous i.e.last, simple comparison is on true rather than false and to the label specified. This should be clearer when the implementation details are discussed.

The contents of the five arrays can be displayed during compilation by using the parameter '/COND'. Examples will be found in appendix 1.

%START-%FINISH Groups

When either of the instructions made conditional is a %START-%FINISH group, the destinations of the final branches are not immediately known. This can only be done after the %FINISH has appeared. Information therefore has to be preserved during the compilation of the group. Since the group may itself contain further conditional statements and hence nested %START-%FINISH groups, a pushdown list is used for the purpose. Furthermore, since routine bodies may also appear anywhere and be nested, a list will potentially be required for each textual level of the program. An array of headcells to lists is used as previously to achieve this:



Two items of information need to be stored for each group. Firstly, a marker which distinguishes between groups before and those after %ELSE keywords and secondly, a label number assigned to the destination of branches around the group. This will be a private label.

Implementation Details

The code generation for conditions is done by integerfn 'COND' in file 'SKIMPC'. This is called from the section of 'STATEMENT' in file 'SKIMPB' which deals with conditional statements. It is at this latter place that the code for the instructions made conditional is generated by calling routine 'INSTR'. In order to deal with the branch instruction optimisation, two parameters in addition to the parameter which indicates where the condition starts in the analysis record are passed in. These give the label numbers of the branches or -1 if either of them is a non-branch instruction. These are then used instead of private labels as will be seen below. The result of this function is the private label number of the label to be put in front of the %ELSE clause.

%START-%FINISH groups are dealt with after the condition code has been generated and also where %FINISH statements are processed. Two subroutines are used to maintain the pushdown lists. These are 'PUSHSTART' and 'POPSTART' and can be found in file 'SKIMPD'.

The five parallel arrays within function 'COND' are named 'TESTP', 'LEVEL', 'ANDOR', 'BRANCH' and 'LABEL'. They hold, respectively, the analysis record positions of the simple comparisons (or 'tests'), the nesting levels of the simple comparisons, the truth or falsity of the jumps(1 for false and 2 for true), the destination of the jumps (in terms of column position in the table) and the labels assigned to simple comparisons. For example, consider:

```
%IF (sc1 %OR sc2 %OR sc3) %AND (sc4 %OR sc5) %THEN instr1 %ELSE instr2
```

The contents of the arrays will be:

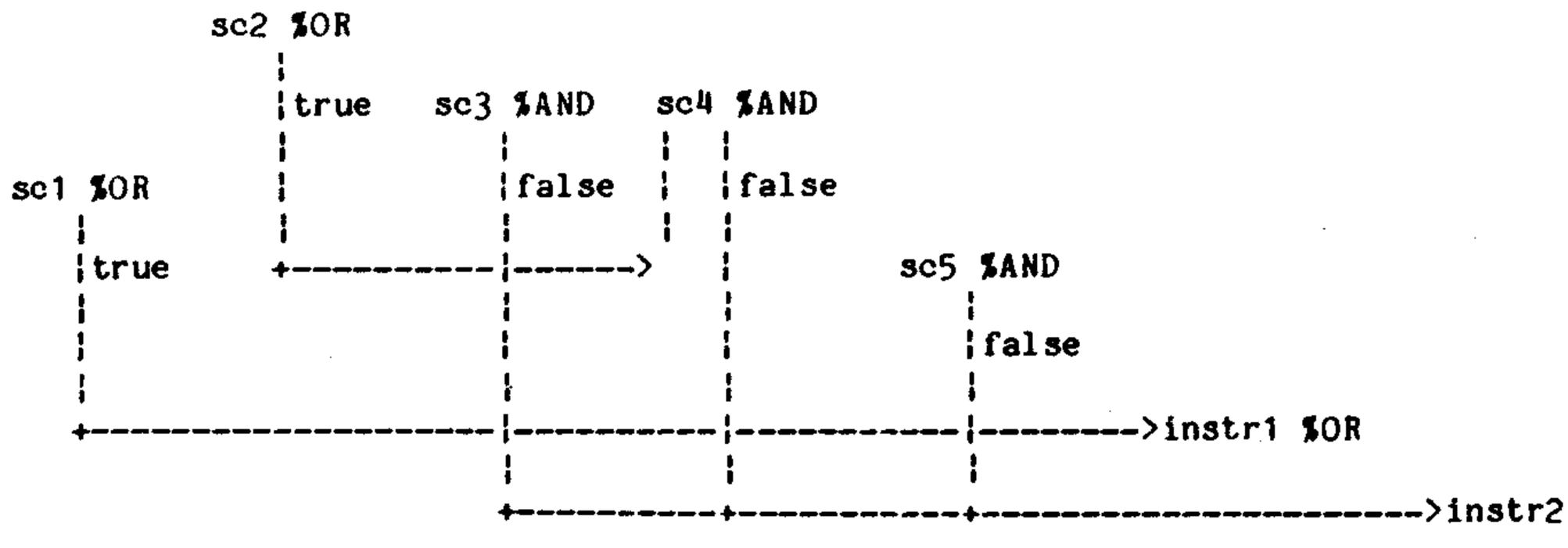
	1	2	3	4	5	6	7
TESTP	ap1	ap2	ap3	ap4	ap5		
LEVEL	1	1	0	1	-1	-2	
ANDOR	2	2	1	2	1	2	
BRANCH	4	4	7	6	7		
LABEL	-1	-1	-1	10000	-1	10002	10001

where the 'ap?' represent analysis record positions. The first pass over the statement is performed by the subroutine 'PROCESSCOND' and fills in rows 'TESTP', 'LEVEL' and 'ANDOR'. It works recursively following the recursive definition of conditions. Subsequently, two extra columns are filled in with -1 and -2 for the pseudo-%AND and the pseudo-%OR. On the next pass the row 'BRANCH'is filled in and following that labels are assigned as necessary. (The -1 indicates that no jump takes place to this simple comparison and hence no label is required. The final pass generates the object code by calling 'EXPR' with the 'CONDFLAG' set as described in the previous chapter to evaluate LHS minus RHS, and dumping the appropriate intermediate jump instructions.

Another example may help to clarify the picture further . Consider the case:

```
%IF sc1 %OR ((sc2 %OR sc3) %AND sc4 %AND sc5) %THEN instr1 %ELSE instr2
```

The structure of the statement is:



The table for this statement will be:

	1	2	3	4	5	6	7
TESTP	ap1	ap2	ap3	ap4	ap5		
LEVEL	0	2	1	1	-1	-2	
ANDOR	2	2	1	1	1	2	
BRANCH	6	4	7	7	7		
LABEL	-1	-1	-1	10004	-1	10003	10005

As a final example, consider a statement containing conditional branches:

```
%IF sc1 %OR sc2 %OR sc3 %THEN ->88 %ELSE ->99
```

The table will be:

	1	2	3	4	5
TESTP	ap1	ap2	ap3		
LEVEL	0	0	-1	-2	
ANDOR	2	2	2	2	
BRANCH	4	4	4		
LABEL	-1	-1	-1	88	99

Chapter 9

Labels and Jumps

As the SKIMP compiler is a one-pass compiler, jumps to labels not yet encountered cannot be completely generated. Backward jumps are, of course, no problem. For the forward jumps, all that can be done is to leave a hole in the branch instruction dumped which can be filled in at some later stage. In this case, the assembler which forms the first stage of the 'SKIMPAI' interpreter system has the job to do. In order for this to be possible, information must be recorded somewhere concerning the positions of the holes to be filled in and what to fill them in with. The method adopted is to use the holes themselves as links in lists which relate to each label and for the compiler to dump 'FILL' assembler directives which are really the headcells to these lists and which also include the value to be filled in. The 'FILL' directives are dumped whenever a label which has been previously referenced is encountered. An example should make this clear. Consider the program:

```
          . . . .
          ->88
          . . . .
          ->99
          . . . .
          ->99
          . . . .
    99:   . . . .
          . . . .
```

The object code dumped will take the form:

```
. .       . . . .
                  +-------+
12$       B,,,|   0   |
                  +-------+
. .       . . . .
                  +-------+
34$       B,,,|  12   |
                  +-------+
. .       . . . .
                  +-------+
56$       B,,,|  34   |
                  +-------+
. .       . . . .

78$       FILL,,56,78

. .       . . . .
```

As labels are local to each level of routine nesting, once again separate lists must be used for each level. Each cell of a list in this case contains three items of information, the label number, a flag and an address:

label	flag	address
(16)	(1)	(15)

The flag indicates whether the address is that of the latest forward reference to the label i.e. the current top of the forward reference list, or the actual address of the label which will be needed for any backward references. The value 1 indicates the former and 0 the latter. When additional forward references appear therefore, the address is pushed down 'through' this cell.

A cell of this form is created whenever a label is encountered for the first time - either as a label or as a jump. The list for the current textual level is first searched and a new cell added if no existing cell refers to that label. The flag allows two vital checks to be made. Firstly that the label has not occurred before (as a label) and secondly at the end of the routine that the label has in fact appeared.

Routine calls use the 'BAL' instruction and are always backward references since SKIMP does not allow routine %SPECs. The address of the routine body is, in other words, already known when any routine calls are made. SKIMP allows routines to be placed anywhere in a program, in particular at the head of surrounding routines. Jumps around these routine bodies must therefore be dumped. These are forward references but are simple in that only one backward reference will occur. They are therefore treated specially without having to invent a private label.

Calls on the built-in input-output routines are also treated specially. Since these routines are implemented in the 'SKIMPAI' assembler/interpreter, the 'BAL' instructions dumped contain the marker 'EXT' and a number which refers to the routine required. Were external references part of the language, the names themselves would have to appear in the object file and likewise for any external variables in the program. A linker would then resolve the references between programs.

Implementation Details

The routines which accomplish the functions decribed above are named 'FILL LABEL' and 'FILL BRANCH', in file 'SKIMPD'. A function 'GET LABEL' extracts a label from the analysis record and checks that it is valid i.e.less than 10000. 'POP LABELS' pops up the cells and checks for missing labels at %END (and %ENDOFPROGRAM).

Appendix 1

Examples of SKIMP Object Code Files

LEX, ANAL Example

```
                    SKIMP COMPILER MKII

FILE: TESTL
OPTIONS: LEX ANAL


%begin

  130

  (1/STATEMENT)    8

    0$              LDA,COT,,0
    1$              LDA,DR1,,0
    2$              LDA,STP,DR1,0


%integerfn r

  136  134  256   82

  (1/STATEMENT)    6    5    6    8  (5/PROC)    2  (6/NAME)    1
   82  (8/FORMAL)    2

    3$              B,,,0
    4$              STR,DR2,STP,0
    5$              LDA,DR2,STP,0
    6$              STR,WK,STP,1
    7$              LDA,STP,STP,0


%integer i,j,k

  136  256   73   44  256   74   44  256   75

  (1/STATEMENT)    5    3  (3/ARRAY)    2    6    8  (6/NAME)    1
   73  (8/NAMES)    1   11   13  (11/NAME)    1   74  (13/NAMES)    1
   16   18  (16/NAME)    1   75  (18/NAMES)    2



i=j+k

  256   73   61  256   74   43  256   75

  (1/STATEMENT)    1    3  (3/INSTR)    1    7    9   10  (7/NAME)    1
   73  (9/ACTUAL)    2  (10/ASSIGN)    1   12  (12/EXPR)    1
   16   17   23  (16/UNARY)    4  (17/OPERAND)    1   20   22
  (20/NAME)    1   74  (22/ACTUAL)    2  (23/EXPRREST)    1   27
   28   34  (27/OP)    9  (28/OPERAND)    1   31   33  (31/NAME)    1
   75  (33/ACTUAL)    2  (34/EXPRREST)    2

    8$              LOAD,ACC,DR2,3
    9$              ADD,ACC,DR2,4
   10$              STR,ACC,DR2,2


%if i>1234 %then %stop

  135  256   73   62  257 1234  145  144
```

```
(1/STATEMENT)    2    5   36   37  (5/COND)    1    8   35  (8/TEST)     1
 12   24   25  (12/EXPR)    1   16   17   23  (16/UNARY)    4
(17/OPERAND)    1   20   22  (20/NAME)    1   73  (22/ACTUAL)    2
(23/EXPRREST)    2  (24/COMP)    6  (25/EXPR)    1   29   30
 34  (29/UNARY)    4  (30/OPERAND)    2   32  (32/CONST)    1
1234  (34/EXPRREST)    2  (35/CONDREST)    3  (36/INSTR)    6
(37/ELSE)    2

 11$            LOAD,ACC,DR2,2
 12$            SUB,ACC,COT,0
 13$            BNG,ACC,,0
 14$            STOP,,,0
 15$            FILL,10000,13,15
```

```
%result=i+4321

  141   61  256   73   43  257 4321

  (1/STATEMENT)    1    3  (3/INSTR)    5    5  (5/EXPR)    1
   9   10   16  (9/UNARY)    4  (10/OPERAND)    1   13   15  (13/NAME)    1
  73  (15/ACTUAL)    2  (16/EXPRREST)    1   20   21   25  (20/OP)    9
 (21/OPERAND)    2   23  (23/CONST)    1 4321  (25/EXPRREST)    2


   15$            LOAD,ACC,DR2,2
   16$            ADD,ACC,COT,1
   17$            LDA,STP,DR2,0
   18$            LOAD,DR2,STP,0
   19$            LOAD,WK,STP,1
   20$            B,,WK,0
```

```
%end

  131

  (1/STATEMENT)    7    3  (3/OFPROG)    2

   21$            FILL,ALLOC,7,5
   21$            STOP,,,0
   22$            FILL,SKIP,3,22
```

```
%endofprogram

  131  139

  (1/STATEMENT)    7    3  (3/OFPROG)    1

   22$            FILL,ALLOC,2,2
   22$            STOP,,,0
   23$            FILL,COT,0,23
   23$            CONST,,,1234
   24$            CONST,,,4321
   25$            FILL,STACK,1,25

    $    0 FAULTS IN PROGRAM
```

