A CONCURRENT PASCAL COMPILER FOR MINICOMPUTERS

Thesis by

Alfred C. Hartmann

In Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy

California Institute of Technology

Pasadena, California

1976

(Submitted September 22, 1975)

Acknowledgements

This compiler is the product of many fruitful hours of discussion with Per Brinch Hansen. The development of Concurrent Pascal has been supported by the National Science Foundation under grant number DCR74-17331.

Abstract

This paper describes a seven-pass compiler for the Concurrent Pascal programming language. Concurrent Pascal is an abstract programming language for computer operating systems. The language extends sequential Pascal with the monitor concept for structured concurrent programming. Compilation of Concurrent Pascal on a minicomputer is done by dividing the compiler into seven sequential passes. The passes, written in sequential Pascal, generate virtual code that can be interpreted on any 16-bit minicomputer. It has been running on a PDP-11/45 computer at Caltech since January 1975.

A Pascal Compiler for Minicomputers

Table of Contents

1. Introduction 1

2. Definitions 8

3. Pass Structure 10

4. Lexical Analysis . . . 15

5. Syntax Analysis . . . 26

6. Name Analysis 44

7. Declaration Analysis . 67

8. Body Analysis 81

9. Code Selection 93

10. Code Assembly 100

11. Interpass Topics . . . 102

12. The Virtual Machine . 107

13. Implementation 112

14. References 132

Appendix: Syntax Graphs . 133

1. Introduction

This paper describes a seven-pass compiler for Per Brinch Hansen's Concurrent Pascal [1, 2] programming language. Concurrent Pascal is an abstract programming language for computer operating systems. The language extends sequential Pascal [7] with the process, monitor, and class concepts for structured concurrent programming. A monitor is a shared data structure together with a well-defined set of operations that are the only operations possible on the data structure. Concurrent Pascal's runtime system enforces mutually exclusive access to a monitor by competing concurrent processes. A class gives a single process controlled access to a private data structure by means of a well-defined set of operations.

The Concurrent Pascal compiler has been running on a DEC PDP-11/45 computer at Caltech since January 1975. It requires 16,500 16-bit words of storage and compiles source text at the rate of 240 characters per second (about 9-10 lines per second). It generates code for an ideal virtual machine that is simulated by the real machine. The compiler is written in sequential Pascal and is easily transported to other machines.

The main contributions of this work to programming methodology are:

1. <u>This is the first implementation of the monitor concept in a high-level language.</u> The monitor concept allows the programming of operating systems with most time-dependent errors being detected at compile time. Implementing this concept in a high-level programming language makes the structured programming of operating systems possible.

2. <u>This is the first implementation of the class concept with strict access rules.</u> The compiler checks that data structures are accessed only by procedures associated with the data. This makes it possible in a large program to ignore implementation details of data types and think of them in terms of their abstract properties.

3. <u>This implementation departs from traditional block structure scope rules.</u> Concurrent Pascal supports modular construction of operating systems. The access rights of modules to other modules is controlled by the operating system designer. Pure block structure scope rules only enforce tree

structured relationships. Concurrent Pascal enforces arbitrary directed graph relationships at compile time.

4. This implementation allows selective control of the access rights of separately compiled sequential programs. The operating system designer may define the operating system interface of separately compiled programs. Many different interfaces may be defined to allow a range of access rights to the operating system. These access rights are enforced by the Concurrent Pascal compiler and by the companion sequential Pascal compiler used for the separate compilations. No run time enforcement of access rules is required.

5. This is the first multi-pass Pascal compiler. This compiler uses many (seven) small, sequential passes. Each pass performs a single well-defined part of the total compilation process. Conceptually, each pass functions as a class whose input is the intermediate code produced by the previous pass. The output is the intermediate code to be processed by the following pass.

The Gier Algol compiler [5], completed in 1962, is the best known example of a <u>many-pass compiler</u>. It inspired the Siemens Cobol compiler [3], completed in 1965.

6. <u>This implementation supports a multi-language system.</u> Formal interface definitions between Concurrent Pascal and sequential Pascal programs are provided, as mentioned previously. The companion sequential Pascal compiler, supporting a variant of the sequential Pascal language, was edited from the Concurrent Pascal compiler. This operation required one month. The compilation scheme used here can also be used to implement a variety of standard programming languages (such as Algol 60, Fortran, Cobol, and PL/I).

7. <u>This implementation performs strictly sequential input/output.</u> Large one-pass compilers in demand paging systems, multipass compilers with dynamic overlays, and compilers that "spill" symbol tables to secondary storage all perform a large amount of random access I/O. Like previous many-pass compilers, the Concurrent Pascal compiler uses passes so small that paging is unnecessary. Unlike

previous efforts, dense assembler language packing of code and data is not used. The passes operate sequentially, from first pass to last pass. Each pass reads a sequential input file and produces a sequential output file. No pass uses a complete symbol table. Only partial symbol tables are used, and these can reside entirely in main memory. The Concurrent Pascal compiler uses a disk to store the passes and the intermediate code produced by them.

8. _This is the first many-pass compiler written in a high-level language._ The Concurrent Pascal compiler is written in a sequential subset of Pascal. It is compiled by a seven-pass compiler for sequential Pascal.

9. _This is the first many-pass compiler for a structured programming language._ Current structured programming languages incorporate data structuring facilities and classes. The semantic processing of such facilities as pointers, records, and classes is spread over three separate semantic analysis passes. Semantic analysis is divided into subanalyses performed in separate passes. The clear division of labor among these passes is one of the main

contributions of this work.

10. This compiler incorporates many features of good software design. Many of the techniques of structured programming, top-down design, and systematic testing were incorporated in the compiler. A clearly defined pass division was developed using syntax graphs to specify the relationship of the passes. Then the passes were written individually starting from the last pass and ending with the first pass. The passes were individually tested using a systematically developed set of test cases. The compiler test mechanism, the same as used in the Gier Algol compiler, is a permanent part of the compiler and can be turned on to test compiler modifications or document errors discovered by users.

As many machine-dependent aspects of the compiler as possible are made into changeable constant definitions. The compiler's semantic analysis passes are isolated from the virtual machine by two code assembly passes. So not only can different real machines interpret the virtual machine, but the code assembly passes can be changed to view different virtual machines. This permits

redesign of the final instruction set without significantly affecting the compiler.

In the chapters to follow, basic terms are defined, the pass breakdown is described, each pass is described, the virtual machine is defined, and the implementation is discussed. Many of the compilation techniques used here are well-known, but, taken as a whole, this compiler is an engineering product that may serve as a prototype for industrial compiler writers. For this reason, the description of the compiler is made as self-contained as possible.

2. Definitions

The problem is to accept programs written in Concurrent Pascal [1, 2], the source language, and translate them to an equivalent representation in a machine language, the target language. Programs that solve this problem are termed compilers; compilers map the source language into the target language. Multipass compilers map the source language by degrees into the target language. The first pass of a multipass compiler maps the source language into the first intermediate language. The second pass maps the first intermediate language into the second intermediate language. This process continues until the last pass maps the final intermediate language into the target language. An instance of a source program is termed the source text, its intermediate versions are the intermediate code, and its target program is the final code.

The source text is a file of characters that represents a Concurrent Pascal program. A program consists of declarations and a body. The declarations assign names to constants, types, variables, and routines. The body contains statements to be executed by the machine.

The intermediate code is a file of integers. Each integer is either an operator in the intermediate language, or an argument of an operator.

The final code consists of instructions for a machine. The machine comprises a _program store_ and a _data store_. The program store contains the code. The data store contains the program's constants, variables, and expressions.

The process of _compilation_ consists of:

1. _lexical analysis_: recognizing the symbols of Pascal;

2. _syntax analysis_: checking the program syntax;

3. _semantic analysis_: checking the program semantics; and

4. _code assembly_: generating machine code.

In a multipass compiler, it's convenient to use this functional division as a guideline for pass division. A compiler might consist of four passes which perform the four functions above. Or it might consist of two passes each performing a pair of functions. The Concurrent Pascal compiler numbers seven passes, including three passes for semantic analysis and two passes for code assembly.

3. Pass Structure

The compiler comprises seven passes:

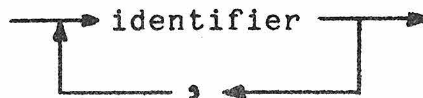
1. lexical analysis
2. syntax analysis
3. name analysis
4. declaration analysis
5. body analysis
6. code selection
7. code assembly

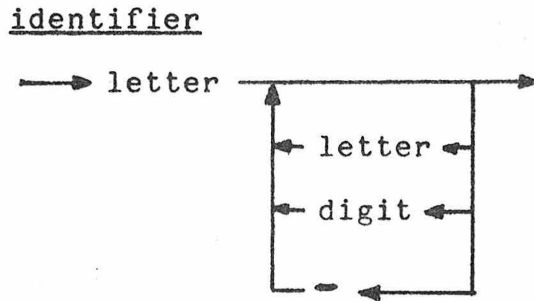
It deals with eight languages: the source language, the six intermediate languages, and the target language. In the design of a compiler the source and target languages are normally given, and it remains to define the intermediate languages. In this project we started with a clean slate. The source language was defined first. It is essentially the sequential Pascal language [7] extended with classes, monitors, and processes [1, 2]. Next the target language was designed. Borrowing from Niklaus Wirth's work on portable Pascal compilers, our target language is the language of an ideal virtual machine. This machine, designed by Per Brinch Hansen, is tailored to Concurrent Pascal. It is simulated by the real machine, a Digital Equipment Corporation PDP-11/45. After this the six intermediate languages were defined, starting

with the last intermediate language and ending with the first intermediate language. Each pass is now defined as a separate compiler in terms of its input language and its output code. In particular, the details of data structures and procedures used within a given pass are irrelevant to other passes. Once the pass breakdown and intermediate languages are determined, very few major decisions remain in the design. Given this importance, a convenient means of specifying these languages is essential. Brinch Hansen chose the syntax graph of Wirth [7] to define the intermediate languages.

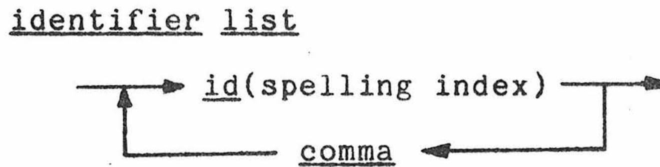
Syntax graphs are directed graphs with nodes that define the syntactic elements of the language. Operators are underscored. They may be followed by arguments enclosed in parentheses. For example, the syntax of an identifier list in the source language is:

identifier list





In the first intermediate language the same construct appears as:



The input and output graphs of lexical analysis shows that this pass converts identifiers from a string of characters into a numeric index. These graphs clarify the function of each pass.

Lexical analysis transforms the program into a sequence of integers representing identifiers, constants, and operators. Unique identifiers are replaced by unique spelling indices. These integers are easier for later passes to recognize, lookup, and switch on than the original character representation of a program.

Syntax analysis checks the syntax of the first intermediate code. The output of syntax analysis is postfix notation (operands followed by operators). Syntax analysis eliminates redundant operators and replaces ambiguous operators by unique ones. The output is syntactically correct independent of what the input is.

Name analysis converts spelling indices to unique name indices. Because of the block structure of Pascal, the same identifier may be used with different meanings. Name analysis resolves this ambiguity.

Declaration analysis enforces the semantic rules of declarations. It assigns virtual addresses to all variables and analyzes data types. This information is distributed in the body of the program.

Body analysis checks the compatibility of operand types and operators in statements. Operator ambiguities are resolved, and the resulting intermediate code is nearly ready for the machine.

It remains for this code to be "assembled". This process consists of computing the storage requirements of blocks, and replacing symbolic labels by program addresses. A classic two-pass design is used for this assembly phase. The first assembly pass, code selection,

assigns addresses to labels and places them in a table that survives to the next pass. The second assembly pass, _code assembly_, replaces program labels in the code by their addresses from the table. The resultant code is the final code for the machine. Two passes are required since the address of forward labels is not known in the first assembly pass.

4. Lexical Analysis

* function *

A Pascal program consists of identifiers, constants, and operators. Lexical analysis converts the source text character by character into the first intermediate code. This conversion is performed as follows:

```
initialize;
repeat
  read a character;
  classify the character by symbol group;
  collect the symbol;
  output its intermediate code
until source text exhausted
```

Each symbol begins with a unique class of character. Identifiers begin with letters; numeric constants begin with digits; string constants begin with quotation marks, and so on. Classification of characters is done most conveniently by a <u>case</u> statement. So lexical analysis can be further refined as:

```
var done: boolean; ch: char;
begin
  initialize;
  done:= false;
  repeat
    read(ch);
    case ch of
      'a'..'z': scan identifier;
      '0'..'9': scan number;
      '''': scan string constant;
      '<': scan operator;
         .
         .
         .
      ' ': skip blanks;
      '"': skip comment;
      EM : done:=true
    end "classification"
  until done
end "lexical analysis".
```

* identifier scan *

Scanning an identifier consists of collecting the identifier in a string variable, searching for it in a table of identifiers, and outputing the corresponding intermediate code. An identifier may be either a program defined identifier or a reserved word. The intermediate representation of an identifier is an <u>id</u> operator followed by the index of the identifier. The intermediate representation of a reserved word is an operator corresponding to it.

Identifiers may be one to eighty characters long. They are stored in a table together with their spelling indices. Reserved words are treated as identifiers with negative indices. The identifier table is a fixed length array (because Pascal has no dynamic arrays). To save space within the array, only the first ten characters of identifiers are stored in the table. Long identifiers are broken down into pieces of ten characters each. The first piece resides in the table entry. Additional pieces are allocated dynamically and chained to the identifier table entry. The identifier table may be defined as:

```
type
  spelling index = integer;
  piece = array [1..10] of char;
```

```
    piece ptr = @ id piece;
    id piece = record
                   part: piece;
                   next: piece ptr
               end;
    table entry = record
                     spix: spelling index;
                     id: id piece
                  end;
  var
    table: array [0..table limit] of table entry;
    this id: array [1..8] of piece
             "80 character identifier";
```

The lexical analyzer scans an identifier by reading it character by character into a string variable, 'this id'. As each character is read, the ordinal value of the character is used to compute an index. Historically, this index is termed a hash key. The hash function computes the product of the ordinal values of the identifier characters modulo the table length. This hash key is then used as an index into the table of identifiers.

Different identifiers may have the same hash key. When a new identifier collides with one already in the table, a cyclical search is performed starting with the existing

entry. The search stops whenever the new identifier is found in the table or an empty table entry is encountered. If an empty table entry is reached, the identifier is given a new spelling index and inserted in the table.

New identifiers are inserted in the table as they are encountered in the program. Because collisions must be expected, the table must not be allowed to fill or searches will be long. The percentage of occupied entries is termed the table <u>loading</u>. A practical maximum loading depends on the application. The compiler uses a limit of 98%. Beyond this point a successful search would require more than twenty probes on the average. If insertion of a new identifier would exceed this loading, lexical analysis is terminated. Subsequent passes receive intermediate code up to the point of termination.

* number scan *

Numeric constants are scanned by this algorithm:

```
"ch is the current character"
while ch in digits do collect integer portion;
if ch = '.' then collect fractional portion;
if ch = 'e' then collect exponent portion;
construct numeric constant;
output intermediate code
```

The only difficulties in handling numeric constants are the avoidance of truncation errors and overflow. All numbers are handled by real arithmetic since real values have more significant digits than integer values on most machines. The integer portion of a number is collected as an integral real value. If no fractional portion or exponent portion is present, then the number is assumed to be an integer. If it is not greater than the largest allowable integer, it is truncated to an integer and output as the <u>int</u> <u>const</u> operator followed by the integer value.

If a fractional portion or an exponent portion exists, then the number is a real. The integer portion and the fractional portion are collected in the same manner:

```
number:= 0.0;
while ch in digits do
  if number < real limit then
    number:= number * 10.0 + (ord(ch) - ord('0'))
```

where <u>real</u> <u>limit</u> is the maximum real number divided by ten. It is important that the fractional portion be treated as above, and not be constructed by dividing successive digits by 10, 100, 1000, etc. since this would accumulate roundoff error. Rather, the fractional portion is treated as belonging to the integer part and the exponent is adjusted.

Following this the exponent portion, if any, is collected. Assuming the number is real, its representation must be constructed. First the exponent is checked to see if it is within range. If it is then it is constructed as a power of ten. Again it is important only positive powers of ten be constructed to avoid truncation error. If the exponent is a negative power of ten, it is divided into the number to produce the result; overflow is impossible. If the exponent is a positive power of ten, then multiplying it by the number could produce overflow, but:

```
if number = 0.0 then result:= 0.0
else
  "number >= 1 and
   number * power of ten <= maximum real
   => power of ten <= maximum real / number
      <= maximum real"
  if power of ten <= maximum real / number
    then number:= number * power of ten
    else error
```

The intermediate code is a file of integers. To place a real number in the intermediate code use is made of Concurrent Pascal's <u>universal</u> <u>type</u> facility. Universal types allow arguments of passive types [1] to be passed to procedures as long as they occupy the same number of machine words as the procedure's corresponding parameter. In our

implementation a real value occupies four integer locations. So the following suffices to output a real constant:

```
type  split real = array [1..4] of integer;
procedure put real (argument: univ split real);
var i: 1..4;
begin
  for i:= 1 to 4 do put(argument[i])
end;
   .
   .
   .
put real(number);
```

* efficiency *

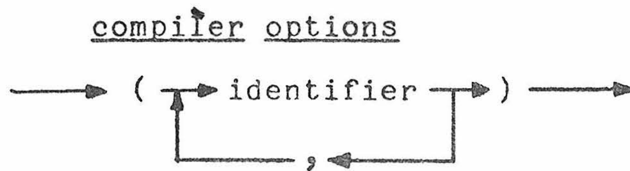
A pass's work load varies with its input. The input to Pass 1 is measured in characters, while the input to later passes is measured in integers. Lexical analysis processes from 70% to 500% more input symbols than any other pass (see Chapter 13). This large amount of input combined with the slowness of character I/O makes Pass 1 a bottleneck. In Chapter 13's example lexical analysis consumes about 37% of the elapsed time for compilation. A little attention paid to optimization here is worthwhile.

Character scanning must be as fast as possible. The source program used as an example is 1280 lines. For a standard 80-column card this is over 100,000 characters. Fortunately in our operating system the card reader routine (not a part of the compiler) truncates trailing blanks from cards. This results in an average line length of only 20 characters, or a reduction to 25,000 characters. So every 10 microseconds saved in a character scan saves 1/4 second in elapsed time. Lexical analysis scans a character by calling the operating system once to read the next character and once to write it. These two calls are placed inline wherever needed. The compiler always produces a listing file of the source text. The user can then tell the operating system whether or not to print the listing. This avoids the overhead of a listing option within the compiler.

Trailing blanks from lines are suppressed before they reach Pass 1. The only other place a string of blanks might often be found is at the beginning of a line. So at the end of every line (signaled by an NL character) blanks at the beginning of the next line are skipped. Within statements, blanks usually appear singly, so looping to skip blanks is not worthwhile.