EXPR example

SKIMP COMPILER MKII

```
FILE: TESTE
OPTIONS: EXPR

%begin
    0$          LDA,COT,,0
    1$          LDA,DR1,,0
    2$          LDA,STP,DR1,0


%integer i,j,k,l

%integerarray a(1:10)
```

-4* 1

```
    3$          LDA,ACC,,1
    4$          STR,ACC,DR1,6
```

-4* 10

```
    5$          LDA,ACC,,10
    6$          LDA,ACC,ACC,1
    7$          STR,ACC,DR1,7
    8$          SUB,STP,DR1,6
    9$          STR,STP,DR1,8
   10$          ADD,STP,DR1,7


i=j+k
```

+

-3 01010003 | -3 01010004 | 9* 1 3

```
   11$          LOAD,ACC,DR1,3
   12$          ADD,ACC,DR1,4
   13$          STR,ACC,DR1,2


a(j+k)=i*l-j*k
```

*

-3 01010002 | -3 01010005 | 8 1 3 | -3 01010003 | -3 01010004

8 * 8 10 | 10* 5 12

-

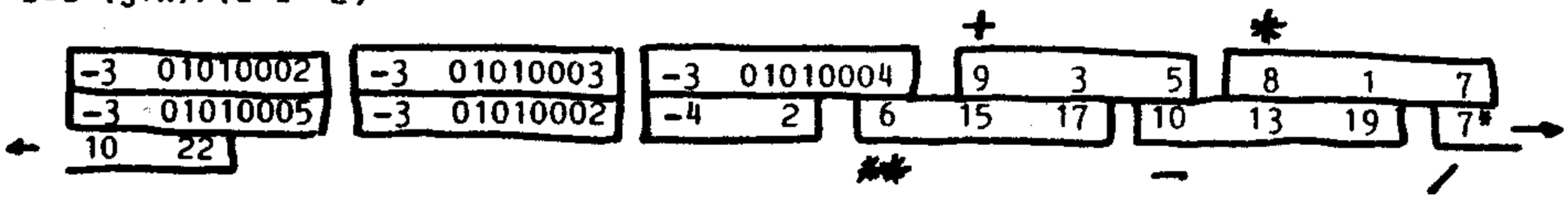
```
   14$          LOAD,ACC,DR1,3
   15$          MLT,ACC,DR1,4
   16$          STR,ACC,DR1,6
   17$          LOAD,ACC,DR1,2
   18$          MLT,ACC,DR1,5
   19$          SUB,ACC,DR1,6
   20$          STR,ACC,DR1,6
```

+

-3 01010003 | -3 01010004 | 9* 1 3

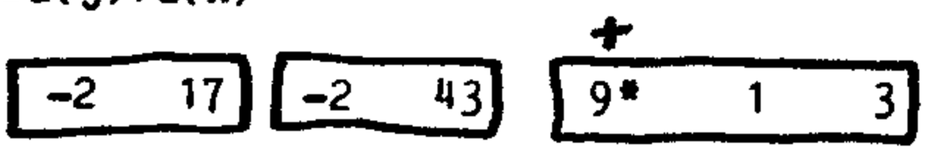
```
   21$          LOAD,ACC,DR1,3
   22$          ADD,ACC,DR1,4
   23$          ADD,ACC,DR1,8
   24$          LOAD,WK,DR1,6
   25$          STR,WK,ACC,0
```

```
i=i*(j+k)/(1-i**2)
```

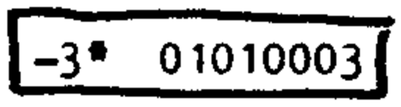



```
   26$          LOAD,ACC,DR1,2
   27$          EXP,ACC,COT,0
   28$          STR,ACC,DR1,6
   29$          LOAD,ACC,DR1,5
   30$          SUB,ACC,DR1,6
   31$          STR,ACC,DR1,6
   32$          LOAD,ACC,DR1,3
   33$          ADD,ACC,DR1,4
   34$          MLT,ACC,DR1,2
   35$          DIV,ACC,DR1,6
   36$          STR,ACC,DR1,2
```

```
i=a(j)+a(k)
```




```
   37$          LOAD,ACC,DR1,4
   38$          ADD,ACC,DR1,8
   39$          LOAD,ACC,ACC,0
   40$          STR,ACC,DR1,6
```




```
   41$          LOAD,ACC,DR1,3
   42$          ADD,ACC,DR1,8
   43$          LOAD,ACC,ACC,0
   44$          ADD,ACC,DR1,6
   45$          STR,ACC,DR1,2


$endofprogram
   46$          FILL,ALLOC,2,9
   46$          STOP,,,0
   47$          FILL,COT,0,47
   47$          CONST,,,2
   48$          FILL,STACK,1,48

     $    0 FAULTS IN PROGRAM
```

TAGS Example

```
                    SKIMP COMPILER MKII

FILE: TESTT
OPTIONS: TAGS


%begin
    0$              LDA,COT,,0
    1$              LDA,DR1,,0
    2$              LDA,STP,DR1,0


%routine a(%integer i,j,k)
    3$              B,,,0
    4$              STR,DR2,STP,0
    5$              LDA,DR2,STP,0
    6$              STR,WK,STP,1
    7$              LDA,STP,STP,0


%end
    8$              FILL,ALLOC,7,5

 75   K             01020004
 74   J             01020003
 73   I             01020002

    8$              LDA,STP,DR2,0
    9$              LOAD,DR2,STP,0
   10$              LOAD,WK,STP,1
   11$              B,,WK,0
   12$              FILL,SKIP,3,12


%integerfn b(%integername 1)
   12$              B,,,0
   13$              STR,DR2,STP,0
   14$              LDA,DR2,STP,0
   15$              STR,WK,STP,1
   16$              LDA,STP,STP,0


%routine c(%integerarrayname m,n)
   17$              B,,,0
   18$              STR,DR3,STP,0
   19$              LDA,DR3,STP,0
   20$              STR,WK,STP,1
   21$              LDA,STP,STP,0


%end
   22$              FILL,ALLOC,21,4

 78   N             31130003
 77   M             31130002
```

```
   22$              LDA,STP,DR3,0
   23$              LOAD,DR3,STP,0
   24$              LOAD,WK,STP,1
   25$              B,,WK,0
   26$              FILL,SKIP,17,26


%end
   26$              FILL,ALLOC,16,3

 67    C           40220012
                       31130002
                       31130003
 76    L           11020002

   26$              STOP,,,0
   27$              FILL,SKIP,12,27


%integer i,j


%endofprogram
   27$              FILL,ALLOC,2,4

 74    J           01010003
 73    I           01010002
 66    B           4111000D
                       11020002
 65    A           40310004
                   01020002
                   01020003
                   01020004

27$              STOP,,,0
28$              FILL,COT,0,28
28$              FILL,STACK,1,28

  $    0 FAULTS IN PROGRAM
```

Basic Example

```
                    SKIMP COMPILER MKII

FILE: HANOI
OPTIONS:


%begin
    0$              LDA,COT,,0
    1$              LDA,DR1,,0
    2$              LDA,STP,DR1,0


%routine hanoi(%integer n,p1,p2)
    3$              B,,,0
    4$              STR,DR2,STP,0
    5$              LDA,DR2,STP,0
    6$              STR,WK,STP,1
    7$              LDA,STP,STP,0


%if n>0 %then %start
    8$              LOAD,ACC,DR2,2
    9$              BNG,ACC,,0


hanoi(n-1,p1,6-p1-p2)
   10$              LOAD,ACC,DR2,2
   11$              SUB,ACC,COT,0
   12$              STR,ACC,STP,2
   13$              LOAD,ACC,DR2,3
   14$              STR,ACC,STP,3
   15$              LDA,ACC,,6
   16$              SUB,ACC,DR2,3
   17$              SUB,ACC,DR2,4
   18$              STR,ACC,STP,4
   19$              BAL,WK,,4


write(p1,1) ;
   20$              LOAD,ACC,DR2,3
   21$              STR,ACC,STP,2
   22$              LDA,ACC,,1
   23$              STR,ACC,STP,3
   24$              BAL,WK,EXT,11


write(p2,1) ;
   25$              LOAD,ACC,DR2,4
   26$              STR,ACC,STP,2
   27$              LDA,ACC,,1
   28$              STR,ACC,STP,3
   29$              BAL,WK,EXT,11


newline
   30$              BAL,WK,EXT,7
```

```
hanoi(n-1,6-p1-p2,p2)
    31$             LOAD,ACC,DR2,2
    32$             SUB,ACC,COT,0
    33$             STR,ACC,STP,2
    34$             LDA,ACC,,6
    35$             SUB,ACC,DR2,3
    36$             SUB,ACC,DR2,4
    37$             STR,ACC,STP,3
    38$             LOAD,ACC,DR2,4
    39$             STR,ACC,STP,4
    40$             BAL,WK,,4


%finish
    41$             FILL,10000,9,41


%end
    41$             FILL,ALLOC,7,5
    41$             LDA,STP,DR2,0
    42$             LOAD,DR2,STP,0
    43$             LOAD,WK,STP,1
    44$             B,,WK,0
    45$             FILL,SKIP,3,45


%integer a,b,c


1:read(a)
    45$             LDA,ACC,DR1,2
    46$             STR,ACC,STP,2
    47$             BAL,WK,EXT,10


%if a=0 %then %stop
    48$             LOAD,ACC,DR1,2
    49$             BNZ,ACC,,0
    50$             STOP,,,0
    51$             FILL,10001,49,51


read(b) ;
    51$             LDA,ACC,DR1,3
    52$             STR,ACC,STP,2
    53$             BAL,WK,EXT,10


read(c)
    54$             LDA,ACC,DR1,4
    55$             STR,ACC,STP,2
    56$             BAL,WK,EXT,10
```

```
hanoi(a,b,c)
    57$             LOAD,ACC,DR1,2
    58$             STR,ACC,STP,2
    59$             LOAD,ACC,DR1,3
    60$             STR,ACC,STP,3
    61$             LOAD,ACC,DR1,4
    62$             STR,ACC,STP,4
    63$             BAL,WK,,4


->1
    64$             B,,,45


%endofprogram
    65$             FILL,ALLOC,2,5
    65$             STOP,,,0
    66$             FILL,COT,0,66
    66$             CONST,,,1
    67$             FILL,STACK,1,67

       $    0 FAULTS IN PROGRAM
```

Appendix 2

Possible Extensions to SKIMP

1. a. Identifier labels instead of numerical labels:

```
LOOP:  ->NEXT
NEXT:  ->LOOP
```

b. Switches:

```
%SWITCH  TYPE(1:9)
TYPE(1):  ->TYPE(I*J)
```

2. a. Indefinite loops:

```
%CYCLE
. . . .
%REPEAT
```

b. Incremental loops:

```
%FOR I=1,1,N %CYCLE
. . . .
%REPEAT
```

c. 'While' loops:

```
FRED %WHILE I>0
%WHILE I>0 %CYCLE
. . . .
%REPEAT
```

d. 'Until' loops:

```
FRED %UNTIL I>0
%CYCLE
. . . .
%REPEAT %UNTIL I>0
```

e. %EXIT and %CONTINUE statements.

3. a. Reversed order conditional statements:

```
I=J*K %IF I=0
```

b. 'Unless' conditions:

```
%UNLESS I=J %THEN K=L
```

c. 'else-if' clauses:

```
%IF I=0 %THEN J=0 %ELSE %IF K=0 %THEN L=0
```

d. Multi-sided comparisons:

```
%IF A<B<C<D %THEN E=F
```

4. a. Multi-dimensional arrays:

```
%INTEGERARRAY A(1:9,-1:1),B,C(1:N,1:M*N,0:1)
```

b. Array bound checking:

5. a. Own variables:

```
%OWNINTEGER I,J=123,K=456
```

b. Own arrays:

```
%OWNINTEGERARRAY L(1:10)=1,2,3,4,5,6(5)
```

c. Constant variables and arrays:

```
%CONSTINTEGER I=1234
%CONSTINTEGERARRAY J(1:3)=2,7,4
```

6. Real variables:

```
%REAL X,Y,Z
%REALARRAY A(1:N)
%REALFN RF(%REAL X,%REALNAME Y,%REALARRAYNAME Z)
```

SKIMPAI has, built-in, the real operations:

ADDF, SUBF, MLTF, DIVF, EXPF, NEGF, FLT

7. String variables:

```
%STRING(15) S
%STRING(7)%ARRAY T(1:10)
%STRING(99)%FN SF(%STRING(3) S)
S=S."FRED".SF("JOE").T(I)
```

8. Records:

```
%RECORDFORMAT RF(%INTEGER I,J,%RECORD(RF)%NAME K)
%RECORD(RF) R
R_I=R_J+1
X=R_K_K_I
```

9. a. INTERDATA object code

b. PDP-11 object code

c. VAX 11-780 object code

Appendix 3

The SKIMP Assembler/Interpreter

The object file output from the SKIMP compiler forms the input to the assembler/interpreter. The first stage identifies statements to be assembled by means of the '$' symbol following code addresses. It is the remainder of such lines that are regarded as significant. Any other lines such as compiler monitoring are ignored. This also allows interpreter directives to be inserted in the program by using SKIMP comments as shown:

```
! $ . . . .
```

The interpreter directives available are:

```
MONITOR
TRON
TROFF
```

The first of these, 'MONITOR', when encountered during interpretation prints out the contents of the registers and the stack. Since this may generate a substantial amount of output, this facility should be used sparingly. The second and third turn on and off respectively a trace facility which prints out the value of the program counter for each instruction interpreted. This can also be turned on by means of a '/TR' parameter in the call of 'SKIMPAI'. Tracing may also generate large amounts of output! For example:

```
! $ MONITOR
```

The interpreter will not attempt to interpret programs which failed to compile correctly. Furthermore, the system makes various checks during interpretation in an endeavour to ensure that the program has not gone out of control. These include unassigned variable checking, that data addresses lie within the stack or constant table and that program counter addresses lie within the code area. A limit of 10000 instructions executed is also imposed to catch loops.

To demonstrate the use of these facilities, consider the following program:

```
    %BEGIN
    %INTEGERARRAY A(1:10)
    %INTEGER i
    ! $ TRON
    I=1
1:  A(I)=I*I
    I=I+1
    %IF i<=10 %THEN ->1
    ! $ TROFF
    ! $ MONITOR
    %ENDOFPROGRAM
```

The output from the compiler and the interpreter are shown below:

SKIMP COMPILER MKII

```
FILE: TEST
OPTIONS:


%BEGIN
    0$              LDA,COT,,0
    1$              LDA,DR1,,0
    2$              LDA,STP,DR1,0


%INTEGERARRAY A(1:10)
    3$              LDA,ACC,,1
    4$              STR,ACC,DR1,2
    5$              LDA,ACC,,10
    6$              LDA,ACC,ACC,1
    7$              STR,ACC,DR1,3
    8$              SUB,STP,DR1,2
    9$              STR,STP,DR1,4
   10$              ADD,STP,DR1,3


%INTEGER I


! $ TRON
I=1
   11$              LDA,ACC,,1
   12$              STR,ACC,DR1,5


1: A(I)=I*I
   13$              LOAD,ACC,DR1,5
   14$              MLT,ACC,DR1,5
   15$              STR,ACC,DR1,2
   16$              LOAD,ACC,DR1,5
   17$              ADD,ACC,DR1,4
   18$              LOAD,WK,DR1,2
   19$              STR,WK,ACC,0


I=I+1
   20$              LOAD,ACC,DR1,5
   21$              ADD,ACC,COT,0
   22$              STR,ACC,DR1,5


%IF I<=10 %THEN ->1
   23$              LOAD,ACC,DR1,5
   24$              SUB,ACC,COT,1
   25$              BNG,ACC,,13
```

```
! $ TROFF
! $ MONITOR
%ENDOFPROGRAM
    26$             FILL,ALLOC,2,6
    26$             STOP,,,0
    27$             FILL,COT,0,27
    27$             CONST,,,1
    28$             CONST,,,10
    29$             FILL,STACK,1,29

      $    0 FAULTS IN PROGRAM
```

```
11$   12$   13$   14$   15$   16$   17$   18$   19$   20$
21$   22$   23$   24$   25$   13$   14$   15$   16$   17$
18$   19$   20$   21$   22$   23$   24$   25$   13$   14$
15$   16$   17$   18$   19$   20$   21$   22$   23$   24$
25$   13$   14$   15$   16$   17$   18$   19$   20$   21$
22$   23$   24$   25$   13$   14$   15$   16$   17$   18$
19$   20$   21$   22$   23$   24$   25$   13$   14$   15$
16$   17$   18$   19$   20$   21$   22$   23$   24$   25$
13$   14$   15$   16$   17$   18$   19$   20$   21$   22$
23$   24$   25$   13$   14$   15$   16$   17$   18$   19$
20$   21$   22$   23$   24$   25$   13$   14$   15$   16$
17$   18$   19$   20$   21$   22$   23$   24$   25$   13$
14$   15$   16$   17$   18$   19$   20$   21$   22$   23$
24$   25$
```

```
COT  27
DR1  29
STP  45
ACC  1
WK  100
 29$   ?   ? 100  11  34  11   1   4   9  16  25  36  49  64  81
 100   ?   ?   ?   ?   ?   ?   ?   ?   ?   ?   ?   ?   ?   ?   ?
   ?   ?   ?
```

```
STOPPED AT 26$, 143 INSTRUCTIONS EXECUTED
```

Source file: SKIMPA.IMP Compiled on 25-OCT-1979 at 09:38:57

Computer Science IMP77 Compiler. Version 6.01

```
1  %externalintegerarray a(1:500)
2  ! initialisation for i/o routines
3  %externalbyteintegerarray named(1:1024)=10,'R','E','A','D','S','Y','M',
4+   'B','O','L',10,'N','E','X','T','S','Y','M','B','O','L',10,'S','K',
5+   'I','P','S','Y','M','B','O','L',11,'P','R','I','N','T','S','Y','M',
6+   'B','O','L',5,'S','P','A','C','E',6,'S','P','A','C','E','S',7,'N',
7+   'E','W','L','I','N','E',8,'N','E','W','L','I','N','E','S',7,'N','E',
8+   'W','P','A','G','E',4,'R','E','A','D',5,'W','R','I','T','E',0(930)
9  %externalintegerarray namedlink(0:255)=0,76,0(12),89,0(54),84,0(118),
10+   52,0(11),1,12,23,34,0(4),67,0(23),46,0(5),59,0(17)
11 %externalintegerarray taglink(0:255)=0,13,0(12),16,0(54),14,0(118),8,
12+   0(11),1,3,4,5,0(4),11,0(23),7,0(5),10,0(17)
13 %externalintegerarray tag(1:512)=16_40100001,16_01010002,16_41000002,
14+   16_40000003,16_40100004,16_01010002,16_40000005,16_40100006,
15+   16_01010002,16_40000007,16_40100008,16_01010002,16_40000009,
16+   16_4010000a,16_11010002,16_4020000b,16_01010002,16_01010003,0(494)
17 %externalintegerarray link(1:512)=2,0,0,0,6,0,0,9,0,0,12,0,0,
18+   15,0,17,18,0,0(494)
19 %externalinteger namedp=95
20 %externalinteger tagasl=19
21 %externalinteger expropt=0
22 %externalinteger condopt=0
23 %externalinteger tagsopt=0
24 !-----------------------------------------------------------------------
25 %externalroutinespec statement(%integer statementp)
26 %externalstring(255)%fnspec strint(%integer n,p)
27 %externalroutinespec fault(%string(63) mess)
28 %externalroutinespec dump(%string(7) opn,reg,base,%integer disp)
29 %externalstring(255)%fnspec name(%integer ident)
30 !-----------------------------------------------------------------------
SKIMP
31 %externalroutine skimp(%string(63) s)
32 %routinespec readps
33 %routinespec read statement
34 %routinespec rpsym(%integername l)
35 %integerfnspec findk(%string(*)%name k)
36 %integerfnspec compare(%integer p)
37 %recordformat kdf(%byteinteger l,n,a,b)
38 %record(kdf)%array kd(1:255)
39 %string(15)%array pn(256:319)
40 %integerarray pp(256:319)
41 %integerarray ps(1:512)
42 %integerarray t,tt(1:256)
43 %integer tp,ap,ttp,i,j,psflag
44 %string(63) file,options,option,as
45 %owninteger lexopt=0
46 %owninteger analopt=0
47   %if s->("/").options %then %start
48     %unless options->options.(" ").file %then file=""
49     %cycle
50       %unless options->option.("/").options %then %c
51+         option=options %and options=""
52       %if option->("NO").option %then i=0 %else i=1
53       %if option="LEX" %then lexopt=i %else %c
54+       %if option="ANAL" %then analopt=i %else %c
55+       %if option="EXPR" %then expropt=i %else %c
```

```
56+       %if option="COND" %then condopt=i %else %c
57+       %if option="TAGS" %then tagsopt=i %else %c
58+       print string(option." OPTION ?
59+ ") %and %stop
60        %repeat %until options=""
61     %finish %else file=s
62     readps
63     %if file="" %then open output(1,".TT") %else open output(1,file.".LIS")
64     select output(1)
65     print string("
66+
67+                        SKIMP COMPILER MKII
68+
69+ FILE: ".file."
70+ OPTIONS: ")
71       %if lexopt=1 %then print string("LEX ")
72       %if analopt=1 %then print string("ANAL ")
73       %if expropt=1 %then print string("EXPR ")
74       %if condopt=1 %then print string("COND ")
75       %if tagsopt=1 %then print string("TAGS ")
76       newline
77       %if psflag#0 %then fault("PHRASE STRUCTURE FAULTY") %and %stop
78       %if file="" %then open input(1,".tt") %else open input(1,file.".IMP")
79       select input(1)
80       ! set up tags available space list
81       %for i=tagasl,1,511 %cycle
82         link(i)=i+1
83       %repeat
84       %cycle    ;! for each statement
85         read statement
86         ttp=tp-1
87         tp=1
88         ap=1
89         %if compare(258)=0 %or tp#ttp %then fault("SYNTAX ?") %else %start
90           %if analopt=1 %then %start
91             newline
92             j=0
93             %for i=1,1,ap-1 %cycle    ;! print analysis record
94               %if a(i)<0 %then as="  (".strint(i,1)."/".pn(a(i)<<1>>17). %c
95+               ")" %and print string(as) %and j=j+length(as) %and %c
96+               a(i)=a(i)&16_ffff
97               write(a(i),4)
98               j=j+5
99               %if j>60 %then newline %and j=0
100            %repeat
101            newlines(2)
102          %finish %else %start
103            %for i=1,1,ap-1 %cycle    ;! remove phrase numbers
104              %if a(i)<0 %then a(i)=a(i)&16_ffff
105            %repeat
106          %finish
107          statement(1)    ;! generate code for statement
108        %finish
109      %repeat
110   !-----------------------------------------------------------------------
READPS 111   %routine readps
112   ! read phrase structure from file 'SKIMPPS' and reduce it
113   %string(31)%array ka(1:128)
114   %integerarray kna(1:128)
115   %string(31) k
```

```
116  %integer kap,kdasl,kn,i,l,psp,pnp,alt
117  %integername np
118  %routinespec insert(%string(15) k)
119  %routinespec extract(%integer i,%string(15) k)
120  %routinespec assign(%integer i)
121  %integerfnspec newkd
122  %routinespec returnkd(%integer i)
123  %routinespec returnlist(%integer i)
124  %integerfnspec phrase
125  %routinespec literal
126   %routinespec keyword
127    open input(2,"SKIMPPS.IMP")
128    select input(2)
129    open output(2,"SKIMPPSL.LIS")
130    select output(2)
131    print string("
132+
133+ PHRASE STRUCTURE
134+
135+ ")
136    ! scan file to build keyword dictionary
137    kap=1
138    %cycle
139      rpsym(l)
140      %if l='$' %then %exit
141      %if l='"' %then %start
142        k=""
143        %cycle
144          rpsym(l)
145          %if l='"' %then %exit
146          %if 'A'<=l<='Z' %then k=k.tostring(l)
147        %repeat
148          ka(kap)=k
149          kap=kap+1
150      %finish
151    %repeat
152    %for i=1,1,26 %cycle
153      kd(i)=0
154    %repeat
155    %for i=27,1,254 %cycle
156      kd(i)_b=i+1
157    %repeat
158    kdasl=27
159    i=1
160    insert(ka(i)) %and i=i+1 %until i=kap
161    kn=128
162    %for i=1,1,26 %cycle
163      %if kd(i)_l#0 %then assign(i)
164    %repeat
165    kap=1
166    %for i=1,1,26 %cycle
167      %if kd(i)_l#0 %then extract(i,"")
168    %repeat
169    print string("
170+
171+ KEYWORDS
172+
173+ ")
174    %for i=1,1,kap-1 %cycle
175      print string(strint(kna(i),3)."  ".ka(i)."
```

```
176+ ")
177     %repeat
178     ! reread file and reduce phrase structure
179     reset input
180     pn(256)="NAME"
181     pp(256)=0
182     pn(257)="CONST"
183     pp(257)=0
184     pnp=258
185     psp=1
186     %cycle    ;! for each phrase definition
187       read symbol(l)
188        %if l='$' %then %exit
189       %if l='<' %then %start    ;! start of phrase definition
190         pp(phrase)=psp
191         %cycle    ;! for each alternative
192           alt=psp
193           np==ps(psp+1)
194           np=0    ;! number of phrases
195           psp=psp+2
196           %cycle    ;! for each item
197             read symbol(l)
198             %if l='<' %then ps(psp)=phrase %and psp=psp+1 %and np=np+1
199             %if l='''' %then literal
200             %if l='"' %then keyword
201             %if l=',' %or l=';' %then %exit
202           %repeat
203           ps(alt)=psp
204           %if l=';' %then %exit
205         %repeat
206         ps(psp)=0
207         psp=psp+1
208       %finish
209     %repeat
210     psflag=0
211     %for i=258,1,pnp-1 %cycle
212       %if pp(i)=0 %then fault("<".pn(i)."> NOT DEFINED") %and psflag=1
213     %repeat
214     print string("
215+
216+ PHRASES
217+
218+ ")
219     %for i=256,1,pnp-1 %cycle
220       print string(strint(i,3).strint(pp(i),6)."   ".pn(i)."
221+ ")
222     %repeat
223     print string("
224+
225+ REDUCED PHRASE STRUCTURE
226+
227+ ")
228     %for i=1,1,psp-1 %cycle
229       %if (i-1)&15=0 %then print string("
230+ ".strint(i,3)."   ")
231       write(ps(i),3)
232     %repeat
233     newlines(2)
234     %return
235   !-----------------------------------------------------------------------
```

```
INSERT  236  %routine insert(%string(15) k)
        237  ! search for and insert keyword into dictionary
        238  %integer i,j,l
        239    l=charno(k,1)
        240    k->(tostring(l)).k
        241    i=l-'A'+1
        242    %if kd(i)_l#0 %then %start
        243  search:%if k=="" %then %start
        244         %if kd(i)_a#0 %then extract(kd(i)_a,"") %and %c
        245+          returnlist(kd(i)_a) %and kd(i)_a=0
        246         %return
        247       %finish
        248       %if kd(i)_a=0 %then insert(k) %and %return
        249       l=charno(k,1)
        250        k->(tostring(l)).k
        251       i=kd(i)_a
        252       %cycle
        253         %if kd(i)_l=l %then ->search
        254         %if kd(i)_b=0 %then %exit
        255         i=kd(i)_b
        256       %repeat
        257       j=i
        258       i=newkd
        259       kd(j)_b=i
        260     %finish
        261     ! insert remainder of letters
        262     %cycle
        263       kd(i)_l=l
        264       %if k=="" %then %return
        265       l=charno(k,1)
        266       k->(tostring(l)).k
        267       j=i
        268         i=newkd
        269         kd(j)_a=i
        270     %repeat
        271   %end
        272   !-----------------------------------------------------------------
EXTRACT 273   %routine extract(%integer i,%string(15) k)
        274   %string(15) kk
        275     %if i=0 %then %return
        276     kk=k.tostring(kd(i)_l)
        277     %if kd(i)_a=0 %then ka(kap)=kk %and kna(kap)=kd(i)_n %and kap=kap+1%c
        278+      %else extract(kd(i)_a,kk)
        279     extract(kd(i)_b,k)
        280   %end
        281   !-----------------------------------------------------------------
ASSIGN  282   %routine assign(%integer i)
        283     %if i=0 %then %return
        284     %if kd(i)_a=0 %then kd(i)_n=kn %and kn=kn+1 %else assign(kd(i)_a)
        285     assign(kd(i)_b)
        286   %end
        287   !-----------------------------------------------------------------
NEWKD   288   %integerfn newkd
        289   %integer i
        290     %if kdasl=0 %then print string("KD ASL EMPTY") %and %stop
        291     i=kdasl
        292     kdasl=kd(i)_b
        293     kd(i)=0
        294     %result=i
        295   %end
```

```
296  !-----------------------------------------------------------------------
297  %routine returnkd(%integer i)
298    kd(i)_b=kdasl
299    kdasl=i
300  %end
301  !-----------------------------------------------------------------------
302  %routine returnlist(%integer i)
303    %if i=0 %then %return
304    returnlist(kd(i)_a)
305    returnlist(kd(i)_b)
306    returnkd(i)
307  %end
308   !----------------------------------------------------------------------
309  %integerfn phrase
310  %string(15) p
311  %integer i,l
312    p=""
313    %cycle
314      read symbol(l)
315      %if l='>' %then %exit %else p=p.tostring(l)
316    %repeat
317    %for i=256,1,pnp-1 %cycle
318      %if pn(i)=p %then %result=i
319    %repeat
320    pn(pnp)=p
321    pp(pnp)=0
322    pnp=pnp+1
323    %result=pnp-1
324  %end
325  !-----------------------------------------------------------------------
326  %routine literal
327  %integer l
328    %cycle
329      read symbol(l)
330      %if l='''' %then %return %else ps(psp)=l %and psp=psp+1
331    %repeat
332  %end
333  !-----------------------------------------------------------------------
334  %routine keyword
335  %string(31) k
336  %integer l
337    k=""
338    %cycle
339      read symbol(l)
340      %if l='"' %then %exit
341      %if 'A'<=l<='Z' %then k=k.tostring(l)
342    %repeat
343    ps(psp)=findk(k) %and psp=psp+1 %until k=""
344  %end
345  %end
346  !-----------------------------------------------------------------------
347  %routine read statement
348  %routinespec store(%integer l)
349  %routinespec keyword
350  %routinespec name
351  %routinespec const
352  %integer i,l,ksh,ttp,ttpp
353    ! line reconstruct phase
354    newlines(2)
355    ttp=1
```