* compiler options *

Pass 1 must scan and interpret compiler options. This requires a simple syntax:



Compiler options must precede the program. They are scanned by Pass 1 immediately after pass initialization, before entering the main scan loop. Only the first character of the option identifier is recognized. Currently three options are implemented: number indicates the generated code will only identify line numbers at the start of routines; check indicates the generated code will not make range checks of constant enumeration arguments; test will print the intermediate output of all passes, including Pass 1.

Compiler options must be communicated to later passes. Pass communication is governed by an interpass record that remains in the heap during compilation. Essential information that must precede the intermediate code is placed in the interpass record. It is defined as:

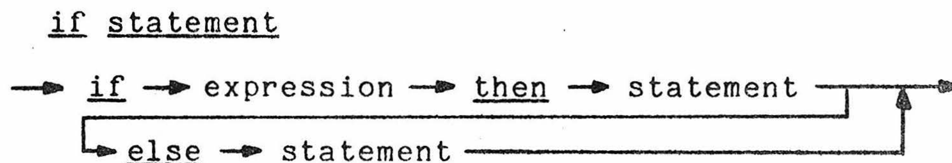
```
type
  interpass record =
    record
      options:  set of option;
      "other information"
      table: @pass dependent table
    end;
```

Pass 1 allocates the interpass record on the heap. At the end of each pass, the pass link (a pointer to the interpass record) is passed as an argument to the next pass.

5. Syntax Analysis

* function *

Syntax analysis checks ("parses") the program syntax. It consists of a set of recursive procedures that gradually examines the syntax in more and more detail. A recursive descent parser contains a possibly recursive procedure for each syntactic construct, represented by a syntax graph. For example, the if statement construct is:



If we avoid the problem of error recovery, a procedure to parse the above might be:

```
procedure get "next symbol";
begin
  "read next symbol into variable 'sy'"
end;
procedure if statement;
begin
  get "past if symbol";
  "boolean" expression;
  if sy = then then get else error;
```

```
    "then" statement;
    if sy = else then begin
      get "past else symbol";
      "else" statement
    end
  end;
```

When the parser is inside its *if* statement procedure, the sequence of previous procedure calls might be:

program
declarations
body
statement
if statement

and we can see that the statement procedure will now be called recursively to parse the *then* statement of the *if* statement. This nesting can become quite deep, reaching to thirty levels for even simple programs.

* error recovery *

Each parsing procedure is a simple sequence of statements that follow the syntax graphs. The parser can be written directly from the syntax graph. Error recovery is also dictated by the syntax graphs. Error recovery is done to detect more errors during a single compilation and to

prevent a cascade of error messages caused by a single error. Systematic syntactic error recovery is an original contribution of this thesis.

To develop the error recovery scheme, consider the input to the parser. The first intermediate code consists of operators possibly followed by arguments. Syntax analysis ignores all operator arguments, since these are concerned with semantics. There are 66 distinct operators in the first intermediate language. Using Pascal's _set types_, it is possible to create sets of operators. The operators that may begin a particular syntactic construct are its _handles_. The handle of an _id list_ is the set [id]. The set of statement handles is [id, begin, if, case, while, repeat, for, cycle, with, init].

Whenever a syntax error is detected, zero or more input symbols are skipped until a _key_ symbol is obtained. A key symbol is any symbol from which compilation may resume. A set of key symbols, called _keys_, is passed to an error routine along with an error number:

```
type symbols = set of symbol;
procedure error (number: integer; keys: symbols);
begin
  give error indication;
  while not (sy in keys) do get "next symbol"
end;
```

This basic idea was used in the original transportable Pascal compiler produced by Wirth's group. Its unsystematic application there flawed that compiler's reliability. To apply the method systematically, the key sets are derived directly from the syntax graphs. If an error occurs at a given point in a syntax graph, compilation may resume downstream from the given point. The keys contain any operators that can be reached in the current graph. They also contain the handles of any other graphs that may be reached. The process is so systematic that recursive descent parsers with error recovery might be generated automatically from the language definition itself. Examples will follow.

This scheme implies that every parsing procedure accepts as input the keys of its caller. This permits each parsing procedure to ignore the context in which it is called. Local keys are added to the initial ones whenever the given procedure calls another parsing procedure. So in

general the set of keys increases as parsing procedures are called, and decreases as these calls are completed. The keys contain key symbols from each active level of the syntactic hierarchy. So when an error is detected, a minimum of input symbols will be skipped. The first rule of error recovery is:

Error Recovery Rule 1:

> The keys contain all symbols from which compilation may resume.

This rule is not enough to completely determine the parser's error recovery. One more rule is required to indicate where error checking is to be performed. Of course if a particular symbol is expected, then its absence is an error. But if one parsing procedure calls another, who should check for an error, the caller or the called? Should a parsing procedure assume when it is called that the current symbol is a key symbol? Or should it ensure that when it returns to the caller the current symbol is a key symbol? Or should these decisions be made for each single parsing procedure?

The solution, it turns out, is quite simple. If only a single symbol is expected, as the <u>then</u> symbol after the boolean expression of an <u>if</u> statement, then its absence is an error. Otherwise we must presume several different

symbols are expected, as the statement procedure expects any statement handle. When this occurs, a decision must be made. This is the case whenever a branch appears in the syntax graphs. So the second rule is:

Error Recovery Rule 2:

> Whenever a branch is encountered in the syntax graphs, check that the current symbol is a key symbol.

To implement this check, a procedure exists:

```
procedure check (number: integer; keys: symbols);
begin
  if not (sy in keys)
    then error(number, keys)
end;
```

and this procedure is called at every branch point in the syntax graphs.

To summarize then, only two rules exist. The keys contain every symbol from which it is possible to resume compilation. A check is made before each decision. These rules may appear so obvious as to not be worth mentioning. But together with the syntax graphs they completely determine the error recovery scheme! A language designer has only to design his language; the syntax analysis and error recovery is then purely automatic.

* syntax design considerations *

But in order to work effectively, the language designer must obey two simple rules.

Syntax Design Rule 1:

Symbols must be used unambiguously.

A symbol is used ambiguously when it occurs in two different constructs, and, worse, these constructs may be nested. If the inner occurrence of this symbol is missing it is possible for the outer occurrence of the symbol to be associated with the inner construct. When these are different constructs the result is disastrous. Pascal itself is a gross violator of this rule. For example the begin - end keywords may delimit a compound statement, a procedure, or a program, and each of these may be nested. If the end of a compound statement is missing, then the end of the procedure is taken as the end of the compound statement. The end of the program is taken as the end of the procedure, and the body of the program is then assumed to be missing.

The error message will indicate an improperly terminated program, when actually the compound statement is improperly terminated. On the other hand, if an extra end appears then it will terminate the compound statement. The end of the compound statement will terminate the

procedure. Then when no begin appears, the program body will be in error. This could be avoided with a properly chosen syntax. In Concurrent Pascal procedures may not be nested, which would detect some errors of this sort earlier. Much better is to avoid these ambiguities entirely when designing a language.

A corollary to the above rule can be incorporated as a second rule in its own right. That is:

Syntax Design Rule 2:

> All major syntactic constructs should be uniquely delimited.

Ideally every construct would be delimited by a unique set of symbols. This would supply ample redundancy to detect errors as soon as possible, and prevent as much as possible the mismatching of symbols when an error is encountered. It would also eliminate the compound statement whose overnesting creates problems even for humans. This rule is a point in favor of such eyesores as if - fi and case - esac, and a point against the semicolon as a statement separator.

Concurrent Pascal violates both rules of syntax design to be compatible with sequential Pascal. Nevertheless the error recovery scheme is quite robust and still does a fair job. For a well-designed syntax it can

do a superb job.

* three general cases *

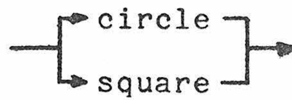
The method can be illustrated on three abstract graphs. Any syntax graph is comprised of a combination of <u>sequencing</u>, <u>branching</u>, and <u>looping</u>. These constructs are given below along with their associated parsing procedures. We use two abstract constructs, a circle and a square, and one abstract operator, a <u>spiral</u>.

1. Sequence

→ circle → <u>spiral</u> → square →

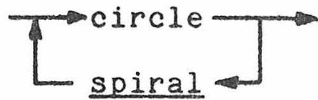
```
procedure sequence (keys: symbols);
begin
  circle(keys or [spiral] or square handles);
  if sy = spiral then get
    else error(sequence error, keys or
    square handles);
  square(keys)
end;
```

2. Branch




```
procedure branch (keys: symbols);
begin
  check(branch error, keys or circle handles
    or square handles);
  if sy in circle handles then circle(keys)
  else if sy in square handles
    then square(keys)
  else error(branch error, keys)
end;
```

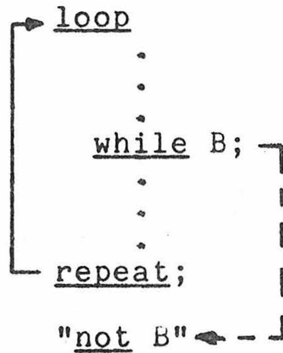
3. Loop




```
procedure loop (keys: symbols);
var  loop keys, all keys: symbols;
     done: boolean;
begin
  loop keys:= circle handles or [spiral];
  all keys:= keys or loop keys;
  done:= false;
  repeat
    circle(loop keys);
    check(loop error, all keys);
    if sy in loop keys then
      if sy = spiral then get
      else error(loop error, all keys)
    else done:= true
  until done
end;
```

The _loop_ procedure may appear complicated. However it merely follows the rules already outlined. The test for termination of the loop involves an auxiliary boolean variable since actually the loop terminates in the middle. If Concurrent Pascal possessed a _loop_ statement similar to

that proposed by Dahl and advocated by Knuth [4], namely



then the loop would become:

```
loop
  circle(loop keys);
  check(loop error, all keys);
  while sy in loop keys;
  if sy = spiral then get
    else error(loop error, all keys)
repeat
```

The structure is much clearer in this version. If a spiral is forgotten between two circles, compilation gives an error message and resumes as though the spiral had been present. This conforms to Rule 1. After a circle there is a check made before deciding which branch of the syntax graph to take. This conforms to Rule 2. Note also that the test for termination involves a test against the loop keys. Assuming the hypothetical loop construct may be nested, it would be incorrect to test for termination by saying:

```
while not (sy in keys);
```

but it would be correct (though unclear and inefficient) to say

```
while not (sy in keys - loop keys);
```

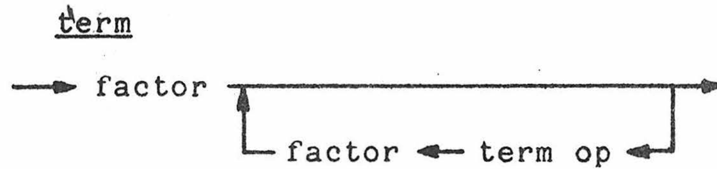
Before programming a parser in this scheme, one must master the three basic constructs. Then more complicated constructs only require strict adherance to the rules. As an example, the *if* statement combines the sequence and branch:

```
procedure if statement (keys: symbols);
begin
  get "past if symbol";
  "boolean" expression(keys or
    statement handles or [then, else]);
  if sy = then then get
    else error(if error, keys or
      statement handles or [else]);
  "then" statement(keys or
    statement handles or [else]);
  check(if error, keys or
    statement handles or [else]);
  if sy = else then begin
    get "past else symbol";
    "else" statement(keys)
  end
end;
```

This example can be simplified by taking advantage of context. A valid assertion for this procedure is 'statement handles <= keys'. Whenever the 'if statement' procedure is called, the keys already contain the statement handles.

As another example, a term combines the sequence and the loop:

term




```
procedure term (keys: symbols);
var  term keys, all keys: symbols;
begin
  term keys:= factor handles or term operators;
  all keys:= keys or term keys;
  factor(all keys);
  loop
    check(term error, all keys);
    while sy in term keys;
    if sy in term operators then get
      else error(term error, all keys);
    factor(all keys)
  repeat
end;
```

* the output *

The discussion has sofar described the parsing technique and the error recovery scheme. To complete the description of syntax analysis, the generation of the second intermediate code must be explained. The second intermediate code is a syntactically correct (but possibly

meaningless) program in postfix notation. The _if_ statement:

_if_ B _then_ S1 _else_ S2

in postfix notation becomes:

B _if_ S1 _then_ S2 _else_.

In postfix notation each operator is preceded by its operands. The _if_ operator takes the boolean expression as its operand. If B is false a jump is made to statement S2. The _then_ operator causes a branch around statement S2, and it indicates the start of S2. The _else_ operator indicates the end of the _if_ statement. In terms of the intermediate code this becomes:

B _falsejump_(L1) S1 _jump_(L2) L1: S2 L2:

If no _else_ clause were present, the second intermediate code would be:

B _falsejump_(L1) S1 L1:

Syntax analysis, like the other passes, uses several standard output routines. Procedure _put_ appends an operator to the output intermediate code file. Procedure _put1_ appends an operator and an argument to the output intermediate code file. Similarly for procedure _put2_, but with two arguments. We can now extend the _if_ statement procedure to its full form:

```
type  label = integer;
var  current label: label; "initially zero"
procedure new label (var l: label);
begin
  current label:= succ(current label);
  l:= current label
end;
   .
   .
   .
```

```
procedure if statement (keys: symbols);
var  l1, l2: label;
begin
  get "past if symbol";
  "boolean" expression(keys or
     [then, else]);
  new label(l1);
  put1(false jump, l1);
  if sy = then then get
    else error(if error, keys or [else]);
  "then" statement(keys or [else]);
  if sy = else then begin
    get "past else symbol";
    new label(l2);
    put1(jump, l2);
    put1(label, l1);
    "else" statement(keys);
    put1(label, l2)
  end else put1(label, l1)
end;
```

This completes the description of syntax analysis.

6. Name Analysis

* function *

Name analysis converts spelling indices to name indices and enforces Concurrent Pascal's scope rules. Lexical analysis has already converted all unique identifiers into unique spelling indices. Concurrent Pascal allows the same identifier to name different constants, types, variables, or routines in different blocks. Name analysis converts these possibly ambiguous spelling indices into unique name indices. A name index refers to a single constant, type, variable, or routine throughout its lifespan.

Name analysis also enforces the scope rules. The _scope rules_ define the rules for recognition of identifiers. To be recognized, an identifier must first be _known_. Identifiers are known after they have been _introduced_. An introduction is either a _declaration_ or a _qualification_. Declaration associates an identifier with a particular constant, type, variable, or routine. Qualification associates _field_ or _entry_ identifiers with a particular record variable or system component. A qualification may be either the variable name followed by a period, or it may be a _with_ statement. The scope rules are:

1. An identifier is only known with a given meaning after its introduction (with that meaning) and until the completion of the block, record, or qualification that introduced that identifier (with that meaning).

2. No identifier may be given more than one meaning in a single block or record.

3. An identifier may be introduced with another meaning in another block, record, or qualification. Where this occurs, the new meaning applies until the completion of the block, record, or qualification.

4. Within a system component are known:
   a) all identifiers introduced in the system component type (except for entry routine identifiers);
   b) all constant and type identifiers declared in enclosing system component types.

5. Within a routine is known, in addition to the above, all identifiers introduced in the routine.

The fourth rule is a departure from pure block structure scope rules. It forbids a nested component definition from referencing the parameters or variables of an enclosing component. This rule gives the operating system designer explicit control over the access rights of components.

* the tables *

Name analysis implements these rules through several tables:

a) the <u>spelling table</u> translates a spelling index to a unique name;

b) the <u>update stack</u> contains old spelling table entries that have been temporarily replaced;

c) the <u>display</u> marks the update stack for each level and contains other information associated with levels.

Around these data structures revolves the entire structure of the pass.

* the spelling table *

The spelling table contains an entry for every possible spelling index. Associated with each spelling index used is its name. To enforce the scope rules, an access attribute and a nesting level are also associated with the index. This structure appears as:

```
type
  spelling index = 0..spelling max;
  access attribute = (general, external, internal,
                      incomplete, unresolved,
                      qualified, functional,
                      undefined);
  level index = 0..level max;
  name pointer = @ name entry;
  spelling entry = record
                     name: name pointer;
                     access: access attribute;
                     level: level index
                   end;
var
  spelling table: array [spelling index]
                    of spelling entry;
```

The name is a pointer to an entry in a table containing all information associated with the name in this pass. We postpone this discussion til later.

The access attribute and level index determine the program's access rights to the name as defined by the scope rules. This gives the operating system designer selective control over access to operating system components. Names with general access may be referenced in the block in which they are defined and in any nested blocks. Constant and type names have general access.

Names with external access may only be referenced outside the block in which they are declared. A system component may not reference its own entry routines, and so they have external access.

Names with internal access may only be referenced in the system component or routine in which they are defined. Unlike general access, these names may not be referenced in nested system component types. This distinction between general and internal access involves a comparison of the name's level with the current component type's level. No level comparison is required with general access. Variable, parameter, and non-entry routine names have internal access. A system component's variables and parameters may be accessed in the component and its routines, but not in

nested system component types. A routine's variables and parameters may only be referenced inside the routine. Routines may not be nested.

Names with _incomplete access_ may not be referenced. Type and procedure names have incomplete access until the completion of their declaration. A type declaration may not reference itself; a procedure may not be recursive.

Names with _unresolved access_ may only be referenced in the interface list of a sequential program declaration. A name may be introduced in such a list. When this happens its access changes from undefined to unresolved. After the entry routine is resolved, its access becomes external.

Names with _qualified access_ are introduced by _with_ statements. A _with_ statement selects a record variable or system component for processing. This introduces the field or entry names, and they are included in the spelling table with qualified access.

The name of a function in the body of a function has _functional access_. This means a value may be assigned to the function result, but the function may not be referenced recursively.

Undefined names have <u>undefined access</u>. They may not be referenced before being declared. This is the last of the access attributes.

* the update stack *

Updating of the spelling table is accomplished via an update stack, a technique due to Naur [5]. Whenever a name is introduced its previous spelling entry value is pushed on the update stack. At the end of the scope (block, record declaration, or qualification) that introduced the name, the old spelling entry is popped from the update stack and put back into the spelling table. This requires that the "base" of the current portion of the update stack be marked at the beginning of new scopes (also called levels). Analagous to the storing of base addresses in a run-time <u>display</u>, the base indices of the update stack are stored in a compile-time display. These two structures may be described as:

```
type
  update index = 0..update max;
  update entry = record
                   location: spelling index;
                   old entry: spelling entry;
                 end;
  display index = 0..display max;
```

```
display entry = record
                  level entry: name pointer;
                  base: update index;
                  previous component level:
                    level index;
                  previous qualification list:
                    qualification pointer
                end;
var
  display: array [display index] of display entry;
  update stack: array [update index]
                of update entry;
  current level,
  current update: update index;
  current component level: display index;
  current qualification list: qualification pointer;
```

The display contains all information relevant to the nesting of levels. When a new level is entered in either a declaration or with statement, a new entry is pushed on the display. This new entry contains a name pointer to the system component type, routine, or with temporary associated with the level. The base of the update entries for this level is marked. The previous system component level is remembered in case this is a nested system component type. The previous qualification list is also saved. Entry names

or field names associated with a system component or record type are maintained in a qualification list. A list is associated with each level since these types may be nested. Qualification lists will be discussed in more detail later.