RETURNKD

RETURN LIST

PHRASE

LITERAL

KEYWORD

READ STATEMENT

```
356     ksh=0
357     %cycle    ;! for each character
358       rpsym(l)
359       %if l='%' %then ksh=128 %else %start
360         %unless 'A'<=l<='Z' %then ksh=0
361         %if l#' ' %then %start    ;! discard spaces
362           %if l='!' %and ttp=1 %then %start
363             rpsym(l) %until l=';' %or l=nl    ;! discard comments    !....;
364           %finish %else %start
365             store(l)
366             %if l='''' %then %start
367               rpsym(l) %and store(l) %until l=''''
368             %finish %else %start
369               %if l=';' %or l=nl %then %start
370                 %if ttp=2 %then ttp=1 %else %start
371                   %if l=';' %then newline %and %exit
372                   %if tt(ttp-2)='C'+128 %then ttp=ttp-2 %else %exit
373                 %finish
374               %finish
375             %finish
376           %finish
377         %finish
378       %finish
379     %repeat
380     ! lexical phase
381     tp=1
382     ttpp=1
383     %cycle    ;! for each lexical item
384       i=tt(ttpp)
385       %if i>=128 %then keyword %else %start
386         %if 'A'<=i<='Z' %then name %else %start
387           %if '0'<=i<='9' %or i='''' %then const %else %c
388+            t(tp)=i %and tp=tp+1 %and ttpp=ttpp+1
389           %finish
390         %finish
391       %repeat %until ttpp=ttp
392       %if lexopt=1 %then %start
393         newline
394         %for ttpp=1,1,tp-2 %cycle
395           write(t(ttpp),4)
396           %if ttpp&16_f=0 %then newline
397         %repeat
398         newline
399       %finish
400       %return
401     !-----------------------------------------------------------------
STORE 402     %routine store(%integer l)
403       %if ttp>256 %then fault("STATEMENT TOO LONG") %and %stop
404       tt(ttp)=l+ksh
405       ttp=ttp+1
406     %end
407     !-----------------------------------------------------------------
KEYWORD 408     %routine keyword
409     %string(255) k
410     %integer i
411       k=""
412       %while tt(ttpp)>128 %then k=k.tostring(tt(ttpp)-128) %and ttpp=ttpp+1
413       i=findk(k) %and t(tp)=i %and tp=tp+1 %until k="" %or i=0
414     %end
415     !-----------------------------------------------------------------
```

```
416  %routine name
417  %string(*)%name sname
418  %integer i,l,hash
419    sname==string(addr(named(namedp)))
420    hash=0
421    sname=""
422    l=tt(ttpp)
423    %cycle
424      %if namedp+length(sname)>=1022 %then fault("NAME DICTIONARY FULL")%c
425+       %and %stop
426      %if length(sname)=255 %then fault("NAME TOO LONG") %and %stop
427      sname=sname.tostring(l)
428      hash=hash<<8!l
429      ttpp=ttpp+1
430      l=tt(ttpp)
431    %repeat %until l<'0' %or '9'<l<'A' %or l>'Z'
432    hash=hash-hash//251*251
433     i=hash
434    %cycle    ;! scan dictionary
435      %if namedlink(i)=0 %then namedlink(i)=namedp %and %c
436+       namedp=namedp+length(sname)+1 %and %exit    ;! insert name
437      %if sname=string(addr(named(namedlink(i)))) %then %exit
438      i=(i+1)&255
439      %if i=hash %then fault("NAME DICTIONARY FULL") %and %stop
440    %repeat
441    t(tp)=256    ;! <name>
442    t(tp+1)=i    ;! ident
443    tp=tp+2
444  %end
445  !-----------------------------------------------------------------
CONST 446  %routine const
447  %integer l,value,flag,count,maxby10,maxld
448    value=0
449    flag=0
450    %if tt(ttpp)='''' %then %start
451      count=0
452      %cycle
453        ttpp=ttpp+1
454        %if tt(ttpp)='''' %then %start
455          ttpp=ttpp+1
456          %if tt(ttpp)#'''' %then %exit
457        %finish
458        value=value<<8!tt(ttpp)
459        count=count+1
460      %repeat
461      %unless 1<=count<=4 %then flag=1
462    %finish %else %start
463      maxby10=16_7fffffff//10
464      maxld=16_7fffffff-maxby10*10
465      l=tt(ttpp)
466      %cycle
467        %if value>maxby10 %or (value=maxby10 %and l>maxld) %then flag=1 %c
468+         %else value=value*10+l-'0'
469        ttpp=ttpp+1
470        l=tt(ttpp)
471      %repeat %until l<'0' %or l>'9'
472    %finish
473    t(tp)=257    ;! <const>
474    %if flag#0 %then fault("CONSTANT INVALID") %and value=0
475    t(tp+1)=value
```

```
476     tp=tp+2
477   %end
478   %end
479   !-------------------------------------------------------------------------
RPSYM 480   %routine rpsym(%integername l)
481     read symbol(l)
482     print symbol(l)
483     %if 'a'<=l<='z' %then l=l-'a'+'A'
484   %end
485   !-------------------------------------------------------------------------
486   %integerfn findk(%string(*)%name k)
487   ! look keyword up in dictionary
488   %integer i,l
489     l=charno(k,1)
490     k->(tostring(l)).k
491     i=l-'A'+1
492     %if kd(i)_l=0 %then %result=0
493   search:%if k="" %or kd(i)_a=0 %then %result=kd(i)_n
494     l=charno(k,1)
495     k->(tostring(l)).k
496      i=kd(i)_a
497     %cycle
498       %if kd(i)_l=l %then ->search
499       %if kd(i)_b=0 %then %result=0
500       i=kd(i)_b
501     %repeat
502   %end
503   !-------------------------------------------------------------------------
COMPARE 504   %integerfn compare(%integer p)
505   %integer app,tpp,alt,altend,psp,psi
506     a(ap)=p<<16!16_80000001          ;! phrase number & alternative 1
507     %if p<=257 %then %start          ;! <name> or <const>
508       %if p=t(tp) %then %start       ;! success
509         a(ap+1)=t(tp+1)
510         ap=ap+2
511         tp=tp+2
512         %result=1
513       %finish %else %result=0
514     %finish
515     tpp=tp          ;! preserve text pointer
516     app=ap          ;! preserve analysis record pointer
517     psp=pp(p)       ;! start of phrase definition
518     %cycle          ;! for each alternative
519       alt=ap+1
520       altend=ps(psp)
521       ap=alt+ps(psp+1)     ;! leave gap for forward pointers
522       %if ap>255 %then fault("ANALYSIS RECORD TOO LONG") %and %stop
523       psp=psp+2
524       %cycle    ;! for each item
525         %if psp=altend %then %result=1    ;! success
526         psi=ps(psp)
527         %if psi>=256 %then %start         ;! phrase
528           a(alt)=ap                       ;! forward pointer
529           %if compare(psi)=0 %then %exit
530           alt=alt+1
531         %finish %else %start              ;! literal or keyword
532           %if psi#t(tp) %then %exit
533           tp=tp+1
534         %finish
535         psp=psp+1
```

```
536       %repeat
537       %if ps(altend)=0 %then %result=0    ;! failure
538       psp=altend
539       tp=tpp              ;! backtrack text pointer
540       ap=app              ;! backtrack analysis record pointer
541       a(ap)=a(ap)+1       ;! next alternative number
542     %repeat
543   %end
544   %endofprogram
Code 9996 bytes  Glap 10136 bytes  Diags 2355 bytes   Total size 22487 bytes
   464 statements compiled
```

```
Source file: SKIMPB.IMP                Compiled on 25-OCT-1979 at 09:39:49

      Computer Science IMP77 Compiler.  Version 6.01

 1  %externalintegerarrayspec a(1:500)
 2  %externalintegerarrayspec taglink(0:255)
 3  %externalintegerarrayspec tag(1:512)
 4  %externalintegerarrayspec link(1:512)
 5  !-----------------------------------------------------------------------
 6  %externalroutinespec expr(%integer exprp)
 7  %externalintegerfnspec cond(%integer condp,tlabel,flabel)
 8  %externalstring(255)%fnspec strint(%integer n,p)
 9  %externalintegerfnspec getwork
10  %externalroutinespec returnwork(%integer work)
11  %externalroutinespec clearwork
12  %externalintegerfnspec newtag
13  %externalroutinespec pushtag(%integer ident,form,type,dim,level,rad)
14  %externalroutinespec poptags
15  %externalintegerfnspec getlabel(%integer constp)
16  %externalroutinespec filllabel(%integer label)
17  %externalintegerfnspec fillbranch(%integer label)
18  %externalroutinespec poplabels
19  %externalintegerfnspec nextplabel
20  %externalroutinespec dump(%string(7) opn,reg,base,%integer disp)
21  %externalroutinespec fault(%string(63) mess)
22  %externalstring(255)%fnspec name(%integer ident)
23  %externalroutinespec pushstart(%integer flag,plab)
24  %externalroutinespec popstart(%integername flag,plab)
25  %externalroutinespec clearstart
26  %externalintegerfnspec enter
27  %externalroutinespec dump return
28  %externalroutinespec proc(%integer procp)
29  %externalroutinespec array(%integer arrayp)
30  %externalroutinespec endofprog
31  !-----------------------------------------------------------------------
32  %externalintegerarray nextrad(0:15)
33  %externalstring(4)%array display(0:15)="DR0","DR1","DR2","DR3","DR4",
34+  "DR5","DR6","DR7","DR8","DR9","DR10","DR11","DR12","DR13","DR14","DR15"
35  %externalinteger level,nextcad
36  !-----------------------------------------------------------------------
37  %ownintegerarray proctype(0:15)
38  %ownintegerarray staticalloc(0:15)
39  %ownintegerarray skipproc(0:15)
40  !-----------------------------------------------------------------------
41  %externalroutine statement(%integer statementp)
42  %routinespec instr(%integer instrp)
43  %switch sttype(1:8)
44  %integer condp,instrp,elsep,constp,arrayp,namep,namesp,expr1p,expr2p, %c
45+   instr2p,tlabel,flabel,label,fplabel,tplabel,work1,work2,flag,plabel,%c
46+   procp,formalp,formp,params,procid,ident,form,paramt,paraml,dim
47    ->sttype(a(statementp))
48  !-----------------------------------------------------------------------
49  sttype(1):! <instr>
50    instr(a(statementp+1))
51    %return
52  !-----------------------------------------------------------------------
53  sttype(2):! "IF"<cond>"THEN"<instr><else>
54    condp=a(statementp+1)
55    instrp=a(statementp+2)
```

STATEMENT

```
    elsep=a(statementp+3)
    %if a(instrp)=2 %then %start    ;! branch
      constp=a(instrp+1)
      tlabel=getlabel(constp)
      %if a(elsep)=2 %then filllabel(cond(condp,tlabel,-1)) %else %start
        instrp=a(elsep+1)
        %if a(instrp)=2 %then %start    ;! branch
          constp=a(instrp+1)
          flabel=getlabel(constp)
          filllabel(cond(condp,tlabel,flabel))
          dump("B","","",fillbranch(flabel))
        %finish %else %start
          filllabel(cond(condp,tlabel,-1))
          %if a(instrp)=3 %then pushstart(1,-1) %else instr(instrp)
        %finish
      %finish
    %finish %else %start
      %if a(elsep)=2 %then %start
        fplabel=cond(condp,-1,-1)
        %if a(instrp)=3 %then pushstart(0,fplabel) %else %c
          instr(instrp) %and filllabel(fplabel)
      %finish %else %start
        instr2p=a(elsep+1)
        %if a(instr2p)=2 %then %start    ;! branch
          constp=a(instr2p+1)
          fplabel=cond(condp,-1,getlabel(constp))    ;! result always -1
          instr(instrp)
        %finish %else %start
          fplabel=cond(condp,-1,-1)
          instr(instrp)
          tplabel=nextplabel
          dump("B","","",fillbranch(tplabel))
          filllabel(fplabel)
          %if a(instr2p)=3 %then pushstart(1,tplabel) %else %c
            instr(instr2p) %and filllabel(tplabel)
        %finish
      %finish
    %finish
    %return
!-----------------------------------------------------------------------
sttype(3):! <const>':'<statement>
    constp=a(statementp+1)
    statementp=a(statementp+2)
    label=getlabel(constp)
    filllabel(label)
    statement(statementp)
    %return
!-----------------------------------------------------------------------
sttype(4):! "FINISH"<else>
    elsep=a(statementp+1)
    popstart(flag,plabel)
    %if flag=0 %then %start    ;! first %start/%finish
      %if a(elsep)=1 %then %start
        instrp=a(elsep+1)
        tplabel=nextplabel
        dump("B","","",fillbranch(tplabel))
        filllabel(plabel)
        %if a(instrp)=3 %then pushstart(1,tplabel) %else %c
          instr(instrp) %and filllabel(tplabel)
      %finish %else filllabel(plabel)
```

```
116     %finish %else %start    ;! second %start/%finish
117       %if a(elsep)=1 %then fault("SPURIOUS %ELSE") %else filllabel(plabel)
118     %finish
119     %return
120   !-----------------------------------------------------------------------
121   sttype(5):! "INTEGER"<array>
122     arrayp=a(statementp+1)
123     namep=a(arrayp+1)
124     namesp=a(arrayp+2)
125     %if a(arrayp)=1 %then %start    ;! array declaration
126       expr1p=a(arrayp+3)
127       expr2p=a(arrayp+4)
128       expr(expr1p)
129       work1=getwork
130       dump("STR","ACC",display(level),work1)
131       expr(expr2p)
132       dump("LDA","ACC","ACC",1)
133       work2=getwork
134       dump("STR","ACC",display(level),work2)
135       %cycle
136         pushtag(a(namep+1),2,1,1,level,nextrad(level))
137         dump("SUB","STP",display(level),work1)
138         dump("STR","STP",display(level),nextrad(level))
139         dump("ADD","STP",display(level),work2)
140         nextrad(level)=nextrad(level)+1
141         %if a(namesp)=2 %then %exit
142         namep=a(namesp+1)
143         namesp=a(namesp+2)
144       %repeat
145       returnwork(work1)
146       returnwork(work2)
147     %finish %else %start
148       %cycle
149         pushtag(a(namep+1),0,1,0,level,nextrad(level))
150         nextrad(level)=nextrad(level)+1
151         %if a(namesp)=2 %then %exit
152         namep=a(namesp+1)
153         namesp=a(namesp+2)
154       %repeat
155     %finish
156     %return
157   !-----------------------------------------------------------------------
158   sttype(6):! <proc><name><formal>
159     %if level=0 %then fault("PROCEDURE BEFORE %BEGIN")
160     %if level=15 %then fault("PROCEDURE NESTING TOO DEEP")
161     procp=a(statementp+1)
162     namep=a(statementp+2)
163     formalp=a(statementp+3)
164     procid=a(namep+1)
165     skipproc(level)=nextcad
166     dump("B","","",0)     ;! branch round procedure
167     pushtag(procid,4,a(procp)-1,0,level,nextcad)
168     level=level+1
169     proctype(level)=a(procp)
170     staticalloc(level)=enter
171     nextrad(level)=2
172     %if a(formalp)=2 %then %return     ;! no parameters
173     params=0
174     paraml=taglink(procid)
175     %cycle
```

```
176       formp=a(formalp+1)
177       namep=a(formalp+2)
178       namesp=a(formalp+3)
179       formalp=a(formalp+4)
180       %if a(formp)=1 %then form=3 %and dim=1 %else %start
181         %if a(formp)=2 %then form=1 %else form=0
182         dim=0
183       %finish
184       %cycle
185         ident=a(namep+1)
186         ! declare parameters as locals
187         pushtag(ident,form,1,dim,level,nextrad(level))
188         nextrad(level)=nextrad(level)+1
189         ! append parameter tag cells to procedure tag cell
190         paramt=newtag
191         tag(paramt)=tag(taglink(ident))
192         link(paramt)=link(paraml)
193         link(paraml)=paramt
194         paraml=paramt
195         params=params+1
196         %if params>15 %then fault(name(procid). %c
197+          " HAS TOO MANY PARAMETERS") %and %stop
198         %if a(namesp)=2 %then %exit
199         namep=a(namesp+1)
200         namesp=a(namesp+2)
201       %repeat
202     %repeat %until a(formalp)=2
203     ! insert number of parameters into tag cell
204     tag(taglink(procid))=tag(taglink(procid))!params<<20
205     %return
206   !-----------------------------------------------------------------------
207   sttype(7):! "END"<ofprog>
208     dump("FILL","ALLOC",strint(staticalloc(level),1),nextrad(level))
209     poptags
210     poplabels
211     clearstart
212     clearwork
213     %if proctype(level)=1 %then dump return %else dump("STOP","","",0)
214     level=level-1
✓   215     %if a(a(statementp+1))=2 %then %start    ;! %end
216       %if level<=0 %then fault("SPURIOUS %END") %and endofprog
217       dump("FILL","SKIP",strint(skipproc(level),1),nextcad)
218     %finish %else %start    ;! %endofprogram
219       %if level#0 %then fault("TOO FEW %ENDS")
220       endofprog
221     %finish
222     %return
223   !-----------------------------------------------------------------------
224   sttype(8):! "BEGIN"
225     %if level#0 %then fault("SPURIOUS %BEGIN") %else %start
226       level=1
✓   227       proctype(1)=0
228       staticalloc(1)=enter
229     %finish
230     %return
231   !-----------------------------------------------------------------------
INSTR 232   %routine instr(%integer instrp)
233   %switch instype(1:6)
234   %string(4) base
235   %integer namep,assignp,constp,ident,actualp,exprp,nametag,disp,work
```

```
236     ->instype(a(instrp))
237   !-----------------------------------------------------------------------
238   instype(1):! <name><actual><assign>
239     namep=a(instrp+1)
240     actualp=a(instrp+2)
241     assignp=a(instrp+3)
242     ident=a(namep+1)
243     %if taglink(ident)=0 %then fault(name(ident)." NOT DECLARED") %c
244+      %and %return
245     nametag=tag(taglink(ident))
246     %if a(assignp)=1 %then %start
247       %if nametag>>28=4 %then fault(name(ident)." NOT A DESTINATION") %c
248+        %and %return
249       exprp=a(assignp+1)
250       %if nametag>>28>=2 %then %start     ;! array variable
251         expr(exprp)
252         work=getwork
253         dump("STR","ACC",display(level),work)
254         array(instrp)
255         dump("LOAD","WK",display(level),work)
256         dump("STR","WK","ACC",0)
257         returnwork(work)
258       %finish %else %start
259         expr(exprp)
260         base=display(nametag>>16&16_f)
261         disp=nametag&16_ffff
262         %if nametag>>28=1 %then %start     ;! %name variable
263           dump("LOAD","WK",base,disp)
264           dump("STR","ACC","WK",0)
265         %finish %else dump("STR","ACC",base,disp)
266         %if a(actualp)=1 %then fault(name(ident)." DECLARED AS SCALAR")
267       %finish
268     %finish %else %start
269       %if nametag>>28=4 %and nametag>>24&16_f=0 %then proc(instrp) %c
270+        %else fault(name(ident)." NOT A ROUTINE NAME")
271     %finish
272     %return
273   !-----------------------------------------------------------------------
274   instype(2):! '->'<const>
275     constp=a(instrp+1)
276     label=getlabel(constp)
277     dump("B","","",fillbranch(label))
278     %return
279   !-----------------------------------------------------------------------
280   instype(3):! "START"
281     fault("ILLEGAL %START")
282     %return
283   !-----------------------------------------------------------------------
284   instype(4):! "RETURN"
285     %if proctype(level)#1 %then fault("%RETURN OUT OF CONTEXT")
286     dumpreturn
287     %return
288   !-----------------------------------------------------------------------
289   instype(5):! "RESULT"'='<expr>
290     %if proctype(level)#2 %then fault("%RESULT OUT OF CONTEXT")
291     expr(a(instrp+1))
292     dumpreturn
293     %return
294   !-----------------------------------------------------------------------
295   instype(6):! "STOP"
```

```
 296     dump("STOP","","",0)
 297   %end
 298   %end
 299   %endoffile
Code 7292 bytes  Glap 1256 bytes  Diags 1276 bytes   Total size 9824 bytes
   253 statements compiled
```

Source file: SKIMPC.IMP Compiled on 25-OCT-1979 at 09:51:22

Computer Science IMP77 Compiler. Version 6.01

```
 1  %externalintegerarrayspec a(1:500)
 2  %externalintegerspec condopt
 3  !-----------------------------------------------------------------
 4  %externalroutinespec expr(%integer exprp)
 5  %externalroutinespec dump(%string(7) opn,reg,base,%integer disp)
 6  %externalroutinespec filllabel(%integer label)
 7  %externalintegerfnspec fillbranch(%integer label)
 8  %externalintegerfnspec nextplabel
 9  %externalroutinespec fault(%string(63) mess)
10  !-----------------------------------------------------------------
11  %externalinteger condflag=0
12  !-----------------------------------------------------------------
13  %externalintegerfn cond(%integer condp,tlabel,flabel)
14  %routinespec processcond(%integer condp)
15  %routinespec test(%integer ltestp)
16  %routinespec condrest(%integer condrestp)
17  %routinespec store(%integer testp,level,andor)
18  %routinespec show(%string(7) an,%integerarrayname a,%integer p)
19  %conststring(3)%array true(1:6)="BZ","BNZ","BNG","BL","BNL","BG"
20  %conststring(3)%array false(1:6)="BNZ","BZ","BG","BNL","BL","BNG"
21  %string(3) opn
22  %constintegerarray index(1:17)=1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17
23  %integerarray testpa,levela,andora,brancha(1:16),labela(1:17)
24  %integer p,pp,ppp,testp,level,andor,comp
25    level=0
26    p=1
27    processcond(condp)
28    store(testp,-1,1)     ;! pseudo-%and
29    store(0,-2,2)         ;! pseudo-%or
30    p=p-2
31    %for pp=1,1,p %cycle     ;! find branch destinations
32      level=levela(pp)
33      andor=andora(pp)
34      %for ppp=pp+1,1,p+1 %cycle
35        %if levela(ppp)<level %then %start
36          level=levela(ppp)
37          %if andora(ppp)#andor %then %exit
38        %finish
39      %repeat
40      brancha(pp)=ppp+1
41    %repeat
42    %if tlabel>=0 %then %start
43      andora(p)=2     ;! change last branch to branch on true
44      brancha(p)=p+1
45      labela(p+1)=tlabel
46    %finish
47    labela(p+2)=flabel
48    %for pp=1,1,p %cycle     ;! assign private labels where needed
49      %if labela(brancha(pp))<0 %then labela(brancha(pp))=nextplabel
50    %repeat
51    %if condopt=1 %then %start
52      newline
53      show("       ",index,p+2)
54      show("TESTP ",testpa,p)
55      show("LEVEL ",levela,p+1)
```

```
56        show("ANDOR ",andora,p+1)
57        show("BRANCH",brancha,p)
58        show("LABEL ",labela,p+2)
59        newline
60      %finish
61      %for pp=1,1,p %cycle    ;! generate test code and fill labels
62        %if labela(pp)>=0 %then filllabel(labela(pp))
63        condflag=1
64        expr(testpa(pp))
65        comp=a(a(testpa(pp)+2))
66        %if andora(pp)=1 %then opn=false(comp) %else opn=true(comp)
67        dump(opn,"ACC","",fillbranch(labela(brancha(pp))))
68      %repeat
69      %if labela(p+1)>=0 %and tlabel<0 %then filllabel(labela(p+1))
70      %if flabel>=0 %then %result=-1 %else %result=labela(p+2)
71      !-----------------------------------------------------------------
PROCESS 72  %routine processcond(%integer condp)
COND    73      test(a(condp+1))
74      condrest(a(condp+2))
75  %end
76  !-----------------------------------------------------------------
 TEST   77  %routine test(%integer ltestp)
78      %if a(ltestp)=1 %then testp=ltestp %else level=level+1 %and %c
79+       processcond(a(ltestp+1)) %and level=level-1
80  %end
81  !-----------------------------------------------------------------
CONDREST 82  %routine condrest(%integer condrestp)
83  %integer andor
84      andor=a(condrestp)
85      %unless andor=3 %then %start
86        store(testp,level,andor) %and test(a(condrestp+1)) %and %c
87+           condrestp=a(condrestp+2) %until a(condrestp)=2
88      %finish
89  %end
90  !-----------------------------------------------------------------
STORE   91  %routine store(%integer testp,level,andor)
92      %if p>16 %then fault("CONDITION TOO LONG") %and %stop
93      testpa(p)=testp
94      levela(p)=level
95      andora(p)=andor
96      labela(p)=-1
97      p=p+1
98  %end
99  !-----------------------------------------------------------------
 SHOW  100  %routine show(%string(7) an,%integerarrayname a,%integer p)
101  %integer pp
102      print string(an." ")
103      %for pp=1,1,p %cycle
104        write(a(pp),5)
105      %repeat
106      newline
107    %end
108    %end
109    %endoffile
Code 2492 bytes  Glap 432 bytes  Diags 613 bytes   Total size 3537 bytes
    99 statements compiled
```