Entering and leaving levels of nesting is controlled by the sequence of declarations and with statements. The semantic routines associated with these constructs may use two routines that push and pop display entries to enter and exit levels:

```
procedure push level (level name: name pointer);
begin
  if current level = level max
    then abort compilation
    else current level:= succ(current level);
  with display[current level] do begin
    base:= succ(current update);
    level entry:= level name;
    previous component level:=
      current component level;
    previous qualification list:=
      current qualification list
  end;
  current qualification list:= nil
end;
```

```
procedure pop level;
var this update: update index;
begin
  with display[current level] do begin
    current component level:=
      previous component level;
    current qualification list:=
      previous qualification list;
    for this update:= current update
      downto base do pop update
  end;
  current level:= pred(current level)
end;
```

The pushing and popping of update entries is controlled similarly:

```
procedure push update (this index: spelling index;
  this name: name pointer;
  this access: access attribute);
begin
  if current level > global level then begin
    "save the old entry"
    if current update = update max
      then abort compilation
      else current update:=
             succ(current update);
    with update stack[current update] do begin
      location:= this index;
      old entry:= spelling table[this index]
    end
  end;
  "now fill in the new entry"
  with spelling table[this index] do begin
    name:= this name;
    access:= this access;
    level:= current level
  end
end;
```

```
procedure pop update;
begin
  with update stack[current update] do
    spelling table[location]:= old entry
end;
```

* table size *

Overflow in any of these tables will abort compilation. The pass will terminate and subsequent passes will process intermediate code only up to the point of termination. For this reason the tables must be large enough to accomodate as many names as may be used in the largest program that may run on the machine. The size of the spelling table is determined by the size of the hash table used in lexical analysis. The display is small, as few programs are very deeply nested. Concurrent Pascal does not allow routines to be nested. A few levels of nesting for system component types, record types, and with statements is all that is required.

The update stack can be small since names in the outermost scope (global names) need not be entered. Languages without name qualification, such as Algol 60, only place names in the update stack when they are <u>re</u>defined. This makes level popping less efficient since local names

must be removed from the spelling table by a search for current level numbers. This increase in the cost of level exits is tolerable only when level crossings correspond to block boundaries. In languages with name qualification, level boundaries may be crossed many times within a block. A search of the entire spelling table to "undefine" newly defined entries would be intolerable. For this reason every nonglobal name has its old spelling entry placed in the update stack.

Another performance consideration involves the use of qualification lists. These lists contain the entry names of system component types or the field names of record types. When a system component or record variable name is followed by a period, a new level is entered. Any of that variable's entry or field names is now included in the scope. Since only one entry or field may be selected following the period, it is not worthwhile to update the spelling table with all the possible fields or entries. Instead a linear search of the qualification list is made to retrieve the name of the particular field or entry selected. The spelling table remains unaffected.

The situation is different when a system component or record variable is named in a with statement. Here there may be many selections from the variable. In this case the

spelling table is updated to reflect the change in scope. The qualification list is traversed and each field or entry name is placed in the spelling table. At the conclusion of the with statement the new level is popped.

* the name table *

Another significant data structure of name analysis is the name table. Once a name is recognized through the spelling table, a pointer to the name entry is obtained. The name table contains all information associated with a name whether it be the name of a constant, type, variable, parameter, or routine. Concurrent Pascal does not require that every constant or type possess a name. Name analysis responds to this in two different ways.

Constants are nameless. No name index is assigned to constants. Name analysis removes constant declarations from the intermediate code. Index constants are represented in the name table by their value. All other constants (real or string) are represented in the name table by their displacement in the program's constant area. Wherever constant names appear in the intermediate code, they are replaced by their value or displacement.

Types, on the other hand, are all given a name index, whether or not the programmer names them. This is done so that declaration analysis, the next pass, may refer to types by their names (name indices).

Associated with each name in the name table is only the information required by name analysis. So far we have described three functions of name analysis. It assigns name indices to types, variables, parameters, and routines; it replaces constants by their values or displacements; it enforces the scope rules of Concurrent Pascal. This last function, scope rule enforcement, really means that name analysis controls the access rights of the program. _What_ can or cannot be accessed is determined by this pass. Later passes will determine _how_ these names may be accessed. This forms a clean division between these logically separate aspects of semantic analysis.

Access to a name involves referencing the name table. The name table is a linked list structure that represents the access relationships of types, variables, parameters, and routines. Subrange types are linked to their range types. System component types are linked to their entry routines. Routines are linked to their parameters; functions are also linked to their result types and sequential programs to their interface. Array types are

linked to their index and element types. With statement temporaries are linked to their record or system component types. Record types are linked to their fields. All these relationships are represented in the name table, and this information is distributed where necessary in the intermediate output code. No subsequent pass possesses a linked structure that reproduces these relationships.

A name table entry is defined as:

```
type
  qualification pointer = @ qualification entry;
  qualification entry = record
                          spelling: spelling index;
                          name: name pointer;
                          next qualification:
                            qualification pointer
                        end;
  name index = 0..name max;
  name pointer = @name entry;
  name entry =
    record
      index: name index;
      case kind: name kind of
        index constant: (
          constant type: name index;
          constant value: integer);
```

```
real constant: (
  real displacement: integer);
string constant: (
  string length,
  string displacement: integer);
variable: (
  variable type: name pointer);
parameter: (
  parameter type,
  next parameter: name pointer);
field: (
  field type: name pointer);
scalar type: (
  range type: name index);
component type: (
  initial statement: name pointer;
  entry list: qualification pointer);
routine: (
  parameter list: name pointer;
  function type: name index);
sequential program: (
  sequential parameter list:
    name pointer;
  interface list:
    qualification pointer);
```

```
        array type: (
          index type: name index;
          element type: name pointer);
        with temporary: (
          with type: name index);
        record type: (
          field list: qualification pointer)
      end;
```

* the operand stack *

Name analysis stores operands in a stack since they precede their operator in the input code. An operand entry is similar to a name entry, but there are some differences. After defining an operand entry we will discuss its use.

```
type
  operand index = 0..operand max;
  operand entry =
    record
      case class: operand class of
        index constant: (
          constant type: name index;
          constant value: integer);
```

```
    real constant: (
      real displacement: integer);
    string constant: (
      string length,
      string displacement: integer);
    variable: (
      variable type: name pointer);
    routine: (
      routine entry,
      next parameter: name pointer);
    function result: (
      function type: name index);
    case label: (
      label number,
      case value: integer);
    declaration: (
      declaration entry: name pointer;
      declaration index: spelling index)
    "undefined, factor constant: (
      empty)"
end;
```

Constants encountered in a declaration are pushed on the operand stack. Whether or not the constant value is placed in the intermediate output code depends on the particular construct. Constants appearing in constant

definitions are placed in the name table and not transmitted until they are referenced. Constants in the body appear either as labels or as factors. Constant labels are pushed on the operand stack as case labels. Constant factors are immediately transmitted in the intermediate code and an entry pushed on the operand stack. This entry is empty, though, since the factor value is not required in this pass.

Variables may only be referenced in a body. When a reference appears, the variable type is pushed on the operand stack. Variables may be either "subscripted" or "qualified". A subscript applied to an array variable replaces the name of the array type with the name of the array element type. A period and a field name applied to a record variable replaces the record type with the field type. A period and an entry routine name applied to a system component replaces the variable operand entry with a routine operand entry.

Routines may be referenced in the body or in the interface list of a sequential program declaration. Routine names appearing in an interface list are not placed on the operand stack. Instead they are added to a chain of names associated with the program declaration. This chain is maintained by the same mechanisms used to maintain qualification lists. Routines referenced in the body are

placed on the operand stack. The name of the routine and the name of its first parameter are included in the operand entry. As each argument appears in the input code, the parameter chain is followed to the next parameter. If the parameter chain is shorter than the argument list, an error indicating too many arguments is given. If the argument list is shorter than the parameter chain, an error indicating too few arguments is given.

Routines may not be referenced recursively. The name of a function may be referenced in the function body only to assign a <u>result</u> to the function. For this reason a special access attribute, functional access, is given to the function name inside the function body. Reference to a function name with this attribute places the function result entry on the operand stack.

Names are declared in a declaration part. While the declaration is still incomplete, the operand stack entry indicates a <u>declaration</u>. Associated with the declaration is its spelling index and a pointer to its incomplete name entry. This information is used to update the various tables at the completion of the declaration.

Occurrence of an error in the declaration part or body part may invalidate an operand. As in the Gier Algol compiler [5], no attempt is made to correct an invalid

operand. Its description is changed to <u>undefined</u>. Subsequent accesses to an undefined operand are ignored by the pass, but undefined operands will be placed in the intermediate output code where necessary. No final code is produced for an incorrect program. Undefined operands may result from many different errors. For example an attempt to ambiguously define a name will yield an undefined operand. An attempt to attach an argument list to anything but a routine will yield an undefined operand. Error recovery consists of marking the operand undefined and ignoring further attempts to process the operand. For this reason, every operand access must first check for an undefined operand. This involves far less effort than to correct illegal operands.

* summary *

This pass's output contains unique name indices that are used in later passes to refer to types, variables, parameters, and routines. All access linkages between these quantities are checked and distributed in the output code. The name table is used to represent the structural relationships of language elements. This structural relationship embodies the major complexities of the language. Name analysis isolates this complexity from the balance of semantic analysis. With few exceptions, the

nodes of this structure contain only name indices and links to other nodes. These links are distributed in the intermediate code by transmitting the name index of the node referenced by the link. As examples, a variable appears in the output as the variable's name index followed by its type's name index. A subscript expression is followed by the array index type's name index and the array element type's name index. In this way traversal of linked structures is avoided in later passes. Name analysis is concerned only with names and their relationships. The passes next described deal with what these names represent.

7. Declaration Analysis

* function *

Declaration analysis performs the semantic processing of declaration parts. It analyzes types, assigns addresses to variables and parameters, assigns program labels to routines, and distributes this information in the body parts. A host of semantic rules contained in the original language specification are enforced. These rules have two intentions: to enforce implementation restrictions, and to ensure proper use of language facilities. Examples of implementation restrictions are:

a) case labels must lie in the interval [0, 127];

b) string types must contain an even number of characters;

c) process components must be component variables of the initial process.

Examples of proper usage rules are:

a) universal types must be passive;

b) function parameters must be constant parameters;

c) queue variables must be monitor component variables.

There are more than a score of these rules. Their enforcement depends on the efficient representation of type information in the pass's data structures.

* the symbol table *

The analysis of declarations requires a symbol table. This pass's symbol table contains no pointers. All symbol table links have been analyzed and distributed by name analysis, the previous pass. These links appear in the input as name indices. They are translated to symbol table links through a name table:

```
type
  name index = 0..name max;
  symbol table link = @ symbol table entry;
var
  name table = array [name index]
                    of symbol table link;
```

When a name is declared, an entry is created for it in the symbol table. The link to the entry is then stored in the name table. Subsequent references to the name are processed indirectly through the name table. Note that the link to the entry instead of the entry itself is housed in the name table. This permits symbol table entries to be allocated dynamically as they are declared. In this way small programs may be compiled in less memory space than large programs.

The operand stack used in this pass is a simple vector of symbol table links. Operands appear in the input as name indices. They are translated to links via the name table, and the links are pushed on the operand stack.

This same simplicity carries over to the symbol table. In contrast to the plethora of symbol table variants used in name analysis, there now exist only three non-empty variants. A fourth variant, the undefined entry, is empty. Variables and parameters are combined in the first variant, routines in the second variant, and types in the third variant:

```
type
  symbol table entry =
    record
      case class: entry class of
        value: (
          "variable or parameter information");
        routine: (
          "routine information");
        template: (
          "type information")
    end;
```

* the value variant *

Variables and parameters are represented in the symbol table by a <u>value</u> variant. This variant contains the following information about the value:

a) the address mode

b) the address displacement

c) the declaration context

This information is required by later passes and will be distributed in the output.

The <u>address mode</u> and <u>address displacement</u> are a virtual address in Concurrent Pascal. Classical block structured architectures utilize an address consisting of an address level and address displacement. In Concurrent Pascal routines may not be nested inside other routines, so there exist only two levels, the system component level ("global") and the routine level ("local"). The mode encodes this information, as well as the type of system component or entry routine. Some of the modes represented in this pass are temporary modes; they do not appear in the final code. The modes are:

small constant *)

large constant

simple routine

sequential program

process entry routine

class entry routine

monitor entry routine

process component

class component

monitor component

standard routine *)

undefined *)

*) temporary mode

The address displacement is the displacement of the value within the data record of the component, routine, record, or constants area. Displacements are assigned sequentially as field, variable, and parameter declarations are processed. Displacements may be positive or negative, they may be assigned forwards or backwards, and they may or may not be offset. Record fields have positive forward displacements without offset. For example, in a record with two integer fields, the first field's displacement is zero and the second field's displacement is one word. Variables have negative forward displacement without offset. For example in a routine with two integer variables, the first variable's displacement is minus one word and the second variable's displacement is minus two words. Parameters have positive backwards displacement with an offset of one word. Backwards means their displacements are assigned in order

from last declaration to first declaration. For example in a routine with two integer parameters, the last parameter's displacement is one word and the first parameter's displacement is two words. Function results are displaced similarly to parameters, but the offset is either one or two words depending on the mode. This assignment of displacements may appear a bit intricate (as it did to this writer) but it is largely determined by the address structure of the PDP-11/45. Chapter 12 illustrates these displacements.

Displacements are relative to a particular system component, routine, or record. The previous displacement must be saved whenever a new level is entered. Again, as in name analysis, this entails the use of a compile time <u>display</u>. The display is a stack that has an entry for each level. It contains the previous mode and displacement to be restored upon reentering the level.

The <u>declaration</u> <u>context</u> of a value indicates the context in which the value was declared. This information is used in the next pass (body analysis) primarily to determine if a value may be changed. The different contexts are:

function result

class entry variable

variable

variable parameter

universal variable parameter

constant parameter

universal constant parameter

generic standard function parameter

record field

constant

expression

The "generic standard function parameter" context is used to handle the tricky standard functions, absolute value, successor, and predecessor, whose result types depend on the argument types. This problem is discussed in the body analysis description. The "constant" and "expression" contexts are also used in that pass since no declarations appear for them.

* the routine variant *

Routines are represented in the symbol table by a routine variant. This entry contains the address of the routine and the routine's parameter length and local variable length. A routine may be a local routine of a system component, or it may be an entry routine. This information is encoded in the mode portion of the routine address. The modes are the same as those for values.

Unlike value addresses, though, a displacement is not given. Displacements in the program area will not be known until the code is assembled. In lieu of a displacement, a routine label is given as the second part of the address. These labels are then resolved into program displacements during code assembly.

The parameter length and local variable length are accumulated when the routine is declared. This information will be included in the final code. The parameter length is required in order to pop the parameters from the data stack upon routine exit. The variable length is required in order to push the variable storage area on the data stack during routine entry. The initial statement of a system component is treated, for these purposes, as an entry routine. Associated with it are the parameter length and component variable length of the system component itself. In the case of a process initial statement, the additional stack length, if any, is included in the routine variant. This facility allows the programmer to allocate a fixed additional amount of storage to allow processes to execute sequential programs.

To summarize, then, a routine variant contains:

the routine mode

the routine label

the parameter length

the variable length

the additional stack length.

* the template variant *

All information associated with types is contained in the template variant. This information includes:

the name index

the type length

the active attributes

the type "kind"

information particular to individual kinds

The name index of the type is retained. It is transmitted in the intermediate code for use in type checking by body analysis. The length of the type is used for assigning displacements and may be incorporated in the final code.

* the active attributes *

The active attributes are a set of attributes associated with the type. They indicate whether the type contains an instance of an active type. This information is important since many semantic rules require knowledge of the active attributes of a type. If a type contains no active

types, then it is considered a passive type, and its active attributes are empty. Pascal's simple types, record types with passive fields, and array types with passive elements are all examples of passive types. A class type is an active type with the class attribute. A monitor type is an active type with the monitor attribute. A process type is an active type with the process attribute. A queue type is an active type with the queue attribute. Structured types (array or record types) containing active types are themselves active types. They inherit the attributes of their elements.

The active attributes are represented by a short set. In this implementation of Concurrent Pascal all sets are the same length; they all contain 128 possible elements. Short sets are readily obtained from these rather long sets by using Concurrent Pascal's universal type facility. This facility was discussed in the description of lexical analysis. In composing structured types, these attributes are inherited by taking the union of the element type's attributes with the structured type's attributes. The operation of set inclusion tests for the presence of a particular attribute.

One inportant example of the use of active attributes is this. Queue variables are intended to be monitor component variables. Process access to monitors is mutually exclusive. Since only one process at a time may actively execute a monitor, the monitor may place one process in a queue while it services another. Transfer of the queue variable outside the monitor, say to another monitor, would violate this intent. The only way the monitor could pass the queue variable out to another component would be in an argument list. The queue variable could be placed in an argument list in an <u>init</u> statement or in an entry routine reference. Since initial statements are viewed as entry routines we are left with this one case. Entry routine parameter types may not possess the queue attribute.

* the type kind *

Types are classified into <u>kinds</u>. These various kinds are chosen to facilitate type checking in the body. This type checking will be done by the next pass. The possible kinds are:

integer

real

boolean

character

enumeration

set

string

queue

system component

passive

active

generic

undefined

The standard index types, integer, real, boolean, and char, are each given their own kind. Any other index type is considered an enumeration kind. The standard queue type is a queue kind. The system component types are system component kinds. An array of characters is a string type. Any passive structured type that is not a string type is a passive kind. Any active structured type is an active kind.

The generic standard routine parameter types are of generic kind. The possible generic types are arithmetic, index, and passive. For example the absolute value function takes an arithmetic argument, the successor function takes an index argument, and the input/output procedure takes several passive arguments.

Particular information may be included in the entry for the different kinds. Integer, real, character, and enumeration kinds contain the minimum and maximum values of

their enumeration. System component kinds contain the mode of the component and its variable length. This length will be incorporated in the code as the displacement required to obtain the base address of the component data area. This permits system component data areas to be addressed similarly to routine data areas.

* the pass output *

The symbol table entries are distributed in the intermediate code by declaration analysis. A single entry may be distributed many times, since it is inserted in the output wherever the entry is referenced in the body portions of the program. Only two output formats are used for entries: a value format and a routine format. Type information is included in the value format.

The following information appears in the value format:

the address mode

the address displacement

the declaration context

the type kind

the type name index

the type length.

The value format is preceded by one of two intermediate language operators. These operators are <u>var</u> or <u>vcomp</u>. The

var operator implies an unqualified variable. The *vcomp* operator implies a qualified variable (a variable component). Constants are treated as unqualified variables with a constant declaration context. So in the output constants, variables, and parameters all appear as values.