Source file: SKIMPD.IMP Compiled on 25-OCT-1979 at 09:40:47

Computer Science IMP77 Compiler. Version 6.01

```
 1  %externalintegerarrayspec a(1:500)
 2  %externalbyteintegerarrayspec named(1:1024)
 3  %externalintegerarrayspec namedlink(0:255)
 4  %externalintegerarrayspec taglink(0:255)
 5  %externalintegerarrayspec tag(1:512)
 6  %externalintegerarrayspec link(1:512)
 7  %externalintegerarrayspec nextrad(0:15)
 8  %externalstring(4)%arrayspec display(0:15)
 9  %externalintegerspec tagasl,level,tagsopt,nextcad,namedp
10  !-----------------------------------------------------------------------
11  %externalroutinespec expr(%integer exprp)
12  !-----------------------------------------------------------------------
13  %ownintegerarray worklist(0:15)=0(16)
14  %ownintegerarray namelist(0:15)=0(16)
15  %ownintegerarray branchlist(0:15)=0(16)
16  %ownintegerarray startlist(0:15)=0(16)
17  %ownintegerarray cot(0:127)
18  %owninteger cotp,faults,params
19  !-----------------------------------------------------------------------
STRINT
20  %externalstring(255)%fn strint(%integer n,p)
21  %string(255) r
22  %string(1) s
23    %if n<0 %then s="-" %and n=-n %else s=""
24    r=""
25    r=tostring(n-n//10*10+'0').r %and n=n//10 %until n=0
26    r=s.r
27    %while length(r)<p %then r=" ".r
28    %result=r
29  %end
30  !-----------------------------------------------------------------------
STRHEX
31  %externalstring(8)%fn strhex(%integer n)
32  %conststring(1)%array h(0:15)="0","1","2","3","4","5","6","7","8",
33+   "9","A","B","C","D","E","F"
34  %integer i
35  %string(8) sh
36    sh=""
37    %for i=1,1,8 %cycle
38      sh=h(n&16_f).sh
39      n=n>>4
40    %repeat
41    %result=sh
42  %end
43  !-----------------------------------------------------------------------
FAULT
44  %externalroutine fault(%string(63) mess)
45    print string("*  ".mess."
46+
47+ ")
48    faults=faults+1
49  %end
50  !-----------------------------------------------------------------------
DUMP
51  %externalroutine dump(%string(7) opn,reg,base,%integer disp)
52    print string(strint(nextcad,5)."$           ". %c
53+     opn.",".reg.",".base.",".strint(disp,1)."
54+ ")
55    nextcad=nextcad+1 %unless opn="FILL"
```

```
      %end
      !-----------------------------------------------------------------------
NAME  %externalstring(255)%fn name(%integer ident)
        %unless 0<=ident<=255 %and namedlink(ident)#0 %then %result=""
        %result=string(addr(named(namedlink(ident))))
      %end
      !-----------------------------------------------------------------------
NEWTAG%externalintegerfn newtag
      %integer i
        %if tagasl=0 %then fault("TAG SPACE FULL") %and %stop
        i=tagasl
        tagasl=link(tagasl)
      %result=i
      %end
      !-----------------------------------------------------------------------
RETURN%externalintegerfn returntag(%integer tagi)
 TAG  %integer l
        l=link(tagi)
        link(tagi)=tagasl
        tagasl=tagi
        %result=l
      %end
      !-----------------------------------------------------------------------
GET   %externalintegerfn getwork
 WORK %integername cell
        cell==worklist(level)
        %while cell#0 %cycle
          %if tag(cell)<0 %then tag(cell)=-tag(cell) %and %result=tag(cell)
          cell==link(cell)
        %repeat
        cell=newtag
        tag(cell)=nextrad(level)
        nextrad(level)=nextrad(level)+1
        link(cell)=0
        %result=tag(cell)
      %end
      !-----------------------------------------------------------------------
RETURN%externalroutine returnwork(%integer work)
 WORK %integer cell
        cell=worklist(level)
        %while cell#0 %cycle
          %if tag(cell)=work %then tag(cell)=-work %and %return
          cell=link(cell)
        %repeat
      %end
      !-----------------------------------------------------------------------
CLEAR %externalroutine clearwork
 WORK %integer cell
        cell=worklist(level)
        %while cell#0 %then cell=returntag(cell)
        worklist(level)=0
      %end
      !-----------------------------------------------------------------------
GETCOTI%externalintegerfn getcoti(%integer const)
      %integer coti
        %if cotp>0 %then %start
          %for coti=0,1,cotp-1 %cycle
            %if cot(coti)=const %then %result=coti
          %repeat
        %finish
```

```
116     %if cotp=128 %then fault("CONSTANT TABLE FULL") %and %stop
117     cot(cotp)=const
118     cotp=cotp+1
119     %result=cotp-1
120   %end
121   !-----------------------------------------------------------------
122   %externalroutine pushtag(%integer ident,form,type,dim,level,rad)
123   %integer tagi
124     %if taglink(ident)#0 %and tag(taglink(ident))>>16&16_f=level %then %c
125+     fault("NAME ".name(ident)." DECLARED TWICE")
126     tagi=newtag
127     tag(tagi)=form<<28!type<<24!dim<<20!level<<16!rad
128     link(tagi)=taglink(ident)
129     taglink(ident)=tagi
130     tagi=newtag
131     tag(tagi)=ident
132     link(tagi)=namelist(level)
133     namelist(level)=tagi
134   %end
135   !-----------------------------------------------------------------
136   %externalroutine poptags
137   %integer cell,ident,nametag,params
138   %string(63) s
139     %if tagsopt=1 %then newline
140     cell=namelist(level)
141     %while cell#0 %cycle
142       ident=tag(cell)
143       cell=returntag(cell)
144       nametag=tag(taglink(ident))
145       taglink(ident)=returntag(taglink(ident))
146       %if tagsopt=1 %then %start
147         s=name(ident)
148         print string(strint(ident,3)."   ".s)
149         spaces(10-length(s))
150         print string(strhex(nametag))
151       %finish
152       %if nametag>>28=4 %then %start    ;! procedure type
153         params=nametag>>20&16_f
154         %while params#0 %cycle
155           %if tagsopt=1 %then print string("
156+                        ".strhex(tag(taglink(ident))))
157           taglink(ident)=returntag(taglink(ident))
158           params=params-1    ;! pop up parameter tags
159         %repeat
160       %finish
161       %if tagsopt=1 %then newline
162       %if taglink(ident)=0 %then namedp=namedlink(ident) %c
163+       %and namedlink(ident)=0    ;! backtrack name dictionary
164     %repeat
165     %if tagsopt=1 %then newline
166     namelist(level)=0
167   %end
168   !-----------------------------------------------------------------
169   %externalintegerfn getlabel(%integer constp)
170   %integer label
171     label=a(constp+1)
172     %if label>9999 %then fault("LABEL ".strint(label,1)." TOO LARGE") %c
173+     %and %result=-1 %else %result=label
174   %end
175   !-----------------------------------------------------------------
```

FILL LABEL

```
176  %externalroutine filllabel(%integer label)
177  %integer cell
178    %return %if label<0    ;! for conditional statements
179    cell=branchlist(level)
180    %while cell#0 %cycle
181      %if tag(cell)>>16=label %then %start
182        %if tag(cell)&16_8000=0 %then fault("DUPLICATE LABEL ". %c
183+         strint(label,1)) %else %start
184          dump("FILL",strint(label,1),strint(tag(cell)&16_7fff,1),nextcad)
185          tag(cell)=label<<16!nextcad
186        %finish
187        %return
188      %finish
189      cell=link(cell)
190    %repeat
191    cell=newtag
192    link(cell)=branchlist(level)
193    branchlist(level)=cell
194    tag(cell)=label<<16!nextcad
195  %end
196  !-----------------------------------------------------------------
```

FILL BRANCH

```
197  %externalintegerfn fillbranch(%integer label)
198  %integer cell,cad
199    %result=0 %if label<0
200    cell=branchlist(level)
201    %while cell#0 %cycle
202      %if tag(cell)>>16=label %then %start
203        cad=tag(cell)&16_7fff
204        %if tag(cell)&16_8000#0 %then tag(cell)=label<<16!16_8000!nextcad
205        %result=cad
206      %finish
207      cell=link(cell)
208    %repeat
209    cell=newtag
210    link(cell)=branchlist(level)
211    branchlist(level)=cell
212    tag(cell)=label<<16!16_8000!nextcad
213    %result=0
214  %end
215  !-----------------------------------------------------------------
```

POP LABELS

```
216  %externalroutine poplabels
217  %integer cell
218    cell=branchlist(level)
219    %while cell#0 %cycle
220      %if tag(cell)&16_8000#0 %then fault("LABEL ".strint(tag(cell)>>16,%c
221+       1)." NOT SET (BRANCH LIST ".strint(tag(cell)&16_7fff,1).")")
222      cell=returntag(cell)
223    %repeat
224    branchlist(level)=0
225  %end
226  !-----------------------------------------------------------------
```

NEXT PLABEL

```
227  %externalintegerfn nextplabel
228  %owninteger plabel=9999
229    plabel=plabel+1
230    %result=plabel
231  %end
232  !-----------------------------------------------------------------
```