The routine format contains:

the routine mode

the routine label

the parameter length

the variable length

the additional stack length.

The routine format is also preceded by one of two intermediate language operators: *routine* or *rcomp*. The *routine* operator implies a simple routine, while the *rcomp* operator implies an entry routine. Function references introduce an additional requirement for specification of the function type.

This scheme provides a uniformity of reference to either values or routines. Declarations are consumed and distributed in the body where required. This permits a very simple design for body analysis, the next pass to be described.

8. Body Analysis

* function *

Body analysis performs semantic checking in the body parts of the program. It checks the compatibility of operands and their operators, and generates addressing commands for the machine. This is the final phase of semantic processing. Name analysis has consumed constant declarations, and declaration analysis has consumed type, variable, and routine declarations. Devoid now of declarations, the intermediate input code consists of a simple sequence of bodies.

A short summmary of semantic analysis is: Name analysis checks the access relationships of the program and distributes valid symbol table links in the output code. Declaration analysis checks the declarations of the program and distributes valid symbol table nodes in the body. Body analysis then checks the compatibility of operands and their operators and distributes valid commands in the body. These commands will then be processed by the code assembly passes to produce the final machine code.

* type compatibility checking *

Type compatibility may be of two forms:

a) compatibility of operands with each other;

b) compatibility of operands with their operator.

For example the addition operator requires that its two operands be compatible with each other (a) and that they be arithmetic (b). Checking the compatibility of operands with each other follows the type compatibility rules of Concurrent Pascal. These rules have been especially chosen to minimize the labors of type checking and of learning the language as a programmer. Two types are compatible if any of the following are true:

1) they are defined by the same type definition;
2) both are subranges of a single type;
3) they are string types of the same length;
4) they are set types whose members are the same index type;
5) they are set types, one (or both) of which is the null set type;
6) one type is a universal parameter type and the other type is a passive argument type of the same length;
7) one type is an argument type and the other

type is its generic parameter type.

Type information appears in the input as it was distributed by declaration analysis. So types appear as three arguments: a kind, a name index, and a length. These arguments are chosen to mesh with the compatibility rules in a simple manner. This scheme is made possible by using a small set of primitive attributes to represent the context and type information of operands. This information is contained in the operand stack; _no symbol table exists in this pass._ The operand stack is a linked stack whose entries are:

```
type
  operand entry =
    record
      "address information"
      mode: address mode; displacement: integer;
      "type information"
      kind: type kind; name: name index;
      length: integer;
      case class: operand class of
        value: (
          "value information"
          context: declaration context;
          state: address state);
        routine: (
          "routine information"
          parameter length,
          variable length,
          additional stack length: integer)
        "undefined: (
          empty)"
    end;
```

The address information represents the virtual address of the operand. In the case of routines the 'displacement' is a label. The type information is the same as in declaration analysis. For routines, this would be the

function result type, if any. Except for the address state, to be discussed later, the routine and value information has also been described before.

Type compatibility is checked by a function that compares the type of the top operand, 't', and second to the top operand, 's':

```
function compatible: boolean;
begin
  if t@.context in universal then
    "apply Rule 6"
    compatible:= (s@.kind in passives)
      and (t@.length = s@.length)
  else if t@.kind = s@.kind then
    case t@.kind of
      integer kind, real kind, boolean kind,
      character kind, queue kind:
        "Rules 1, 2"
        compatible:= true;
      enumeration kind, passive kind,
      active kind, component kind:
        "Rules 1, 2"
        compatible:= t@.name = s@.name;
      string kind:
        "Rules 1, 3"
        compatible:= t@.length = s@.length;
```

```
          set kind:
            "Rules 1, 4, 5"
            compatible:= (t@.name = s@.name)
              or (t@.name = null)
              or (s@.name = null);
          undefined kind:
            compatible:= false
            "but suppress error message"
        end
      else if t@.kind = generic kind then
        "Rule 7"
        case t@.name of
          arithmetic genre:
            compatible:= s@.kind in arithmetic;
          index genre:
            compatible:= s@.kind in indexs;
          passive genre:
            compatible:= s@.kind in passives
        end
      else compatible:= false
    end;
```

This simple function performs compatibility checking of two operands. It is usually only invoked for argument type checking where the full range of operand types is possible. Since most operators take limited operator types, the check

can usually be performed even more simply in-line. For example, addition checking need only ask if two kinds are either both integer or both real.

The context of a value, as well as the kind, is also used in compatibility checking. Assignment targets and variable arguments must be assignable. This is checked by examining the context of the value.

* addressing commands *

Before an operand may be utilized by the machine, it must be addressed. Body analysis makes use of an _address state_. The address states are:

direct,

indirect,

addressed, or

expression.

The _direct_ state indicates an operand that is directly addressable. Its mode and displacement are known. Unqualified variables and constant parameters are directly addressable.

The _indirect_ state indicates an operand whose address is directly addressable, for example, a variable parameter.

The <u>addressed</u> state indicates an operand whose address is on the machine's stack (such as a subscripted variable), while the <u>expression</u> state indicates an operand whose value is on the machine's stack.

To utilize an operand, the machine requires its state be either addressed or expression. <u>Short</u> operands may be placed directly in the stack, while <u>long</u> operands may only have their addresses placed in the stack. The short operands are either of byte length (characters within strings), word length (enumeration types), real length (reals), or set length (sets). An address is itself a short (word length) operand. Long operands are of structured type (arrays or records).

If an operator (such as addition) requires a value on the stack, then the 'value' routine within body analysis generates the necessary machine commands. If the operand is long, then its address is pushed on the stack as will be described later. If the operand is short, then the action depends on the address state:

```
case t@.state of
  direct:
    "generate command to load value:
    push value (length code, mode, displacement)"
  indirect:
    "first generate command to load value address:
    push value (word length, mode, displacement)"
    "then generate command to indirectly load value:
    push indirect (length code)"
  addressed:
    "generate command to indirectly load value:
    push indirect (length code)"
  expression:
    "value is already loaded"
end
```

Long operands, assignment targets, and variable arguments have their address, not their value, pushed on the machine stack at runtime. Again, the commands generated by body analysis depend on the address state of the operand:

```
case t@.state of
  direct:
    "generate command to load address:
    push address (mode, displacement)"
  indirect:
    "generate command to load address value:
    push value (word length, mode, displacement)"
  addressed:
    "the address is already loaded"
  expression:
    error "expressions are not addressable"
end
```

These operations of value loading or address loading are performed for most operators. Variable references are a good example. Consider the variable reference

v[i + 1].f

and its resultant loading commands. First the address of 'v' is loaded. Then the value of the subscript expression is loaded. Then the <u>index</u> command performs the indexing leaving the address of 'v[i + 1]' on the stack. Assume this is a record. Then next a <u>field</u> instruction is issued,

taking as its argument the displacement of 'f' within the record. This adds the displacement to the address already on the stack, leaving as a result the address of the field. If this is all that is required then no further commands are issued, else the value is loaded by a <u>push indirect</u> command.

Should 'v[i + 1]' be a system component and 'f' an entry routine, then the commands are different. After the indexing command is generated, a <u>field</u> command is generated with its argument the component offset. Then the routine reference is generated. A routine reference command is of the form.

<u>call</u> (mode, label, parameter length).

* error recovery *

As in previous passes, the error recovery scheme, due to Naur [5], consists of marking an operand undefined. This provides a simple uniform scheme of error recovery over the three semantic analysis passes. The error routines themselves change the operand description to undefined. Two error routines exist. The first routine is for unary operators, the second for binary operators. Each routine checks the operand descriptions on the stack. If they are undefined, no error message is given. If they are defined, then an error message is given and their description is

changed to undefined. This eliminates redundant error messages in this pass.

9. Code Selection

* function *

The compiler's last two passes, both designed and written by Per Brinch Hansen, perform code assembly using a classic two-pass design. The first assembly pass is named code selection. Its function is to define the addresses of program labels, determine the stack requirements of routines and components, construct the constants table, and translate the input code to final code. Code selection will leave four tables behind in the heap for use by the next pass, code assembly. These tables contain the addresses of routine labels, the addresses of jump labels, the stack requirements of components and routines, and the large constants. No arbitrary limit is placed on the size of the program that may be assembled.

* table management *

Code selection constructs four tables in the heap, a routine label table, a jump table, a stack table, and a constants table. Common management routines are used for each of these. Tables are broken up into pieces of 100 entries each:

```
const
  piece length = 100;
type
  piece pointer = @ table piece;
  table piece = record
                  next piece: piece pointer;
                  contents: array [1..piece length]
                                of integer
                end;
```

Body analysis leaves a short record behind in the heap, called the interpass record. This record contains the number of routine labels, the number of jump labels, and the length of the constants area. These entries were computed by earlier passes. Declaration analysis determined the number of routine labels, syntax analysis determined the number of jump labels, and name analysis determined the length of the constants area. Code selection uses this information to allocate the tables as part of pass initialization; dynamic table extension is not required.

Three routines perform table management. The allocate routine takes the number of table entries required and returns a pointer to the constructed table. This routine is called during pass initialization, once for each of the four tables. The enter routine takes a table pointer, an entry

index, and an entry and enters it in a table. The *entry* routine takes a table pointer and an entry index and returns the entry.

* address definition *

Program labels are divided into two groups, routine labels and jump labels. Routine labels appear, one to a routine, at the start of each routine body. These labels were generated by declaration analysis as part of the analysis of routine declarations. Jump labels appear within a routine body. These labels were generated by syntax analysis when it converted statements to postfix notation.

Jump labels appear in the input code as a *label* command followed by a label number. When encountered, the current program address is entered in the jump label table using the label number as an index. Code assembly, the next pass, will use this table to replace the label by a relative address in jump instructions. Three types of jump instructions exist, the *jump*, the *false jump*, and the *case jump*. The *jump* and *false jump* commands are followed by a label number. The *case jump* command is followed by the minimum and maximum case label values, and (maximum - minimum + 1) labels. These jump commands are output by code selection as they came in with the addition of one more

argument. Since all jumps in the machine use relative addressing, the current program address is appended to all jump instructions. Then the next pass will take the difference between the jump label address and the jump instruction address as the argument of the jump instruction.

Routine labels appear in the input code as arguments to the enter command. This command begins all routine bodies. When encountered, the current program address is entered in the routine label table. Code assembly, the next pass, will use this address to replace the label in call instructions. The current program address is again included as an argument to the call instruction.

* stack requirements *

The compiler computes the maximum run-time stack requirements of routines and components. Since routine calls may be dynamically nested, these stack requirements must be computed for the worst case call sequence. This is possible to perform at compile time because Concurrent Pascal forbids recursion. Recursion is allowed in sequential Pascal, and here the programmer may reserve additional stack space for processes that call separately compiled sequential programs.

The absence of recursion and forward references means that only previously defined routines are referenced. This makes it simple to compute a routine's maximum stack requirements. The current stack extent is kept in one global variable, and its high water mark in another global variable. The first routine in a program cannot call any other routine in the program. So its stack requirements are only those for its own variables and temporaries. Subsequent routines may call previous routines. Whenever a call is encountered, the called routine's stack requirements are added to the current stack extent. If this exceeds the high water mark, then the high water mark is updated. Immediately after the call, the current stack extent is decremented by the sum of the previous stack requirements plus the parameter length. (The parameter length is added piecemeal to the current stack extent as code is generated to push each argument on the stack before the call.) In this way the movement of the stack at runtime is simulated by code selection. At the end of the routine, the high water mark becomes the routine's stack requirement.

Routine and component stack requirements are placed in the stack table by code selection. Code assembly will remove them from the table and place them in the enter instruction at the beginning of the routine. The index of the stack requirement in the table is the routine label.

* the constants table *

Concurrent Pascal allows enumeration constants, real constants, and string constants. The empty set is a special case; it is the only set constant. Enumeration constants are short constants. They are included in the code as part of the instruction that references them. All other constants are long constants. Long constants are housed in the constants table and referenced by their displacement. The constants table is constructed piecemeal by code selection as large constants appear in the input code. Code assembly outputs the constants table following the code at the end of the pass.

* command translation *

Code selection performs simple encoding of types into opcodes to make the simulation of the virtual machine faster. This function is performed here to isolate the semantic analysis from details of the machine simulation. It permits peephole optimization of the instruction set without alteration of semantic analysis (this is not done in the present compiler).

Code selection accepts less than fifty different commands from body analysis. By merging arguments with operators, this set of commands is more than doubled. For

example the <u>add</u> command is changed into either the <u>add word</u> instruction or the <u>add real</u> instruction. Even with this larger operator set, many potential instructions are still absent. For example the "<u>push value</u> (word length, local routine mode, displacement)" command becomes the "<u>push local</u> (displacement)" instruction. No similar instruction exists for reals. A "<u>push value</u> (real length, local routine mode, displacement)" command becomes two instructions. A "<u>local address</u> (displacement)" instruction followed by a <u>push real</u> instruction. If all possible permutations of commands and their arguments were made separate instructions, the instruction set would be much larger.

Source line numbers may or may not appear as instructions in the final code. A compiler option determines if line number instructions are to be generated for every source line (and at every jump label) or only at the beginning of routines. This permits a runtime error indication of the particular source line that failed.

10. Code Assembly

* function *

Code assembly is the last compiler pass. It completes the transformation of the program to final machine code. Routine labels and jump labels are replaced by program addresses, stack lengths are inserted in routine entry and process initialization instructions, error messages are listed, and the constants table is output at the end of the program.

* table manipulation *

The four tables constructed by code selection are used in this pass. Label addresses are retrieved from either the routine label table or the jump label table and used to resolve call or jump instructions. Stack lengths are retrieved from the stack table and inserted in routine entry and process initialization instructions. At the conclusion of the pass, the constants table contents are appended to the code.

* error messages *

This pass prints error messages, if any, on the program listing. Earlier passes, whenever they encounter an error, output an _error_ operator with its arguments the pass number and error message number. This insures that error messages from different passes will be listed in order of line number. Code assembly then processes _error_ operators and prints the associated error messages. Error messages are in plain text. They consist of the source line number and a short explanatory message.

11. Interpass Topics

* overview *

Several interesting topics have been left out of the compiler description by passes. These topics spread over several compiler passes and are best described in a separate chapter. Constants handling, the case statement, and the with statement are included. These topics have been mentioned in previous chapters, but not completely or coherently. Their treatment here shows how a number of complicated constructs may be handled in stages.

* constants handling *

Anonymous constants are parsed and their values inserted in the intermediate code by lexical analysis. Anonymous constants fall into five categories: integers, reals, characters, strings, and sets. Integer and character constants are short constants. They will be incorporated in the code as arguments. Real, string, and set constants are long constants. They will appear in a separate constants table. Only one set constant exists, the empty set. The empty set is the first entry in the constants table.

Named constants are given values or constants table displacements in name analysis, the third pass. Name analysis replaces short named constants by their values and long named constants by their displacements. So constant declarations are the sole responsibility of name analysis. All other declarations are handled by declaration analysis.

Constant references are included in the commands by body analysis as part of its operand addressing responsibility. A short constant is referenced by a "push const (value)" instruction, and a long constant is referenced by a "const addr (displacement)" instruction.

* the case statement *

Case labels are constants; they are handled by name analysis. That pass collects all the case label values, assures they lie in the range [0, 127], and insures there are no ambiguous labels. A 128-element array is used for these operations. At the conclusion of the case statement, a transfer vector is placed in the output code. This transfer vector has (maximum - minimum + 1) entries, where [minimum, maximum] is the range of case label values. The transfer vector is indexed by case label values; its entries are the jump labels for the individual cases. For example, the case statement

<u>case</u> x <u>of</u> 3: S1; 5: S2 <u>end</u>

would appear in the name analysis output as the case statement code followed by the transfer vector (l1, ln, l2). Label 'l1' is the jump label of statement 'S1'. Label 'ln' is the jump label of the end of the case statement. This is because no case exists for 'x = 4', so the case statement will be skipped. Label 'l2' is the jump label of statement 'S2'.

Body analysis performs case label type checking to insure that the case selector expression and case labels are of compatible type. Since name analysis collects case labels at the end of the statement, it places special type checking operators in the output. These operators take as their argument the name index of the case label type. Body analysis uses these operators to compare the case label types with the type of the selector expression.

* the <u>with</u> statement *

Concurrent Pascal's with statement may name a system component or a record variable. This introduces the entry names or field names into the scope at that point. From the point of view of name analysis, no real difference exists between these two cases. An example for a record variable might be:

```
var
  record variable:
    record
      field: integer
    end;
begin
  with record variable do
    field:= 0
end;
```

The semantics of the with statement are simple. The address of the with variable is evaluated and treated as a temporary pointer until the end of the statement. This temporary is then used to qualify the entry or field names in the body of the with statement. This can be stated in high level terms. Let the expression

```
p ref v
```

mean "assign the address of variable 'v' to pointer 'p'". Then the with statement example above is treated as

```
with temporary ref record variable do
  temporary@.field:= 0.
```

Name analysis translates the original with statement format to this new format. It explicitly introduces a declaration of the with temporary, and it qualifies entry or field names with the newly created temporary.

Declaration analysis processes the declaration of the with temporary. This is an example of a declaration inside the body. Declaration analysis assigns a displacement to the with temporary. This displacement is the stack displacement where the with variable address is evaluated. At the end of the with statement, the temporary is popped from the stack.

Body analysis generates the commands to evaluate the address of the with variable. Inside the body of the with statement, it generates commands to push the with temporary on the stack wherever a qualified name appears.

* remarks *

Altogether these are good examples of how semantic analysis can be split over several different passes. Each pass performs a well-defined subset of the semantic analysis process. All this is done with no pass having a complete symbol table. Context awareness is strictly limited to the immediate program neighborhood. This requires a systematic design of the entire compiler, with a clean development of each pass and its interrelation to other passes.

12. The Virtual Machine

* introduction *

The Concurrent Pascal compiler generates code for an ideal machine. The transportable compilers developed by Wirth's group at the Technical University (ETH) in Zurich compile code for an ideal machine. Concurrent Pascal's ideal machine was designed by Per Brinch Hansen. The following discussion is adapted from his description of the machine. This ideal machine is simulated by the real machine, in our case a Digital Equipment Corporation PDP-11/45. Certain peculiarities of the real machine (e.g. program relative addressing) appear in the final code. These peculiarities are introduced during the code assembly phase of compilation. For this reason, the machine is best described at the point just before code assembly when the intermediate language is still adaptable to general machine architectures. This then is a description of the machine as viewed by semantic analysis. The machine instruction set is the command set generated by the last semantic pass, body analysis. The two-pass code assembler may be viewed as a postprocessor that adapts this virtual machine code to particular architectures.

The virtual machine is an ideal stack machine. No assumptions are made about particular registers in any real machine. The virtual instruction set may be assembled into real code for any machine on an instruction by instruction (context free) basis.

* data types *

The virtual machine recognizes five types of data:

1) byte - used to represent a single character within a string;
2) word - used to represent enumeration types, queues, and processes; must be one or more bytes in length;
3) real - used to represent a real; must be one or more words in length;
4) set - used to represent a set; must be one or more words in length;
5) structure - used to represent a structured type, class, or monitor.

The first four data types are fixed length, while the structure type is variable length. Byte data represents characters within strings. The representation is ASCII. Word data represents enumeration types. Programmer defined enumeration types are represented by consecutive integers 0, 1, 2, ... Integer and real representations are

determined by the particular machine. Sets are represented as bit strings. The virtual machine uses fixed length sets. Program components are represented by indices 1, 2, 3, . . . defined during system initialization. The index zero represents an uninitialized component. Queues contain process indices. An empty queue contains index zero.

* data addresses *

The data store contains a constant segment, a stack, and a heap. Addresses contain a mode and a displacement. Modes and displacements were described in the discussion of declaration analysis. The apparent intricacy of displacement assignment actually results from a very simple data segment design. The data segment of a component or a routine has the same structure. This structure is diagrammed below:

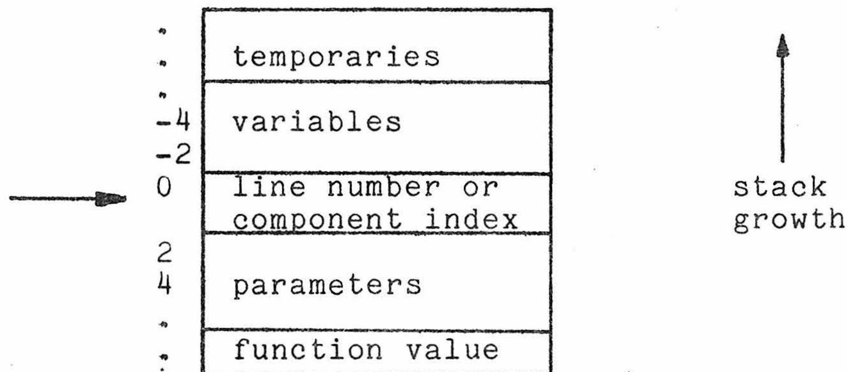


Figure: A data segment.

The base address of a data segment divides the variables from the parameters. Component data segments have their address shifted by the variable length, or <u>component offset</u>, in order to point to the base location. The base location contains either the line number at the point of call for routines, or it contains the component index for system components. The parameter portion may contain more than just the explicit parameters. A call of a routine entry results in the component address being placed on the stack before the explicit parameters. This allows the routine entry to address global component variables. A call of a sequential program places a list of interface routine addresses on the stack before the explicit parameters. After the explicit parameters, the address of the sequential code store is placed on the stack.

* program addresses *

Program addresses are represented by integer labels. A label is either a jump label or a routine label. Jump labels are labels of points within the body of a routine. Routine labels are the labels attached to the beginning of a routine body. All references to jump labels are <u>within</u> a routine. All references to routine labels are <u>between</u> routines.

* the virtual commands *

A virtual command consists of one or more integers. The first integer is the operator. Subsequent integers, if any, are the arguments. For example, the field command takes a displacement as its argument. The top stack location contains an address and the field command increments the address by the displacement. The instruction set is remarkable in that it is unremarkable. It implements Concurrent Pascal with a simple set of commands for manipulation of a stack machine.

13. Implementation

* history *

The Concurrent Pascal compiler was inspired by the Gier Algol compiler [5] of thirteen years ago. That effort showed that compilation can be made simple and efficient by using a large number of small passes. The Gier compiler used nine passes. Lexical analysis used two passes, syntax analysis used one pass, semantic analysis used three passes, and code assembly used three passes.

Additional inspiration for this compiler was obtained from the ten-years' old Cobol compiler for the Siemens 3003 [4]. This was a ten-pass compiler. Lexical analysis used one pass, syntax analysis used one pass, semantic analysis used four passes, and code assembly used four passes.

Both the Gier and Siemens compilers were written in assembly language and generated code for the real machine. This accounts for differences in code assembly between these compilers and the Concurrent Pascal compiler. The small size of the Gier computer required that the compiler separate identifier matching from the balance of lexical analysis to obtain two passes. For syntax analysis, both previous compilers used the traditional transition matrix technique (instead of recursive descent).

* testing *

The Concurrent Pascal compiler transmits intermediate code between its passes. As in the Gier and Siemens compilers, the sole diagnostic information is a listing of the intermediate and final code. If the compiler crashes or loops endlessly, the operating system ensures that all intermediate code up to the point of failure is listed. This listing is controlled by an option switch within the compiler.

The compiler was developed using a set of systematically developed test cases for each pass. These test cases are Concurrent Pascal programs that make each pass generate every possible operator and execute every statement at least once. At least two test programs are written for each pass. One program is entirely correct for the pass; the other program generates every possible error in the pass. Pass 1 is tested first. This pass, lexical analysis, requires a special test mechanism. Since lexical analysis lists the source program, the listing of the first intermediate code must be interleaved with the source listing. Lexical analysis buffers test output between source lines. This alternates source lines with their intermediate code.

Once the first pass is complete, the second pass is tested, and so on. As each pass is finished, the next pass is added. This allows all test programs to be in source text form, and it tests all interpass assumptions. At each phase of the testing <u>all</u> test programs are used, not just the test programs for the new pass. Generally when a new pass is first added, all test programs will fail. Several of these failures will point out different bugs, and these may be discovered and corrected simultaneously. As testing progresses, more and more test programs will be compiled without failure, until finally all test programs compile successfully.

The output of the test cases is the intermediate code. This is a sequence of integers. Each integer is either an operator or an argument in the intermediate language. Operators appearing on the test listing are preceded by the letter 'C', arguments are not. The test output mechanism always remains in the compiler as an option. Once released, users may use the test option if they encounter a compiler failure. The listing can then be mailed to the compiler writer for examination and correction. Compiler changes may also be tested with this mechanism.

Compiler failures during testing are normally detected by runtime checks in the virtual machine. Three types of checking are performed. Variant checking insures that variant fields are only referenced when the tag field contains an appropriate value. This check is vital during testing of name analysis where a large linked structure of variant records is created. In a sample of 64 compiler failures during testing, 18 failures, or 28%, were variant errors. Pointer checking insures that nil pointer values are not used as references. Again this is valuable in any pass with a linked structure. In the sample of 64 failures, 13 failures, or 20%, were pointer errors. Range checking insures that subscripts and case statement selectors are within range. In the sample of 64 failures, 32 failures, or 50%, were range errors.

The sample of compiler failures was taken after name analysis had nearly been completely tested. Name analysis resulted in the most variant and pointer errors. Probably over the entire compiler, the proportions of variant, pointer, and range errors were fairly close. The value of these checks is enormous. In the total sample of 64 failures, only one failure, or less than 2%, was an endless loop!

Systematic testing of the compiler occupied three months, from October through December of 1974. Actual PDP-11 machine time during testing was a twenty-minute session, twice daily. The compiler had been designed and written during the summer, June through September. It was written backwards, starting with the last pass.

* performance - space *

Before writing the Concurrent Pascal compiler, a small group first wrote a compiler for the sequential Pascal language as defined by Wirth [7]. This was a six-pass compiler that generated a combination of real and virtual code for the DEC PDP-11/45. In this compiler semantic analysis used two passes instead of three. Name analysis was split between declaration and body analysis. This made the two semantic analysis passes nearly equal in size, both far larger than the other passes. Declaration analysis constructed the symbol table for the entire program, and it remained in the heap between the two passes. A space requirement histogram for the six passes is shown next. The space measured is the sum of the program and data space for the pass. The data space is sufficient for self-compilation.

Total space requirement for six-pass compiler.

```
Pass   Space requirement; * = 500 16-bit words.
 1  (14k) ****************************
 2  (13k) **************************
 3  (22k) ********************************************
 4  (23k) **********************************************
 5  (12k) ************************
 6  (12k) ************************
```

Lessons learned from the six-pass compiler lead to the development of a totally new design for a seven-pass Concurrent Pascal compiler. The key element in the new design is the name analysis pass, an idea that goes back to the Gier Algol compiler [5]. The space requirement histograms for the seven passes of the Concurrent Pascal compiler are shown next. The improvement in space utilization is striking. The data space is sufficient for compilation of the Solo System, Brinch Hansen's operating system written in Concurrent Pascal. A sequential version of the Concurrent Pascal compiler was constructed in one additional month (January 1975). The total space required for self-compilation of the sequential Pascal compiler is 256 words larger than the maximum space shown for Concurrent Pascal.

Total space requirement for Concurrent Pascal compiler.

```
Pass    Space requirement;  * = 500 16-bit words.

 1  (12   k) ************************
 2  ( 9   k) ******************
 3  (16.5k) *********************************
 4  (13   k) **************************
 5  ( 6.5k) *************
 6  ( 5.5k) ***********
 7  ( 6   k) ************
```

Program space requirement for Concurrent Pascal compiler.

(About 1000 words of program are common I/O routines in each pass.)

```
Pass    Space requirement;  * = 500 16-bit words.

 1  (5.5k) ***********
 2  (6.5k) *************
 3  (9   k) ******************
 4  (7   k) **************
 5  (5   k) **********
 6  (4   k) ********
 7  (4.5k) *********
```

The program space requirements reflect the choice of compile-time options chosen for the pass compilations. Line numbers may optionally appear in the code, as may variant, pointer, and range checks. A line number always appears for each routine even if the line number option is turned off. A range check is built into the indexing and case jump instructions. Turning off the check option will not remove subscript and case selector range checks. With this in mind, all passes were compiled without line numbers. All passes except Pass 1 were compiled with checks. The importance in the choice of compiler options is shown next. The program space for each pass is shown for the three cases (1) with line numbers and with checks, (2) without line numbers but with checks, and (3) without line numbers and without checks. On the average, case (2) is 75% the size of case (1), and case (3) is 70% the size of case (1).

Program space requirements for Concurrent Pascal compiler.

First line with line numbers and with checks.

Second line without line numbers but with checks.

Third line without line numbers and without checks.

Pass   Space requirement; * = 500 16-bit words.

```
    (7000)  **************
 1  (5300)  ***********
    (5000)  **********

    (8500)  *****************
 2  (6600)  *************
    (6300)  *************

   (11300)  ***********************
 3  (8800)  ******************
    (7800)  ****************

    (8800)  ******************
 4  (6800)  **************
    (6000)  ************

    (6800)  **************
 5  (5000)  **********
    (4800)  **********

    (5800)  ************
 6  (4000)  ********
    (4000)  ********

    (6500)  *************
 7  (4600)  *********
    (4300)  *********
```

Pass data for compilation of the Solo System may be estimated as follows:

Data space requirement for Concurrent Pascal compiler.

Pass Space requirement; * = 500 16-bit words.

```
1  (7  k) **************
2  (2.5k) *****
3  (7.5k) ***************
4  (6  k) ************
5  (1.5k) ***
6  (2  k) ****
7  (2  k) ****
```

Common data for all passes (16-bit words)

Call of Solo command interpreter	100
Command interpreter data	370
Call of compiler driver	100
Compiler driver data	100
Call of a pass	100
Pass code buffers	514
Total	1284

<u>Pass 1 data (16-bit words)</u>

Hash table - 7 words * 751 entries	5257
Other variables	230
<u>Fixed data total</u>	<u>5487</u>
Local data	25
Long identifiers - 6 words * 17 entries	102
<u>Dynamic total</u>	<u>127</u>
<u>Fixed + dynamic total</u>	<u>5614</u>

<u>Pass 2 data (16-bit words)</u>

Constant keys sets - 8 words * 66 sets	528
Other variables	10
<u>Fixed data total</u>	<u>538</u>
Recursion - 22 words avg. * 30 levels	660
<u>Fixed + dynamic total</u>	<u>1198</u>

<u>Pass 3 data (16-bit words)</u>

Operand stack - 3 words * 151 entries	453
Case label array - 1 word * 128 entries	128
Update stack - 4 words * 100 entries	400
Display - 4 words * 15 entries	60
Spelling table - 3 words * 701 entries	2103
Miscellaneous	101
<u>Fixed data total</u>	<u>3245</u>

Named constants - 4 words * 66 entries	264
Types - 4 words * 65 entries	260
Fields - 7 words * 19 entries	133
Parameters - 4 words * 170 entries	680
Varibles - 4 words * 123 entries	492
Initial statements - 4 words * 34 entries & simple routines	136
Entry routines - 7 words * 89 entries	623
Interface routines - 3 words * 46 entries	138
Standard entries - 4 words * 39 entries	156
With temporaries - 4 words * 16 entries	64
Local data	50
Dynamic data total	2996
Fixed + dynamic total	6241

Pass 4 data (16-bit words)

Noun table - 1 word * 701 entries	701
Operand stack - 1 word * 101 entries	101
Display - 3 words * 16 entries	48
Miscellaneous	116
Fixed data total	966
Symbol table 7 words	
* 65 types	455
* 19 fields	133
* 170 parameters	1190

* 123 variables	861
* 123 routines	861
* 29 standard entries	203
* 16 with temporaries	112
Local data	25
Dynamic data total	3840
Fixed + dynamic total	4806

Pass 5 data (16-bit words)

Standard operands - 9 words * 5 entries	45
Stack links - 3 words * 3 entries	9
Miscellaneous	111
Fixed data total	165
Stack entries - 12 words * 5 entries	60
Local data	25
Dynamic data total	85
Fixed + dynamic total	250

Pass 6, 7 data (16-bit words)

Miscellaneous	29
Fixed data total	29
Local data	15
Labels - 92 labels	101
Blocks - 123 blocks	202
Large constants - few	101

Stack table - 123 blocks	202
Dynamic data total	621
Fixed + dynamic total	650

Total compiler data space (16-bit words)

Common	1300
Pass 1	5600
Pass 2	1200
Pass 3	6200
Pass 4	4800
Pass 5	250
Pass 6, 7	650
Total	20,000

Total compiler program space (16-bit words)

Common	1000
Pass 1	4000
Pass 2	5600
Pass 3	7800
Pass 4	5800
Pass 5	4000
Pass 6	3000
Pass 7	3600
Total	34,800

Excluding a common prefix of 70 lines, and excluding common I/O routines of 150 lines, the length in lines of each pass's source text is:

<u>Concurrent Pascal compiler source text length (lines).</u>

Common	220
Pass 1	768
Pass 2	1079
Pass 3	1515
Pass 4	1182
Pass 5	943
Pass 6	863
Pass 7	912
Total	7482

* performance - time *

The speed of the original six-pass compiler running under Brinch Hansen's Basic System is shown next. To estimate the internal speed of the compiler, a dummy six-pass compiler was measured that performed only the I/O operations of the real compiler. All times were measured with a stop watch. The average source line length for the test program is 25 characters. Pass 5 of the six-pass compiler served as the test program.

Six-pass compiler speed

Null program time	11 sec
Pass 1 internal speed	2035 char/sec
Compiler internal speed	678 char/sec
Pass 1 overall speed	1221 char/sec
Compiler overall speed	480 char/sec

A similar experiment was performed for the Concurrent Pascal compiler running under Brinch Hansen's Solo System. The average source line length for the test program is 20 characters. The Solo System itself served as the test program.

Concurrent Pascal compiler speed

Null program time	7 sec
Pass 1 internal speed	2318 char/sec
Compiler internal speed	843 char/sec
Pass 1 overall speed	605 char/sec
Compiler overall speed	236 char/sec

The Concurrent Pascal compiler is 24% faster internally than the six-pass compiler. This combines with a 27% reduction in space required for the Concurrent Pascal compiler. The poor showing for overall speed in the new

design reflects two factors. One factor is the greater amount of I/O in the seven-pass compiler due to the additional pass and to distribution of the symbol table in the intermediate code. The size of the output code for each pass of the two compilers for their respective test cases is:

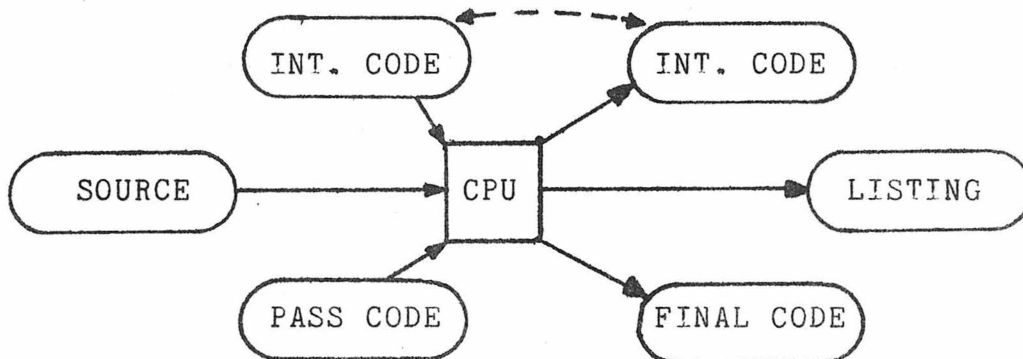
Pass	six-pass	Concurrent
1	8,205	10,368
2	8,240	10,880
3	6,728	10,880
4	7,474	15,744
5	6,060	9,344
6	6,075	5,504
7	0	5,248
Total	42,782	67,968

The sizes of the test cases are nearly the same while the intermediate code size is greatly expanded in the new compiler. Another factor influencing the overall speed difference is the relative I/O efficiency of the Basic System (written in assembly language) and the Solo System (written in Concurrent Pascal). This comparison has not been undertaken.

* file system *

The Concurrent Pascal compiler relies heavily on an efficient implementation of sequential I/O. In the performance evaluation, approximately three-quarters of total elapsed compile time is I/O time. A rough estimate would divide I/O time equally between CPU time for I/O administration and physical wait time for the disk.

The compiler uses six separate files, each of which is sequential. A diagram of the filing system is shown next:



Compiler file system.

The performance of a many-pass compiler is greatly affected by the performance of sequential I/O. The null program time is largely pass loading time. The speed of Pass 1 is largely the speed at which the source file can be read. The speed of later passes is largely the speed the intermediate code files can be accessed.

* further work *

The performance of this system can be greatly improved with little effort. Generating code in byte-length units instead of word-length units would shrink the code size by nearly half. Placing the 1-K word virtual machine interpreter into read-only memory would double its speed. This system uses the slowest main memory and slowest disk manufactured for the PDP-11/45. Peephole code optimization could easily be added to the small code assembly passes.

The removal of classes from sequential Pascal was a mistake. The compilers would probably be smaller and simpler if they were written with classes. The class concept simplifies the handling of data structures by hiding their implementation details. This permits the use of classes as abstract types. Another use of classes is to collect routines into manageable groups. A single routine of 100 statements is difficult to understand; a single program of 100 routines is even worse. Classes impose a hierarchy on routines just as routines impose a hierarchy on statements. Unlike routines, classes may be nested and the program built up in "layers". Concurrent Pascal's scope rules turn this multilevel hierarchy into only two levels at runtime.