PUSH START

```
233  %externalroutine pushstart(%integer flag,plab)
234  %integer cell
235    cell=newtag
```

```
236     tag(cell)=flag<<16!plab&16_ffff     ;! plab may be -1
237     link(cell)=startlist(level)
238     startlist(level)=cell
239   %end
240   !-----------------------------------------------------------------
POP   241   %externalroutine popstart(%integername flag,plab)
START 242   %integer cell
243     cell=startlist(level)
244     %if cell=0 %then %start
245       fault("SPURIOUS %FINISH")
246       flag=0
247       plab=0
248     %finish %else %start
249       flag=tag(cell)>>16
250       plab=tag(cell)&16_ffff
251       %if plab=16_ffff %then plab=-1
252       startlist(level)=returntag(cell)
253     %finish
254   %end
255   !-----------------------------------------------------------------
CLEAR 256   %externalroutine clearstart
START 257   %integer cell
258     cell=startlist(level)
259     %while cell#0 %then fault("%FINISH MISSING") %and cell=returntag(cell)
260     startlist(level)=0
261   %end
262   !-----------------------------------------------------------------
ENTER 263   %externalintegerfn enter
264   %string(4) base
265   %integer cad
266     %if level=1 %then %start
267       %if nextcad#0 %then fault("%BEGIN NOT FIRST STATEMENT")
268       dump("LDA","COT","",0)     ;! cot base address to be filled
269       dump("LDA","DR1","",0)     ;! stack base address to be filled
270       base="DR1"
271     %finish %else %start
272       dump("STR",display(level),"STP",0)
273       dump("LDA",display(level),"STP",0)
274       dump("STR","WK","STP",1)
275       base="STP"
276     %finish
277     cad=nextcad
278     dump("LDA","STP",base,0)     ;! static allocation to be filled
279     nextrad(level)=2
280     %result=cad
281   %end
      282   !-----------------------------------------------------------------
DUMP  283   %externalroutine dumpreturn
RETURN 284     dump("LDA","STP",display(level),0)
285     dump("LOAD",display(level),"STP",0)
286     dump("LOAD","WK","STP",1)
287     dump("B","","WK",0)
288   %end
289   !-----------------------------------------------------------------
ARRAY 290   %externalroutine array(%integer arrayp)
291   %integer namep,actualp,exprp,exprsp,ident,nametag
292     namep=a(arrayp+1)
293     actualp=a(arrayp+2)
294     ident=a(namep+1)
295     %if a(actualp)=1 %then %start
```

```
296       exprp=a(actualp+1)
297       exprsp=a(actualp+2)
298       expr(exprp)
299       nametag=tag(taglink(ident))
300       dump("ADD","ACC",display(nametag>>16&16_f),nametag&16_ffff)
301       %if a(exprsp)=1 %then fault("ARRAY ".name(ident)." HAS EXTRA INDEX")
302     %finish %else fault("ARRAY ".name(ident)." HAS NO INDEX")
303   %end
304   !-------------------------------------------------------------------
PROC 305   %externalroutine proc(%integer procp)
306   %string(4) opn,base
307   %integer namep,ident,nametag,ptagl,l,actualp,exprp,unaryp,operandp, %c
308+    npars,ptag,pnamep,pident,pnametag,pactualp,disp,exprrestp,exprsp, %c
309+    oldparams
310     %if params>2 %then dump("LDA","STP","STP",params)
311     oldparams=params
312     params=2
313     namep=a(procp+1)
314     actualp=a(procp+2)
315     ident=a(namep+1)
316     l=taglink(ident)
317     nametag=tag(l)
318     ptagl=link(l)
319     npars=nametag>>20&16_f
320     %if npars=0 %then %start
321       %if a(actualp)=1 %then fault(name(ident)." HAS PARAMETERS") %c
322+        %and %return
323     %finish %else %start
324       %if a(actualp)=2 %then fault(name(ident)." MISSING PARAMETERS") %c
325+        %and %return
326       exprp=a(actualp+1)
327       exprsp=a(actualp+2)
328       %cycle    ;! for each parameter
329         ptag=tag(ptagl)
330         %if ptag>>28=0 %then expr(exprp) %else %start
331           unaryp=a(exprp+1)
332           operandp=a(exprp+2)
333           exprrestp=a(exprp+3)
334           %unless a(unaryp)=4 %and a(operandp)=1 %and a(exprrestp)=2 %c
335+            %then fault("NOT A %NAME PARAMETER") %else %start
336           pnamep=a(operandp+1)
337           pactualp=a(operandp+2)
338           pident=a(pnamep+1)
339           %if taglink(pident)=0 %then fault(name(pident). %c
340+            " NOT DECLARED") %else %start
341           pnametag=tag(taglink(pident))
342           %if pnametag>>28=4 %then fault(name(pident). %c
343+            " NOT A %NAME") %else %start
344           base=display(pnametag>>16&16_f)
345           disp=pnametag&16_ffff
346           %if ptag>>28=1 %then %start    ;! %name
347             %if pnametag>>28>=2 %then array(operandp) %else %start
348               %if pnametag>>28=1 %then opn="LOAD" %else opn="LDA"
349               dump(opn,"ACC",base,disp)
350               %if a(pactualp)=1 %then fault(name(pident). %c
351+                " DECLARED AS SCALAR")
352             %finish
353           %finish %else %start
354             dump("LOAD","ACC",base,disp)    ;! %array
355             %if a(pactualp)=1 %then fault("%ARRAYNAME ". %c
```

```
356+                    name(pident)." HAS INDEX")
357                   %finish
358                 %finish
359               %finish
360             %finish
361           %finish
362           dump("STR","ACC","STP",params)
363           params=params+1
364           npars=npars-1
365           %if npars=0 %then %start
366             %if a(exprsp)=1 %then fault(name(ident)." HAS EXTRA PARAMETERS")
367             %exit
368           %finish
369           ptagl=link(ptagl)
370           %if a(exprsp)=2 %then fault(name(ident). %c
371+            " IS MISSING PARAMETERS") %and %exit
372           exprp=a(exprsp+1)
373           exprsp=a(exprsp+2)
374         %repeat
375       %finish
376       ! external i/o routines at level 0
377       %if nametag>>16&16_f=0 %then base="EXT" %else base=""
378       dump("BAL","WK",base,nametag&16_ffff)
379       params=oldparams
380       %if params>2 %then dump("SUB","STP","COT",getcoti(params))
381     %end
382     !-----------------------------------------------------------------
383     %externalroutine endofprog
384     %integer i
385       dump("FILL","COT","0",nextcad)
386       i=0
387       %while i<cotp %cycle
388         dump("CONST","","",cot(i))
389         i=i+1
390       %repeat
391       dump("FILL","STACK","1",nextcad)
392       print string("
393+         $".strint(faults,4)." FAULTS IN PROGRAM
394+ ")
395       %stop
396     %end
397     %endoffile
Code 10012 bytes  Glap 1832 bytes  Diags 2391 bytes   Total size 14235 bytes
   345 statements compiled
```

END OF PROG

Source file: SKIMPE.IMP Compiled on 25-OCT-1979 at 09:41:31

Computer Science IMP77 Compiler. Version 6.01

```
1  %externalintegerarrayspec a(1:500)
2  %externalintegerarrayspec taglink(0:255)
3  %externalintegerarrayspec tag(1:512)
4  %externalstring(4)%arrayspec display(0:15)
5  %externalintegerspec level,condflag,expropt
6  !-------------------------------------------------------------------------
7  %externalstring(255)%fnspec strint(%integer n,p)
8  %externalstring(8)%fnspec strhex(%integer n)
9  %externalroutinespec fault(%string(63) s)
10 %externalstring(255)%fnspec name(%integer ident)
11 %externalroutinespec dump(%string(7) opn,reg,base,%integer disp)
12 %externalintegerfnspec getwork
13 %externalroutinespec returnwork(%integer work)
14 %externalroutinespec proc(%integer ap)
15 %externalroutinespec array(%integer ap)
16 %externalintegerfnspec getcoti(%integer const)
17 !-------------------------------------------------------------------------
EXPR
18 %externalroutine expr(%integer exprp)
19 %integerfnspec totree(%integer exprp)
20 %routinespec evaluate(%integer nodep)
21 %integerarray tree(1:64)
22 %integer treep,treenode,treenode1,treenode2,testp,expr1p,expr2p,compp,%c
23+   i,j,l
24 %constintegerarray reversecomp(1:6)=1,2,5,6,3,4
25   treep=1
26   %if condflag=0 %then treenode=totree(exprp) %else %start
27     condflag=0
28     testp=exprp    ;! for <test>=<expr><comp><expr>
29     expr1p=a(testp+1)
30     compp=a(testp+2)
31     expr2p=a(testp+3)
32     treenode1=totree(expr1p)
33     treenode2=totree(expr2p)
34     %if tree(treenode1)=-4 %and tree(treenode1+1)=0 %then %start
35       a(compp)=reversecomp(a(compp))
36       treenode=treenode2
37     %finish %else %start
38       %if tree(treenode2)=-4 %and tree(treenode2+1)=0 %then %c
39+        treenode=treenode1 %else %start
40         tree(treep)=10    ;! -
41         tree(treep+1)=treenode1
42         tree(treep+2)=treenode2
43         treenode=treep
44       %finish
45     %finish
46   %finish
47   %if expropt=1 %then %start
48     newline
49     %if 0<tree(treenode)<=10 %then l=treenode+2 %else l=treenode+1
50     j=0
51     %for i=1,1,l %cycle
52       write(tree(i),4)
53       j=j+5
54       %if i=treenode %then print string("*") %and j=j+1
55       %if tree(i)=-3 %then i=i+1 %and print string("  ". %c
```

```
56+         strhex(tree(i))) %and j=j+10
57          %if j>70 %then newline %and j=0
58         %repeat
59         newlines(2)
60       %finish
61       evaluate(treenode)
62       %return
63     !-------------------------------------------------------------------
64     %integerfn totree(%integer exprp)
65     ! create tree form of expression
66     %routinespec pseval(%integer type,datum)
67     %integerarray os(1:4),ps(1:5)     ;! operator & pseudo-evaluation stacks
68     %integer osp,psp,unaryp,operandp,exprrestp,opp,namep,actualp, %c
69+      ident,nametag
70     %constintegerarray prec(1:12)=3,3,2,1,1,3,2,2,1,1,1,4
71     ! <<,>>,&,!!,!,**,/,*,+,-,-(unary),\
72       unaryp=a(exprp+1)
73       operandp=a(exprp+2)
74       exprrestp=a(exprp+3)
75       %if a(unaryp)<=2 %then os(1)=a(unaryp)+10 %and osp=1 %else osp=0
76       psp=0
77       %cycle     ;! for each operand
78         %if a(operandp)=1 %then %start     ;! <name><actual>
79           namep=a(operandp+1)
80           actualp=a(operandp+2)
81           ident=a(namep+1)
82           %if taglink(ident)=0 %then %start
83             fault(name(ident)." NOT DECLARED")
84             pseval(-3,0)     ;! pseval dummy tag
85           %finish %else %start
86             nametag=tag(taglink(ident))
87           %if nametag>>28<=1 %then %start     ;! scalar variable
88             %if a(actualp)=1 %then %start
89               fault("SCALAR ".name(ident)." HAS PARAMETER")
90               pseval(-3,0)
91             %finish %else pseval(-3,nametag)
92           %finish %else %start
93             %if nametag>>28<=3 %then pseval(-2,operandp) %else %start
94               %if nametag>>24&16_f=0 %then %start
95                 fault("ROUTINE NAME ".name(ident)." IN EXPRESSION")
96                 pseval(-3,0)
97               %finish %else pseval(-1,operandp)
98             %finish
99           %finish
100        %finish
101      %finish %else %start
102        %if a(operandp)=2 %then pseval(-4,a(a(operandp+1)+1)) %else %c
103+          psp=psp+1 %and ps(psp)=totree(a(operandp+1))
104      %finish
105      %if a(exprrestp)=2 %then %exit     ;! no more operands
106      opp=a(exprrestp+1)
107      operandp=a(exprrestp+2)
108      exprrestp=a(exprrestp+3)
109      %while osp>0 %and prec(a(opp))<=prec(os(osp)) %then %c
110+        pseval(os(osp),0) %and osp=osp-1     ;! unstack while prec(new op)<=
111      osp=osp+1     ;! stack new operator
112      os(osp)=a(opp)
113    %repeat
114    %while osp>0 %then pseval(os(osp),0) %and osp=osp-1     ;! unstack rest
115    %result=ps(1)
```

TO TREE

PS EVAL

```
116  !-------------------------------------------------------------------------
117  %routine pseval(%integer type,datum)
118  %routinespec store(%integer t)
119  %integer nodep
120    nodep=treep
121    store(type)
122    %if type>0 %then %start    ;! operator
123      %if type>10 %then store(ps(psp)) %else store(ps(psp-1)) %and %c
124+       store(ps(psp)) %and psp=psp-1
125    %finish %else store(datum) %and psp=psp+1
126    ps(psp)=nodep
127  !-------------------------------------------------------------------------
```

STORE

```
128  %routine store(%integer t)
129    %if treep>64 %then fault("EXPRESSION TOO LONG") %and %stop
130    tree(treep)=t
131    treep=treep+1
132  %end
133  %end
134  %end
135  !-------------------------------------------------------------------------
```

EVALUATE

```
136  %routine evaluate(%integer nodep)
137  ! dump code to evaluate expression
138  %routinespec opn(%integer op,p)
139  %conststring(4)%array strop(0:12)="LOAD","SHL","SHR","AND","XOR","OR",
140+   "EXP","DIV","MLT","ADD","SUB","NEG","NOT"
141  %constintegerarray commut(1:10)=0,0,1,1,1,0,0,1,1,0
142  %integer op,opd1p,opd2p,work
143    %if tree(nodep)<0 %then opn(0,nodep) %and %return    ;! operand
144    op=tree(nodep)
145    opd1p=tree(nodep+1)
146    %if op>10 %then %start    ;! unary operator
147      %if tree(opd1p)>=-2 %then evaluate(opd1p) %else opn(0,opd1p)
148      dump(strop(op),"ACC","",0)
149      %return
150    %finish
151    opd2p=tree(nodep+2)
152    %if tree(opd1p)>=-2 %then %start       ;! operand1 a node
153      %if tree(opd2p)>=-2 %then %start    ;! operand2 a node
154        evaluate(opd2p)
155        work=getwork
156        dump("STR","ACC",display(level),work)
157        evaluate(opd1p)
158        dump(strop(op),"ACC",display(level),work)
159        returnwork(work)
160      %finish %else %start
161        evaluate(opd1p)
162        opn(op,opd2p)
163      %finish
164    %finish %else %start
165      %if tree(opd2p)>=-2 %then %start
166        evaluate(opd2p)
167        %if commut(op)#0 %then opn(op,opd1p) %and %return
168        work=getwork
169        dump("STR","ACC",display(level),work)
170        opn(0,opd1p)
171        dump(strop(op),"ACC",display(level),work)
172        returnwork(work)
173      %finish %else %start
174        opn(0,opd1p)
175        opn(op,opd2p)
```

```
      %finish
    %finish
    %return
!-----------------------------------------------------------------------
%routine opn(%integer op,p)
! dump object code for simple operation
%string(7) base
%integer type,datum,disp
  type=tree(p)
  datum=tree(p+1)
  %if type=-1 %then proc(datum) %else %start     ;! procedure
    %if type=-2 %then %start                     ;! array
      array(datum)
      dump("LOAD","ACC","ACC",0)
    %finish %else %start
      %if type=-3 %then %start                   ;! scalar
        base=display(datum>>16&16_f)
        disp=datum&16_ffff
        %if datum>>28=1 %then %start             ;! %name type
          dump("LOAD","WK",base,disp)
          dump(strop(op),"ACC","WK",0)
        %finish %else dump(strop(op),"ACC",base,disp)
      %finish %else %start    ;! constant
        %if op>0 %or datum>16_ffff %then dump(strop(op),"ACC", %c
          "COT",getcoti(datum)) %else dump("LDA","ACC","",datum)
      %finish
    %finish
  %finish
%end
%end
%endoffile
?STRINT unused
Code 5164 bytes  Glap 680 bytes  Diags 1023 bytes   Total size 6867 bytes
   187 statements compiled
```

OPN

Source file: SKIMPAI.IMP Compiled on 30-OCT-1979 at 10:38:02

Computer Science IMP77 Compiler. Version 6.01

SKIMPAI

```
 1  %externalroutine skimpai(%string(63) file)
 2  %string(15)%fnspec readmn(%integername sep)
 3  %routinespec dump(%integer on,rn,bn,dn)
 4  %integerfnspec intstr(%string(15) s)
 5  %string(15)%fnspec strint(%integer n)
 6  %routinespec fault(%string(255) mess)
 7  %routinespec fail(%string(255) mess)
 8  %routinespec monitor
 9  %conststring(7)%array imn(0:32)="STOP","LOAD","LDA","ADD","SUB","MLT",
10+   "DIV","EXP","STR","NEG","NOT","SHL","SHR","AND","OR","XOR","BAL","B",
11+   "BZ","BNZ","BG","BNG","BL","BNL","ADDF","SUBF","MLTF","DIVF","EXPF",
12+   "NEGF","FLT","FILL","CONST"
13  %string(15)%array rmn(0:15)
14  %integerarray r(0:15)
15  %integerarray s(0:4095)
16  %string(15) opn,reg,base,disp
17  %constinteger sl=4095
18  %constinteger datamask=2_00111110000000011111000111110l0
19  %integer rp,sp,mon,tron,troff,on,rn,bn,dn,sep,i,faults,inst,trace,pc, %c
20+   pcl,tracepc,tracen,efad,stackb,stpr,accr,cyc,ic
21  %switch ins(0:31)
22  %switch ext(1:11)
23    %if file->("/TR ").file %then trace=1 %and tracen=0 %else %start
24      %if file="" %then print string("file ?
25+ ") %and %stop
26     trace=0
27    %finish
28    open input(2,file.".lis")
29    select input(2)
30    select output(1)
31    rmn(0)=""
32    r(0)=0
33    rp=1
34    sp=0
35    mon=0
36    tron=0
37    troff=0
38    faults=0
39  instr:! process next instruction
40    read symbol(i) %until i='$'
41    %cycle
42      i=next symbol
43      %if i=' ' %then skip symbol %else %exit
44    %repeat
45    %if '0'<=i<='9' %then %start    ;! "FAULTS IN PROGRAM"
46      %if i>'0' %then fault("PROGRAM WAS FAULTY") %and %stop
47      %if faults>0 %then fault("ASSEMBLY FAULTY") %and %stop
48      read symbol(i) %until i=nl
49      ->interpret
50    %finish
51    opn=readmn(sep)
52    %for on=0,1,32 %cycle
53      %if opn=imn(on) %then ->gotopn
54    %repeat
55    %if opn="MONITOR" %then mon=1 %and ->instr
```

```
56    %if opn="TRON" %then tron=1 %and ->instr
57    %if opn="TROFF" %then troff=1 %and ->instr
58    fault("INVALID OPERATION : ".opn.tostring(sep))
59    ->instr
60  gotopn:! get operands
61    %if sep#',' %then fault("INVALID FORMAT : ".opn.tostring(sep)) %c
62+     %and ->instr
63    reg=readmn(sep)
64    %if sep#',' %then fault("INVALID FORMAT : ".opn.","."reg. %c
65+     tostring(sep)) %and ->instr
66    base=readmn(sep)
67    %if sep#',' %then fault("INVALID FORMAT : ".opn.","."reg. %c
68+     ","".base.tostring(sep)) %and ->instr
69    disp=readmn(sep)
70    %if opn="FILL" %then %start
71      bn=intstr(base)
72      %if bn<0 %then fault("INVALID FILL : ".reg.","."base) %and ->instr
73      dn=intstr(disp)
74      %unless 0<=dn<=16_FFFF %then fault("INVALID FILL : ".reg. %c
75+       ","".base.","."disp) %and ->instr
76      cyc=0
77      %cycle
78        cyc=cyc+1
79        %if bn>=sp %or cyc>sp %then fault("INVALID FILL LIST AT ". %c
80+         strint(sp)) %and ->instr
81        i=bn
82        bn=s(bn)&16_FFFF
83        s(i)=s(i)&16_FFFF0000!dn
84      %repeat %until bn=0
85      %if reg="COT" %then pcl=dn-1     ;! program counter limit
86      ->instr
87    %finish
88    %if opn="CONST" %then %start
89      dn=intstr(disp)
90      %if dn<0 %then fault("INVALID CONSTANT : ".disp) %and ->instr
91      dump(0,0,0,dn)
92      ->instr
93    %finish
94      %if reg="" %and opn#"B" %and opn#"STOP" %then %c
95+       fault("REGISTER MISSING AT ".strint(sp)) %and rn=0 %and ->gotreg
96    %for rn=0,1,rp-1 %cycle
97      %if reg=rmn(rn) %then ->gotreg
98    %repeat
99    %if rp=16 %then fault("EXCESS REGISTER : ".reg) %and rn=0 %c
100+      %else rmn(rp)=reg %and rn=rp %and rp=rp+1
101 gotreg:%if base="EXT" %and opn="BAL" %then on=31 %and bn=0 %and->gotbase
102 %for bn=0,1,rp-1 %cycle
103     %if base=rmn(bn) %then ->gotbase
104   %repeat
105   %if rp=16 %then fault("EXCESS REGISTER : ".base) %and bn=0 %c
106+     %else rmn(rp)=base %and bn=rp %and rp=rp+1
107 gotbase:dn=intstr(disp)
108   %unless 0<=dn<=16_FFFF %then fault("INVALID DISPLACEMENT : ". %c
109+     disp) %and dn=0
110   dump(on,rn,bn,dn)
111   ->instr
112 !-----------------------------------------------------------------------
113 interpret:! interpretation section
114   select input(1)
115   stpr=0
```

```
116     accr=0
117     %for i=0,1,rp-1 %cycle
118       %if rmn(i)="STP" %then stpr=i
119       %if rmn(i)="ACC" %then accr=i
120       r(i)=0
121     %repeat
122     stackb=sp
123     %while sp<=sl %then s(sp)=16_80000000 %and sp=sp+1
124     ic=0
125     pc=0
126   exi:! execute instruction
127     %if ic>10000 %then fail("10000 INSTRUCTIONS EXECUTED")
128     %unless 0<=pc<=pcl %then fail("PC OUT OF BOUNDS")
129     inst=s(pc)
130     tracepc=pc
131     pc=pc+1
132     %if inst&16_80000000#0 %then monitor
133     %if inst&16_40000000#0 %then trace=1 %and tracen=0
134     %if inst&16_20000000#0 %then trace=0
135     on=inst>>24&16_1F
136     rn=inst>>20&16_F
137     bn=inst>>16&16_F
138     dn=inst&16_FFFF
139     efad=r(bn)+dn
140     %if 1<<on&datamask#0 %then %start
141       %unless pcl<efad<=sl %then fail("DATA ADDRESS ". %c
142+        strint(efad)." OUT OF BOUNDS")
143       %if s(efad)=16_80000000 %then fail("UNASSIGNED VARIABLE AT ". %c
144+        strint(efad))
145     %finish %else %start
146       %if on=8 %then %start
147         %unless stackb<=efad<=sl %then fail("STR ADDRESS ". %c
148+          strint(efad)." OUT OF BOUNDS")
149       %finish
150     %finish
151     ->ins(on)
152   ins(0):print string("
153+ STOPPED AT ".strint(tracepc)."$, ".strint(ic)." INSTRUCTIONS EXECUTED
154+ ") %and %stop
155   ins(1):r(rn)=s(efad) ; ->tracep
156   ins(2):%if rn=stpr %and efad<r(stpr) %then %start
157       ! clear old area back to unassigned
158       %for i=efad+2,1,r(stpr)+2 %cycle
159         s(i)=16_80000000 %unless i>sl
160       %repeat
161     %finish
162     r(rn)=efad
163     ->tracep
164   ins(3):r(rn)=r(rn)+s(efad) ; ->tracep
165   ins(4):r(rn)=r(rn)-s(efad) ; ->tracep
166   ins(5):r(rn)=r(rn)*s(efad) ; ->tracep
167   ins(6):r(rn)=r(rn)//s(efad) ; ->tracep
168   ins(7):r(rn)=r(rn)\\s(efad) ; ->tracep
169   ins(8):s(efad)=r(rn) ; ->tracep
170   ins(9):r(rn)=-r(rn) ; ->tracep
171   ins(10):r(rn)=\r(rn) ; ->tracep
172   ins(11):r(rn)=r(rn)<<s(efad) ; ->tracep
173   ins(12):r(rn)=r(rn)>>s(efad) ; ->tracep
174   ins(13):r(rn)=r(rn)&s(efad) ; ->tracep
175   ins(14):r(rn)=r(rn)!s(efad) ; ->tracep
```

```
176  ins(15):r(rn)=r(rn)!!s(efad) ; ->tracep
177  ins(16):r(rn)=pc ; pc=efad ; ->tracep
178  ins(17):pc=efad ; ->tracep
179  ins(18):%if r(rn)=0 %then pc=efad ; ->tracep
180  ins(19):%if r(rn)#0 %then pc=efad ; ->tracep
181  ins(20):%if r(rn)>0 %then pc=efad ; ->tracep
182  ins(21):%if r(rn)<=0 %then pc=efad ; ->tracep
183  ins(22):%if r(rn)<0 %then pc=efad ; ->tracep
184  ins(23):%if r(rn)>=0 %then pc=efad ; ->tracep
185  ins(24):real(addr(r(rn)))=real(addr(r(rn)))+real(addr(s(efad)))
186    ->tracep
187  ins(25):real(addr(r(rn)))=real(addr(r(rn)))-real(addr(s(efad)))
188    ->tracep
189  ins(26):real(addr(r(rn)))=real(addr(r(rn)))*real(addr(s(efad)))
190    ->tracep
191  ins(27):real(addr(r(rn)))=real(addr(r(rn)))/real(addr(s(efad)))
192    ->tracep
193  ins(28):real(addr(r(rn)))=real(addr(r(rn)))\s(efad)
194    ->tracep
195  ins(29):real(addr(r(rn)))=-real(addr(r(rn))) ; ->tracep
196  ins(30):real(addr(r(rn)))=r(rn) ; ->tracep
197  ins(31):%if stpr=0 %then fail("REGISTER STP NOT DEFINED FOR I/O". %c
198+      " ROUTINE CALL")
199    ->ext(dn)
200  ext(1):read symbol(s(s(r(stpr)+2))) ; ->tracep
201  ext(2):%if accr=0 %then fail("REGISTER ACC NOT DEFINED FOR ". %c
202+      "'NEXT SYMBOL' I/O FUNCTION CALL")
203    r(accr)=next symbol
204    ->tracep
205  ext(3):skip symbol ; ->tracep
206  ext(4):print symbol(s(r(stpr)+2)) ; ->tracep
207  ext(5):space ; ->tracep
208  ext(6):spaces(s(r(stpr)+2)) ; ->tracep
209  ext(7):newline ; ->tracep
210  ext(8):newlines(s(r(stpr)+2)) ; ->tracep
211  ext(9):print symbol(12)
212  ext(10):read(s(s(r(stpr)+2))) ; ->tracep
213  ext(11):write(s(r(stpr)+2),s(r(stpr)+3))
214  tracep:ic=ic+1
215    %if trace#0 %then %start
216      write(tracepc,4)
217      print symbol('$')
218      tracen=tracen+1
219      %if tracen=10 %then newline %and tracen=0
220    %finish
221  ->exi
222  !-----------------------------------------------------------------
READ
MN
223  %string(15)%fn readmn(%integername sep)
224  %string(255) s
225  %integer flag
226    s=""
227    flag=0
228    %cycle
229      read symbol(sep)
230      %if '0'<=sep<='9' %or 'A'<=sep<='Z' %then %start
231        %if length(s)<255 %then s=s.tostring(sep) %else flag=1
232      %finish %else %start
233        %if sep#' ' %then %exit
234      %finish
235    %repeat
```

```
236     %if length(s)>15 %or flag#0 %then fault("INVALID MNEMONIC : ".s) %c
237+       %and %result="" %else %result=s
238   %end
239   !-----------------------------------------------------------------------
DUMP  240   %routine dump(%integer on,rn,bn,dn)
241     %if sp>sl %then fault("PROGRAM TOO BIG") %and %stop
242     s(sp)=mon<<31!tron<<30!troff<<29!on<<24!rn<<20!bn<<16!dn
243     sp=sp+1
244     mon=0
245     tron=0
246     troff=0
247   %end
248   !-----------------------------------------------------------------------
INTSTR 249   %integerfn intstr(%string(15) s)
250   %integer value,d,i
251     value=0
252     %for i=1,1,length(s) %cycle
253       d=charno(s,i)
254       %unless '0'<=d<='9' %then %result=-1
255       value=value*10+d-'0'
256     %repeat
257     %result=value
258   %end
259   !-----------------------------------------------------------------------
STRINT 260   %string(15)%fn strint(%integer n)
261   %string(15) r
262   %string(1) s
263     r=""
264     %if n<0 %then n=-n %and s="-" %else s=""
265     r=tostring(n-n//10*10+'0').r %and n=n//10 %until n=0
266     %result=s.r
267   %end
268   !-----------------------------------------------------------------------
FAULT 269   %routine fault(%string(255) mess)
270     print string("
271+ ".strint(sp)."$   ".mess."
272+ ")
273     faults=faults+1
274   %end
275   !-----------------------------------------------------------------------
FAIL  276   %routine fail(%string(255) mess)
277     print string("
278+ *  ".mess."
279+ PC=".strint(tracepc)."$")
280     monitor
281     %stop
282   %end
283   !-----------------------------------------------------------------------
MONITOR 284   %routine monitor
285   %string(15) v
286   %integer i,j
287     newline
288     %if rp>1 %then %start
289       %for i=1,1,rp-1 %cycle
290         print string(rmn(i)."  ".strint(r(i))."
291+ ")
292       %repeat
293     %finish
294     %if stpr#0 %and r(stpr)>stackb %then %start
295       write(stackb,2)
```

```
 296       print symbol('$')
 297       j=1
 298       %for i=stackb,1,r(stpr)+17 %cycle
 299         %if s(i)=16_80000000 %then print string("   ?") %c
 300+          %else write(s(i),3)
 301         j=j+1
 302         %if j=16 %then newline %and j=0
 303       %repeat
 304       newline
 305     %finish
 306  %end
?V unused
 307  %endofprogram
Code 9580 bytes  Glap 1456 bytes  Diags 1428 bytes   Total size 12464 bytes
   299 statements compiled
```