This project shows that a very simple machine is ideal for Concurrent Pascal. Optimization improves the match between language and machine. Assembly languages and real machines are so closely matched that they are very efficient. Using these same machines with high-level languages can result in a mismatch and loss of efficiency. Optimizing compilers may correct this at some expense. I believe a better way to optimize is to restore the match between language and machine. The language syntax must simply express the intended semantics. These semantics should be simple to understand and implement. And they should readily map onto the instruction set of an ideal machine.

14. References

[1] Brinch Hansen, P. Concurrent Pascal Report. Information Science, California Institute of Technology, June 1975.

[2] Brinch Hansen, P. The programming language Concurrent Pascal. IEEE Transactions on Software Engineering 1, 2 (June 1975), 199-207.

[3] Brinch Hansen, P. and House, R. The Cobol compiler for the Siemens 3003. BIT 6, 1 (1966), 1-23.

[4] Knuth, D. Structured programming with _go to_ statements. Computing Surveys 6, 4 (Dec. 1974), 261-301.

[5] Naur, P. The design of the Gier Algol compiler. BIT 3, 2-3 (1963), 124-140 and 145-166.

[6] Wirth, N. The design of a Pascal compiler. Software 1 (1971), 309-333.

[7] Wirth, N. _Systematic Programming_. Prentice-Hall, Englewood Cliffs, N.J., 1973.

APPENDIX

Concurrent Pascal - Syntax Graphs

Definitions

active type: type containing class types, monitor types, process types, or queue types.

active variable: a variable of active type.

argument: an expression passed in an argument list.

arithmetic type: an integer or real range.

component parameter: a parameter to a system component type.

component variable: a variable declared at the beginning of a system component type.

constant parameter: a parameter defined without the var keyword.

entry routine: a procedure entry, function entry, or initial statement.

index type: a symbolic scalar (including boolean), integer, or character type.

large type: array or record type.

parameter: an identifier declared in a parameter list.

passive type: a type not containing class types, monitor types, process types, or queue types.

passive variable: a variable of passive type.

queue variable: a variable of a type containing a queue type.

routine: a procedure, function, program, or initial statement.

scalar type: a real or index type.

small type: a scalar type or set type.

string type: an array of characters.

system component: a variable of type class, monitor, or process.

type compatibility: two types are compatible if

1) they are defined by the same type definition; or

2) both are subranges of a single type; or

3) they are string types of the same length; or

4) they are set types whose members are of the same index type; or

5) they are set types, one (or both) of which is the null set type.

universal type: a parameter type defined with the <u>univ</u> keyword.

variable parameter: a parameter defined with the <u>var</u> keyword.

Syntax and Rules

The rules are preceded by a parenthesized number that refers to the compiler pass responsible for rule enforcement. The numbers and their associated passes are:

1. lexical analysis
2. syntax analysis
3. name analysis
4. declaration analysis
5. body analysis
6. code selection
7. code assembly

Pass 1 - input syntax description

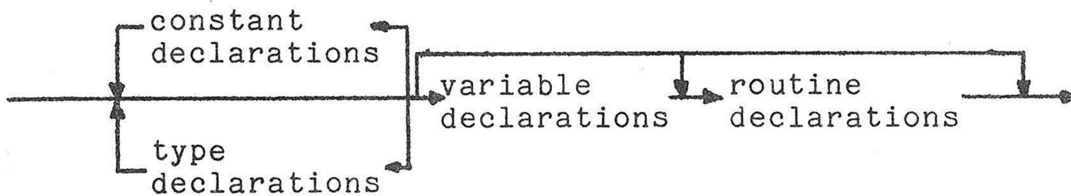
1. program

→ block → . →

2. block

→ declarations → body →

3. declarations



4. constant declarations

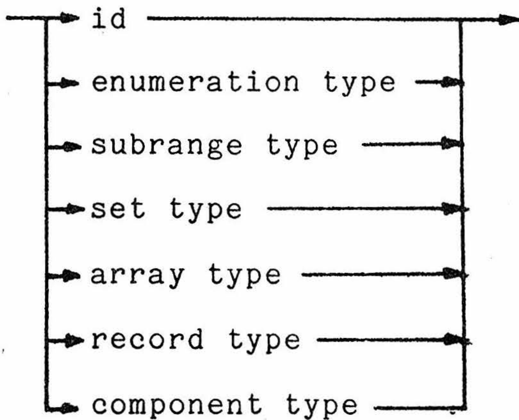
→ const → id → = → constant → ; →

5. type declarations

→ type → id → = → type → ; →

(3) The type definition may not reference its own type identifier.

6. <u>type</u>



(3) The id must be a type identifier.

7. <u>enumeration type</u>

→ (→ id list →) →

(4) No more than 128 values may be enumerated.

(4) It may not be defined within a record type.

8. <u>subrange type</u>

→ constant → .. → constant →

(3) The lower bound must not exceed the upper bound.

(3) The constants must be of compatible index types.

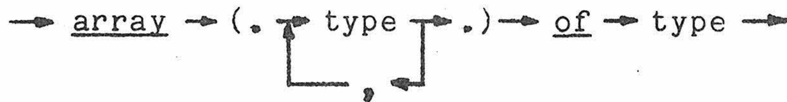
9. <u>set type</u>

→ <u>set</u> <u>of</u> → type →

(4) The member type must be an index type.

(4) The bounds of integer member types must lie in the range 0..127.

10. array type



(4) The subscript types must be index types.

(4) String length mod word length must be zero.

11. record type

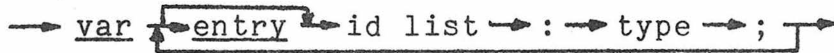
12. component type

(4) A system component type may only be nested within another system component type (but not within a record or routine). The entire program is an implied process type.

(4) The "offset" of system component types must be accumulated and associated with the type.

(4) Stack lengths may only be specified for processes.

13. <u>variable declarations</u>



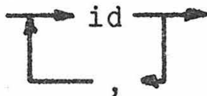
(4) Entry variables must be passive component variables of class types.

(4) Active variables must be component variables.

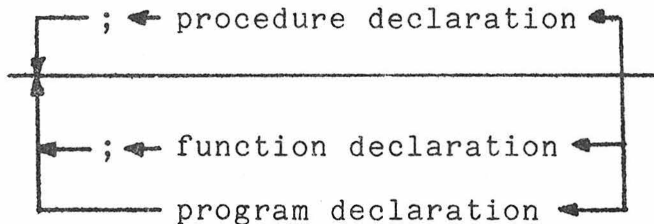
(4) Queue variables must be monitor component variables.

(4) Process components must be component variables of the initial process.

14. <u>id list</u>

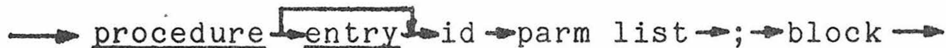


15. <u>routine declarations</u>



(4) Routine declarations cannot be nested.

16. <u>procedure declaration</u>



17. <u>function</u> <u>declaration</u>

→ <u>function</u> [→ <u>entry</u>] → id → parm list → : → id → ; → block →

(3) The last identifier must be a type identifier.

(4) Function types must be index types.

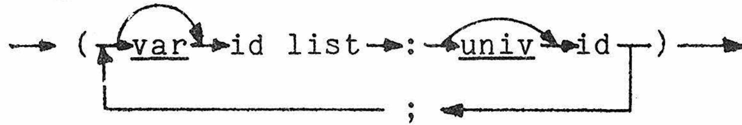
18. <u>program</u> <u>declaration</u>

→ <u>program</u> → id → parm list → ; [→ <u>entry</u> → id list → ;] →

(3) The interface must name only entry routines within the same component type as the given program. These may be forward references.

( ) The last parameter is <u>assumed</u> to be a passive code variable.

19. parm list



(3) The last identifier must be a type identifier.

(4) Universal types must be passive.

(4) Component parameters must be of small type or they must be monitor components, with the exception class components may be parameters of other class components.

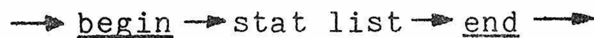
(4) Component parameters must be constant parameters.

(4) Function parameters must be constant parameters.

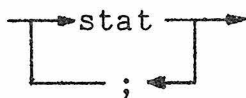
(4) Program parameters must be of passive type.

(4) Entry routine parameters may not contain queues.

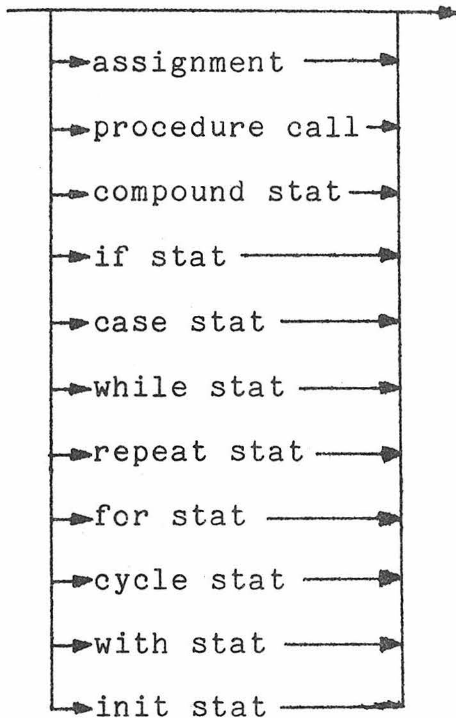
20. body



21. stat list



22. stat



23. assignment

→ variable → := → expr →

(5) The variable must be passive.

(5) The variable may not be a constant parameter.

(5) The types of the variable and the expression must be compatible.

(5) The variable may not be an entry variable outside the present component.

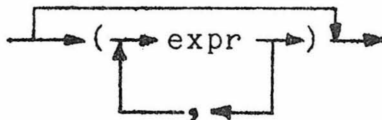
24. procedure call

(3) A routine may not reference itself.

(3) A component type may not reference its own entry routines.

(5) Process entry procedures may not be referenced.

25. arg list



(3) The arguments must correspond in number to the parameters.

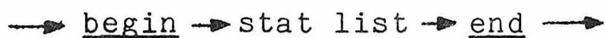
(5) The arguments must correspond in type to the parameters, with the following exceptions:

  (5) Arguments corresponding to parameters of universal type may be of any passive type of the same size as the parameter.

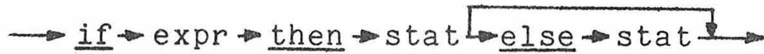
  (5) String arguments corresponding to constant non-universal string parameters may be any length.

(5) Arguments corresponding to variable parameters must themselves be variables.

26. compound stat

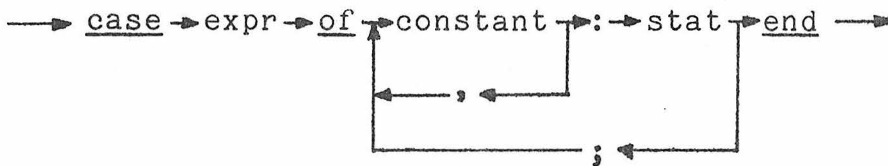


27. <u>if stat</u>



(5) The expression must be boolean.

28. <u>case stat</u>

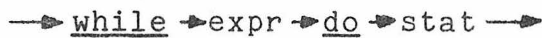


(3) The case label constants must be unique.

(3) Integer case labels must possess values in the range 0..127.

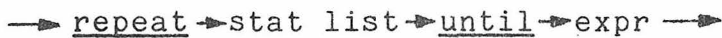
(5) The selector expression and the case label constants must be of compatible index type.

29. <u>while stat</u>



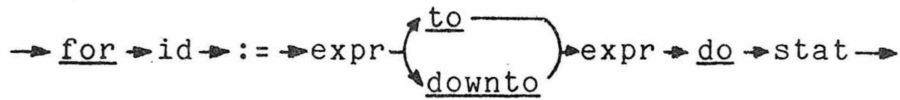
(5) The expression must be boolean.

30. <u>repeat stat</u>



(5) The expression must be boolean.

31. <u>for stat</u>

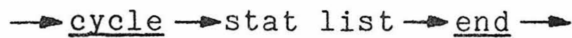


(3) The control variable may not be a record field or a function name.

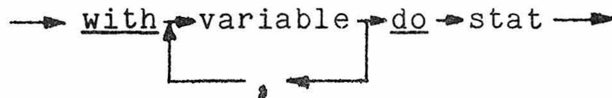
(5) The rules governing assignment apply.

(5) The control variable must be of index type.

32. <u>cycle stat</u>



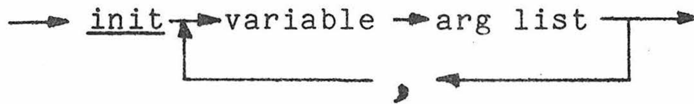
33. <u>with stat</u>



(2) The use of more than one with variable is equivalent to the use of nested with statements.

(3) With variables must be of class, monitor, or record type.

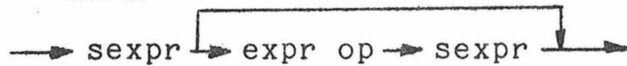
34. <u>init stat</u>



(5) System components may only be initialized within the component in which they are declared as variables (but not where they are declared as parameters).

(3) The variable must be a system component variable.

35. <u>expr</u>



(5) = and <> require compatible passive operands.

(5) <= and >= require compatible small or string operands.

(5) < and > require compatible scalar or string operands.

(5) <u>in</u> requires an index left operand and a set right operand whose member type is compatible with the left operand.

36. <u>sexpr</u>

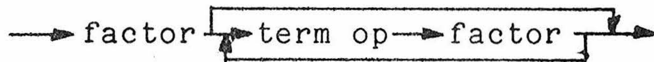
(5) Unary operators (+, -) require arithmetic operands.

(5) Binary + requires compatible arithmetic operands.

(5) Binary - requires compatible arithmetic or set operands.

(5) <u>or</u> requires compatible boolean or set operands.

37. <u>term</u>



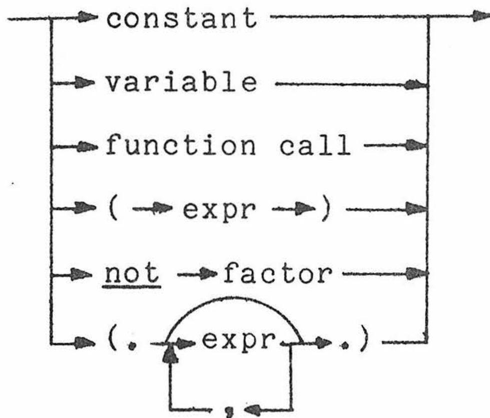
(5) * requires compatible arithmetic operands.

(5) / requires real operands.

(5) <u>div</u> and <u>mod</u> require integer operands.

(5) <u>and</u> (&) requires compatible boolean or set operands.

38. <u>factor</u>



(5) <u>not</u> requires a boolean operand.

(5) Set member expressions must be of compatible index type.

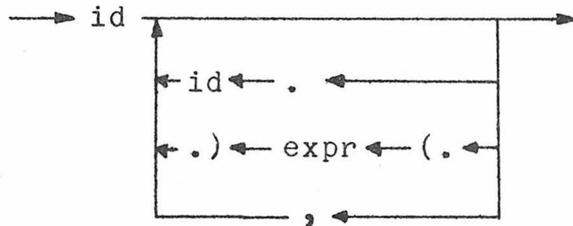
39. <u>function call</u>

(3) The identifier must be a function identifier.

(3) A routine may not reference itself.

(5) Process entry functions may not be referenced.

40. <u>variable</u>



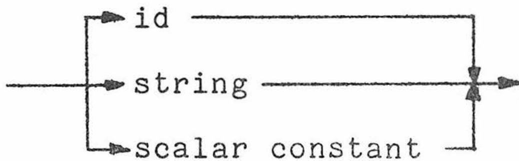
(3) Only class, monitor, or record variables may be qualified.

(3) The field or entry name must exist.

(3) Only array variables may be subscripted.

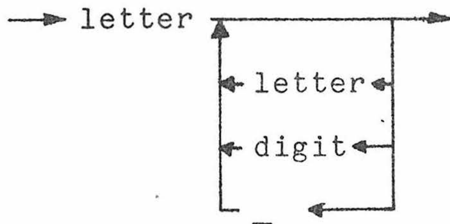
(5) The subscript expressions must be compatible with the subscript types.

41. <u>constant</u>

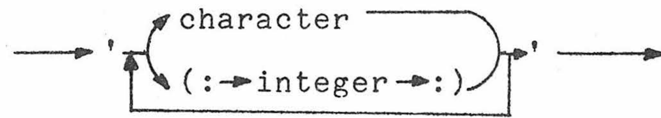


(3) The identifier must be a constant identifier.

42. <u>id</u>



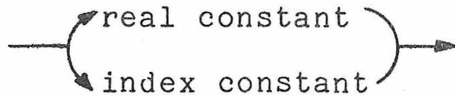
43. <u>string</u>



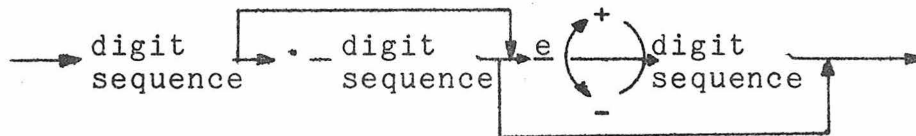
(1) The integer must lie in the range 0..127.

(1) The string length mod word length must be zero.

44. <u>scalar constant</u>

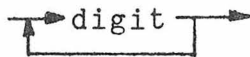


45. <u>real constant</u>

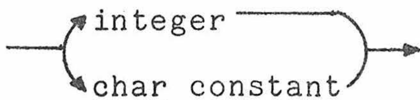


(1) The real constant must be representable on the machine.

46. <u>digit sequence</u>



47. <u>index constant</u>

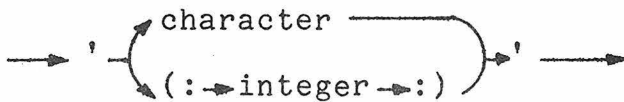


48. <u>integer</u>

→ digit sequence →

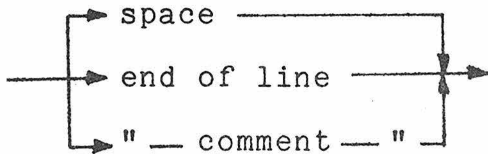
(1) The integer value must be representable on the machine.

49. <u>char constant</u>



(1) The integer value must lie in the range 0..127.

50. <u>separator</u>



(1) An arbitrary number of separators may be inserted between any two symbols except within word delimiters, identifiers, constants, and the composite operators:

.. := >= <= (. .) <>

Pass 2 - input syntax description

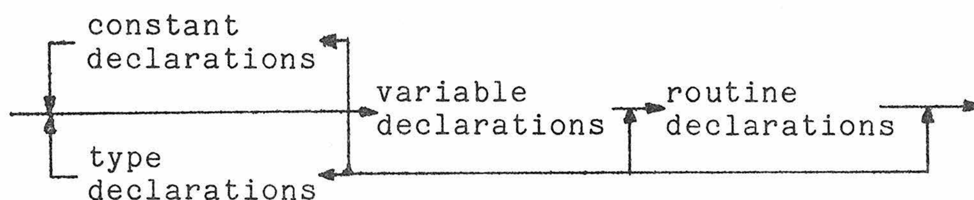
1. program

→ block → period → eom →

2. block

→ declarations → body →

3. declarations



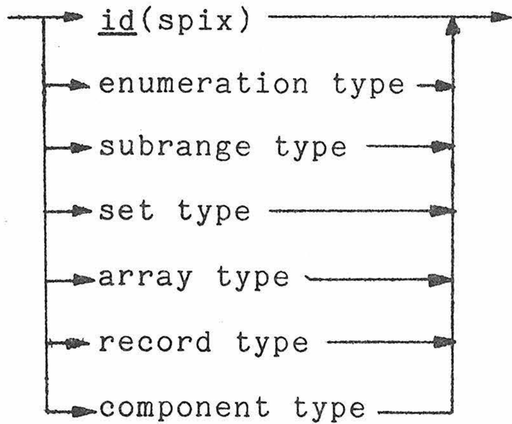
4. constant declarations

→ const → id(spix) → eq → constant → semicolon →

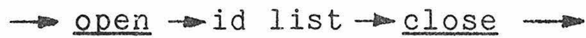
5. type declarations

→ type → id(spix) → eq → type → semicolon →

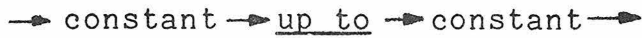
6. <u>type</u>



7. <u>enumeration type</u>

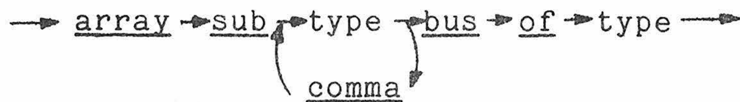


8. <u>subrange type</u>

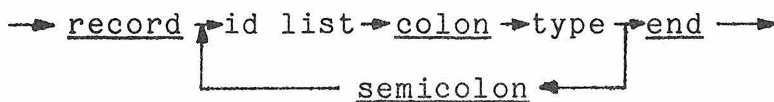


9. <u>set type</u>

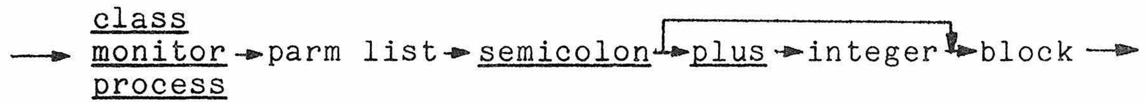
10. <u>array type</u>



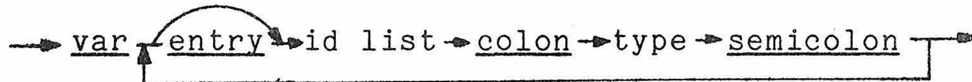
11. <u>record type</u>



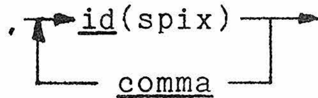
12. component type



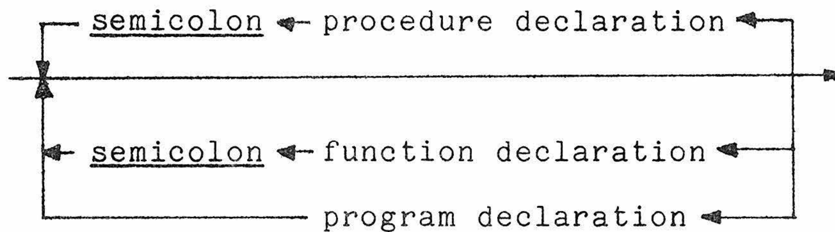
13. variable declarations



14. id list

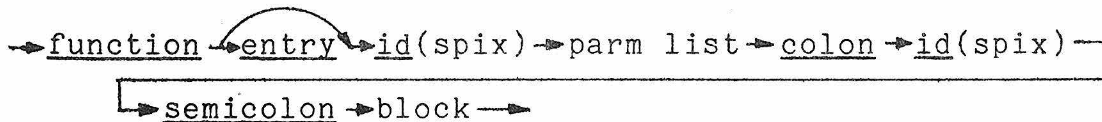


15. routine declarations

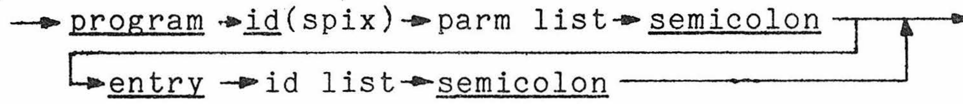


16. procedure declaration

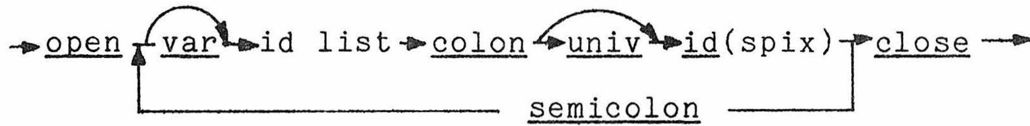
17. function declaration



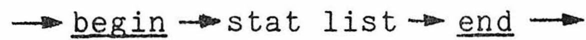
18. program declaration



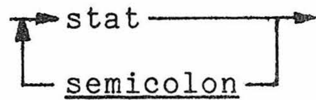
19. parm list



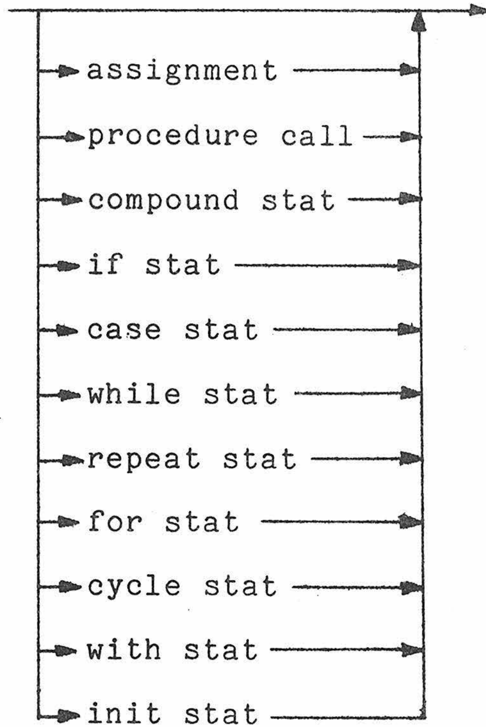
20. body



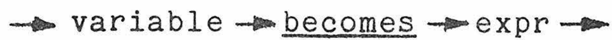
21. stat list



22. <u>stat</u>

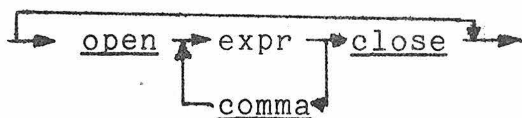


23. <u>assignment</u>



24. <u>procedure call</u>

25. <u>arg list</u>



26. <u>compound</u> <u>statement</u>

→ <u>begin</u> → stat list → <u>end</u> →

27. <u>if</u> <u>stat</u>

→ <u>if</u> → expr → <u>then</u> → stat → <u>else</u> → stat →

28. <u>case</u> <u>stat</u>

→ <u>case</u> → expr → <u>of</u> → constant → <u>colon</u> → stat → <u>end</u> →

← <u>comma</u> ←

<u>semicolon</u> ←

29. <u>while</u> <u>stat</u>

→ <u>while</u> → expr → <u>do</u> → stat →

30. <u>repeat</u> <u>stat</u>

→ <u>repeat</u> → stat list → <u>until</u> → expr →

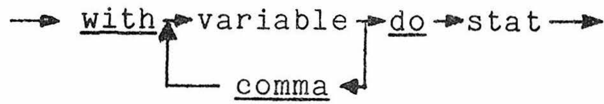
31. <u>for</u> <u>stat</u>

→ <u>for</u> → <u>id</u>(spix) → <u>becomes</u> → expr → <u>to</u> / <u>downto</u> → expr → <u>do</u> → stat →

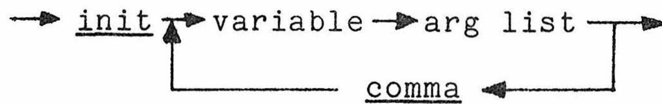
32. <u>cycle</u> <u>stat</u>

→ <u>cycle</u> → stat list → <u>end</u> →

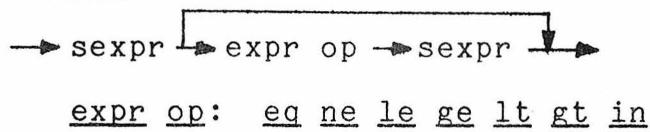
33. with stat



34. init stat

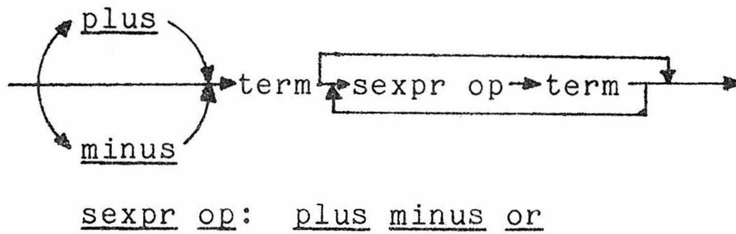


35. expr



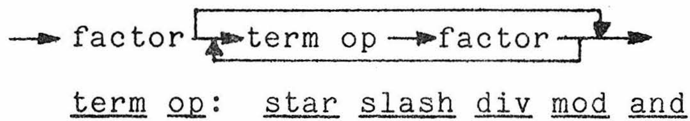
expr op: eq ne le ge lt gt in

36. sexpr



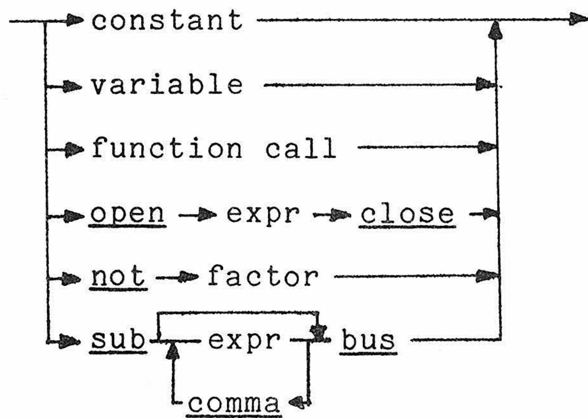
sexpr op: plus minus or

37. term



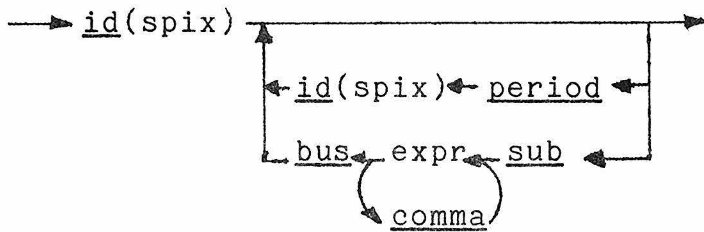
term op: star slash div mod and

38. factor

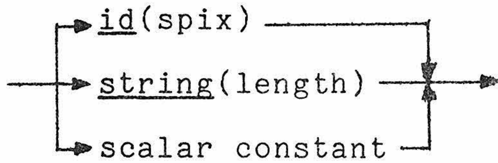


39. function call

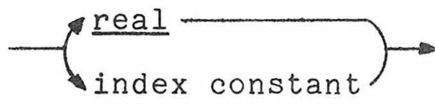
40. variable



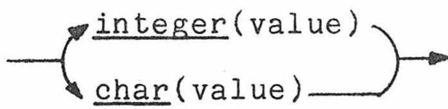
41. constant



44. <u>scalar</u> <u>constant</u>



47. <u>index</u> <u>constant</u>



Pass 3 - input syntax description

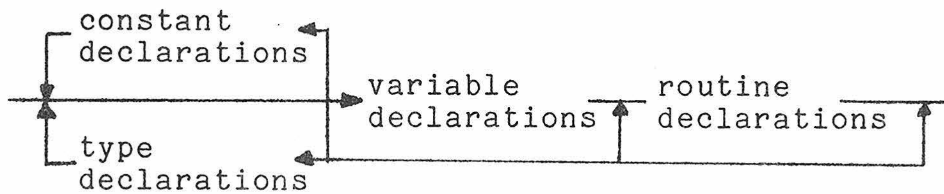
1. program

→ component type → eom →

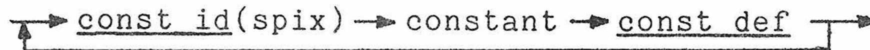
2. block

→ declarations → body →

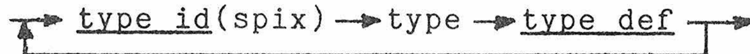
3. declarations



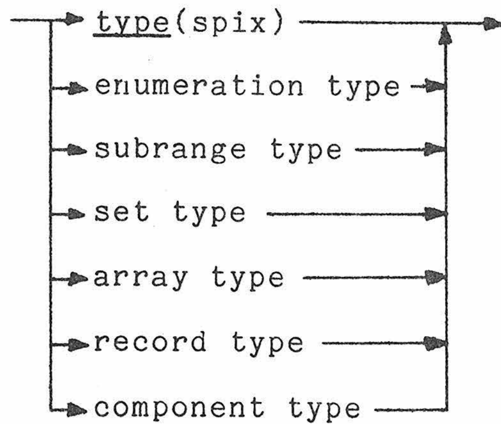
4. constant declarations



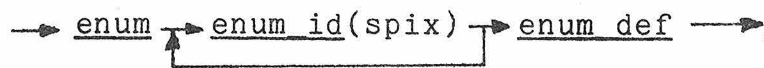
5. type declarations



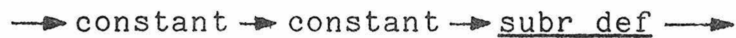
6. <u>type</u>



7. <u>enumeration type</u>



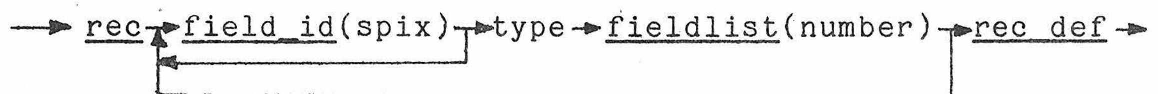
8. <u>subrange type</u>



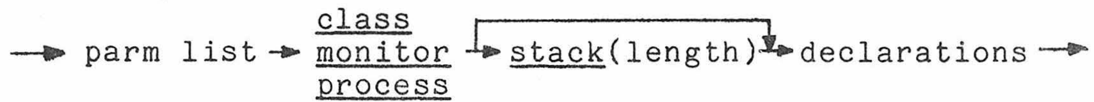
9. <u>set type</u>

10. <u>array type</u>

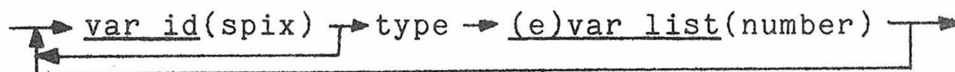
11. <u>record type</u>



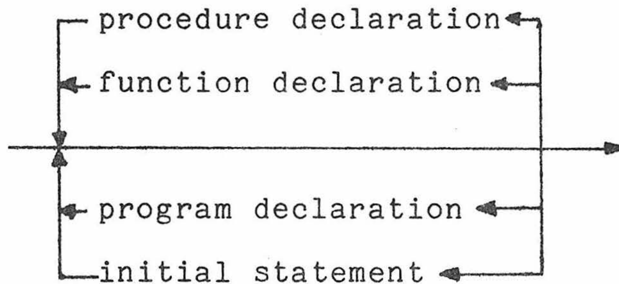
12. component type



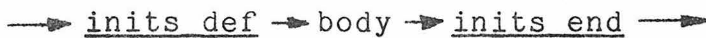
13. variable declarations



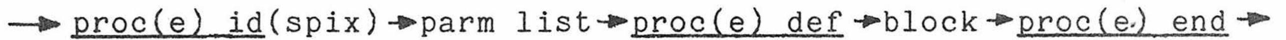
15. routine declarations



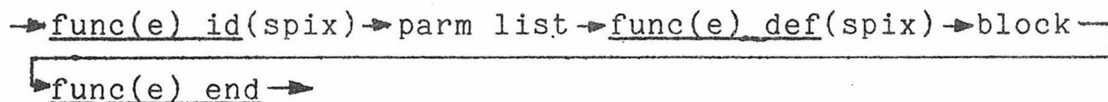
15.1 initial statement



16. procedure declaration



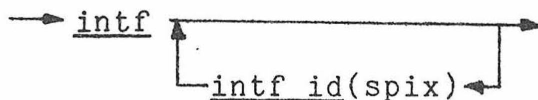
17. function declaration



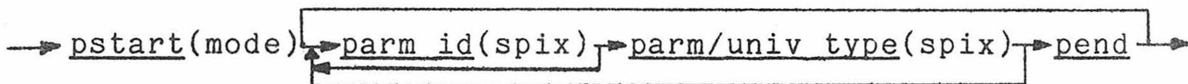
18. program declaration

→ prog id(spix) → parm list → interface → prog def →

18.1 interface



19. parm list

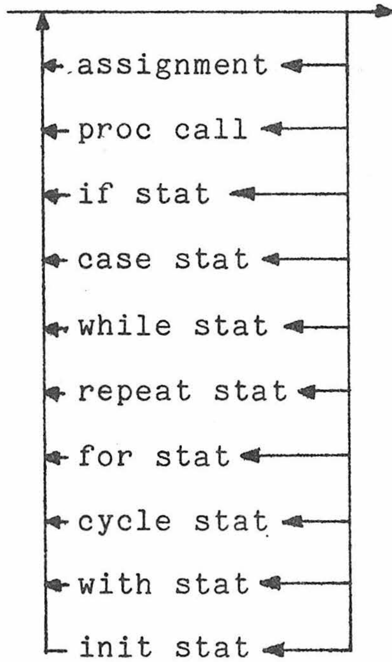


mode is any of: class mode monitor mode process mode proc mode proce mode func mode funce mode program mode.

20. body

→ body → stat → body end →

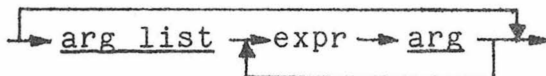
22. stat



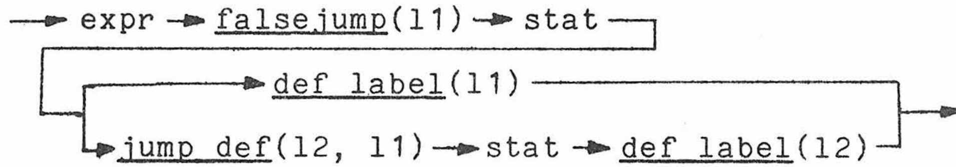
23. assignment

24. proc call

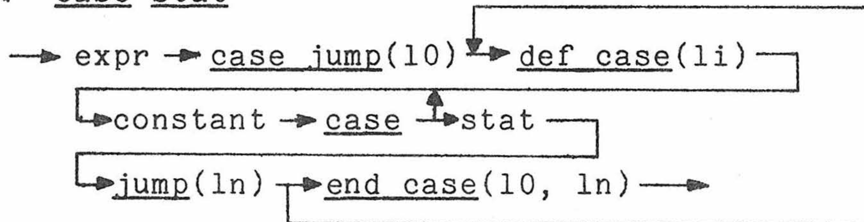
25. arg list



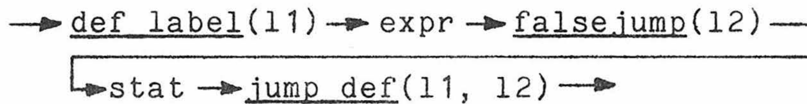
27. <u>if stat</u>



28. <u>case stat</u>

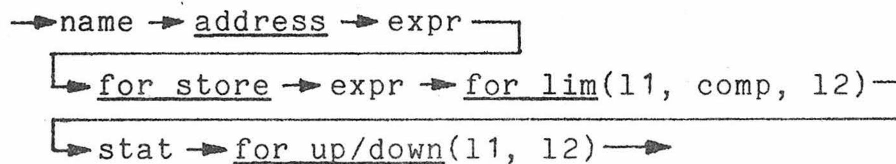


29. <u>while stat</u>

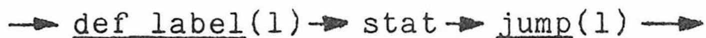


30. <u>repeat stat</u>

31. <u>for stat</u>



32. <u>cycle stat</u>



33. with stat

→ with var → name → with temp → stat → with →

34. init stat

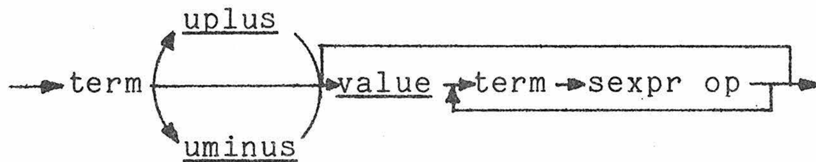
→ name → init name → arg list → init →

35. expr

→ sexpr → value → sexpr — expr op →

expr op: lt eq gt le ne ge in

36. sexpr



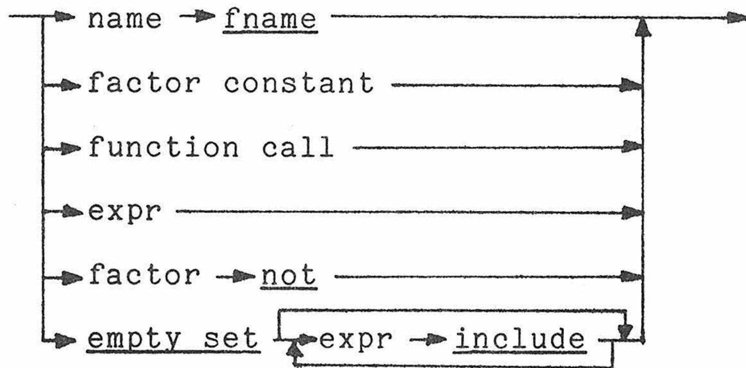
sexpr op: plus minus or

37. term

→ factor → value → factor → term op →

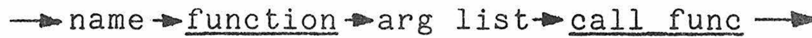
term op: star slash div mod and

38. <u>factor</u>

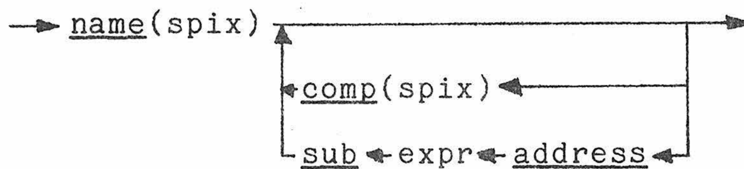


<u>factor constant:</u> <u>constant</u> with an 'f' prefixed to all terminal symbols.

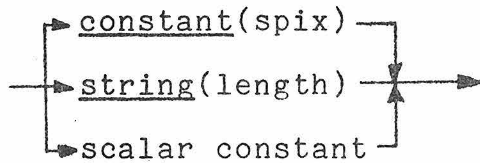
39. <u>function call</u>



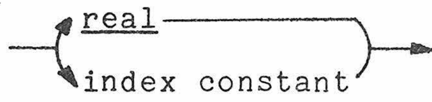
40. <u>name</u>



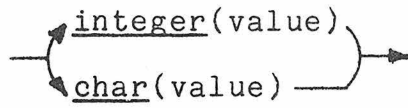
41. <u>constant</u>



44. <u>scalar</u> <u>constant</u>



47. <u>index</u> <u>constant</u>



Pass 4 - input syntax description

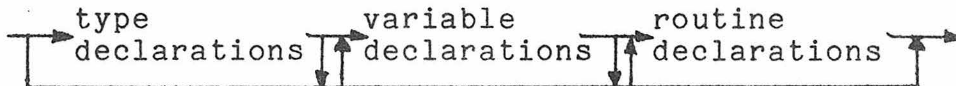
1. program

→ component type → eom →

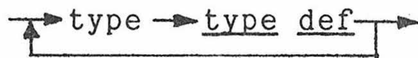
2. block

→ declarations → body →

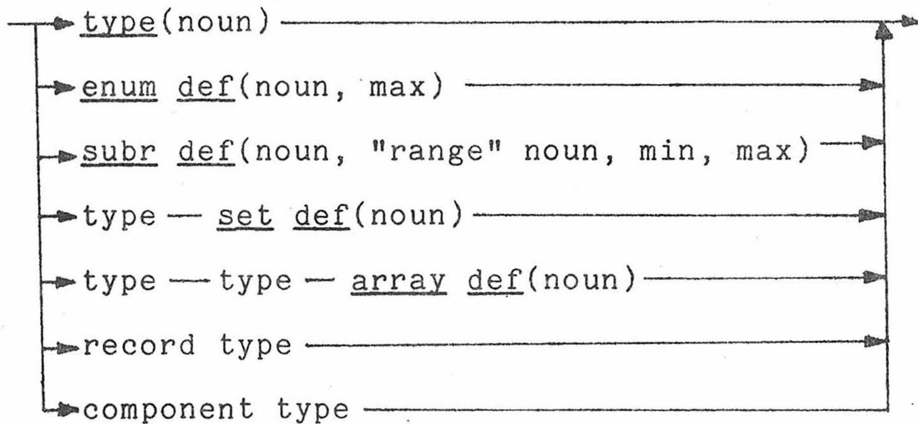
3. declarations



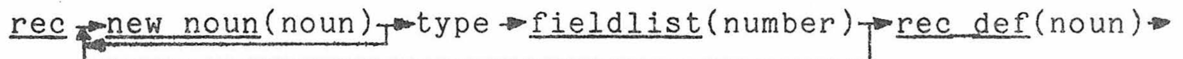
5. type declarations



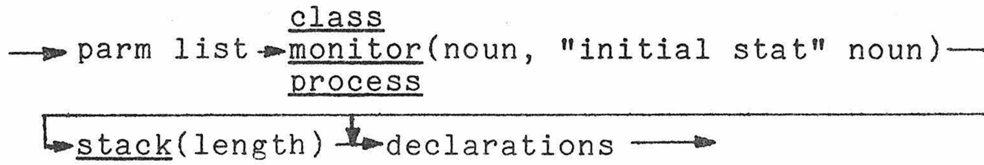
6. type



11. record type

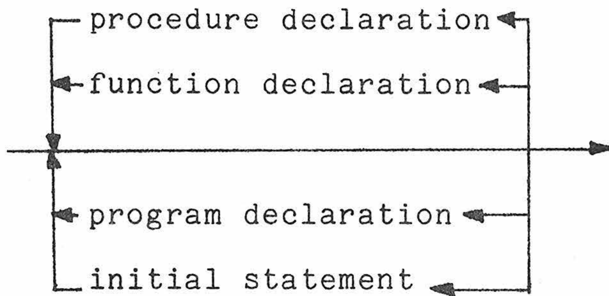


12. component type

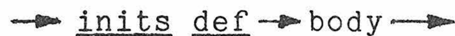


13. variable declarations

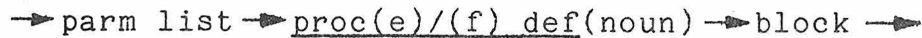
15. routine declarations



15.1 initial statement

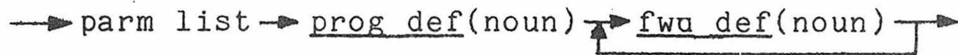


16. procedure declaration

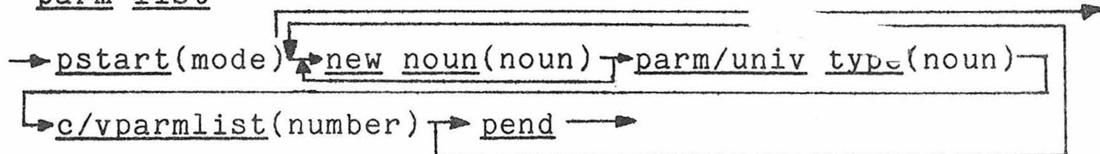


17. function declaration

18. program declaration



19. parm list

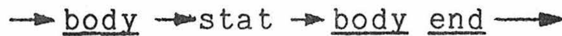


mode: class mode monitor mode process mode

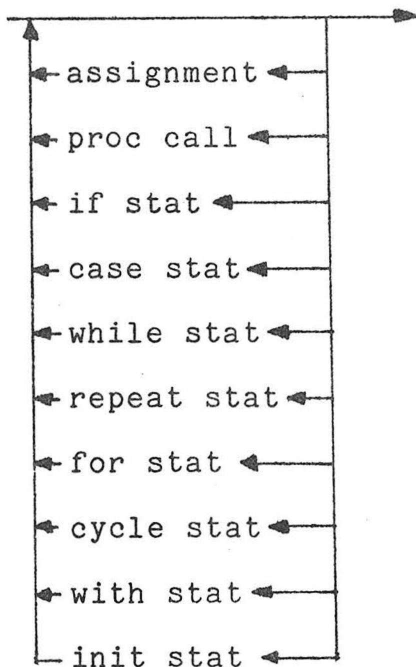
proc mode proce mode func mode funce mode

program mode.

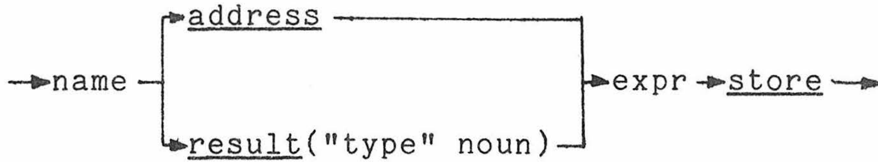
20. body



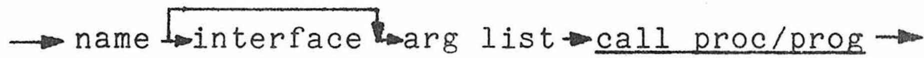
22. stat



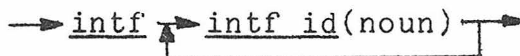
23. assignment



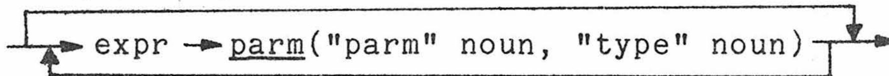
24. proc call



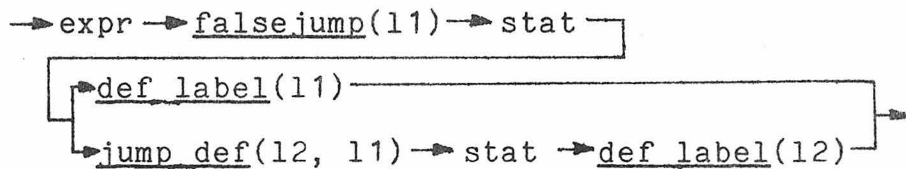
24.1 interface



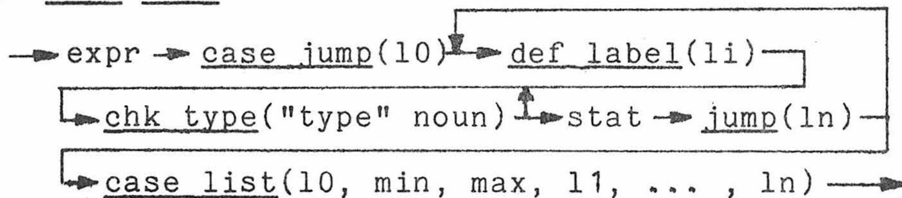
25. arg list



27. if stat



28. case stat



29. <u>while</u> <u>stat</u>

<u>def label</u>(l1) → expr → <u>falsejump</u>(l2) → stat → <u>jump def</u>(l1, l2) →

30. <u>repeat</u> <u>stat</u>

→ <u>def label</u>(l) → stat → expr → <u>falsejump</u>(l) →

31. <u>for</u> <u>stat</u>

→ name → <u>address</u> → expr →
→ <u>for store</u> → expr → <u>for lim</u>(l1, comp, l2) →
→ stat → <u>for up/down</u>(l1, l2) →

32. <u>cycle</u> <u>stat</u>

→ <u>def label</u>(l) → stat → <u>jump</u>(l) →

33. <u>with</u> <u>stat</u>

→ <u>with var</u> → name → <u>with temp</u>(noun) → stat → <u>with</u> →

34. <u>init</u> <u>stat</u>

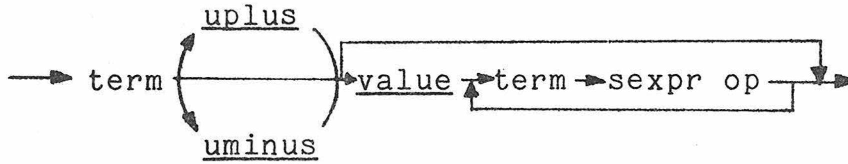
→ name → arg list → <u>init</u> →

35. <u>expr</u>

→ sexpr ⌐→ <u>value</u> → sexpr → expr op ↓→

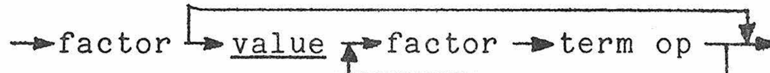
<u>expr op:</u> <u>lt</u> <u>eq</u> <u>gt</u> <u>le</u> <u>ne</u> <u>ge</u> <u>in</u>

36. sexpr



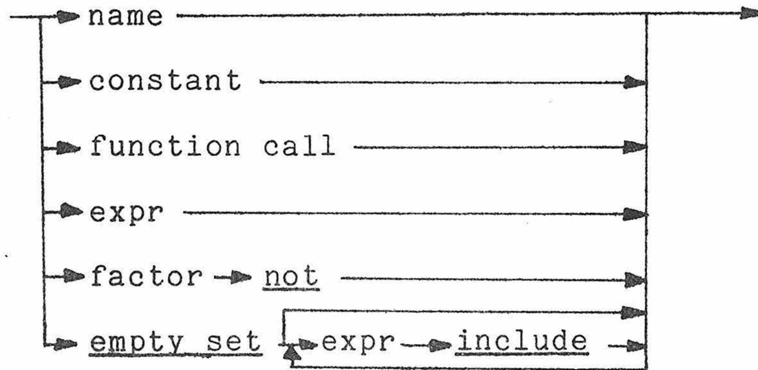
sexpr op: plus minus or

37. term

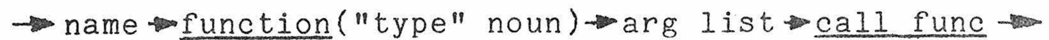


term op: star slash div mod and

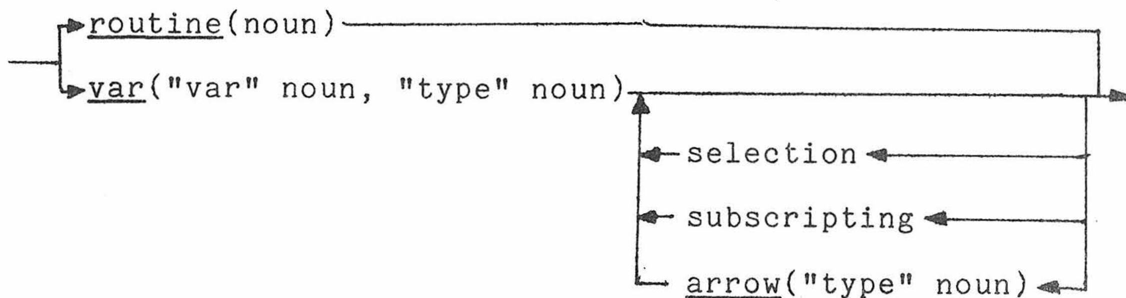
38. factor



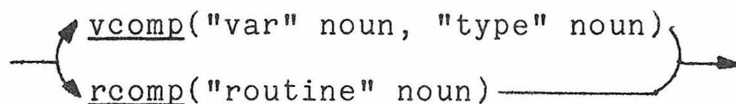
39. function call



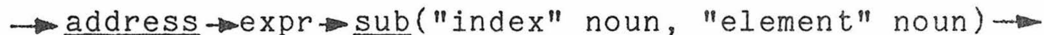
40. name



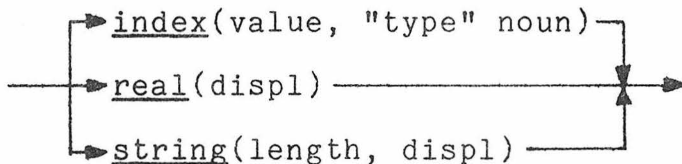
40.1 selection



40.2 subscripting



41. constant



Pass 5 - input syntax description

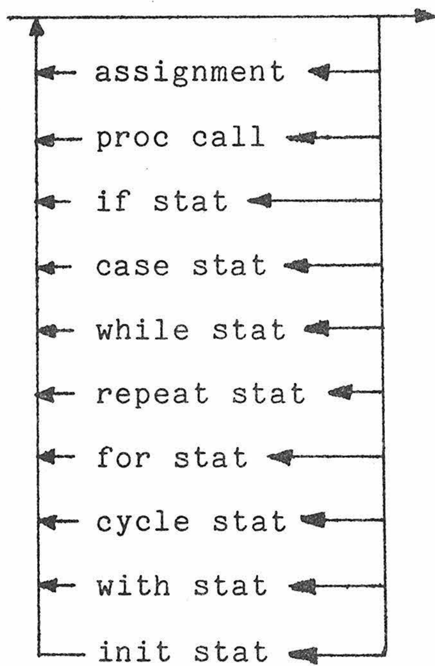
1. program

→ body → eom(varlength) →

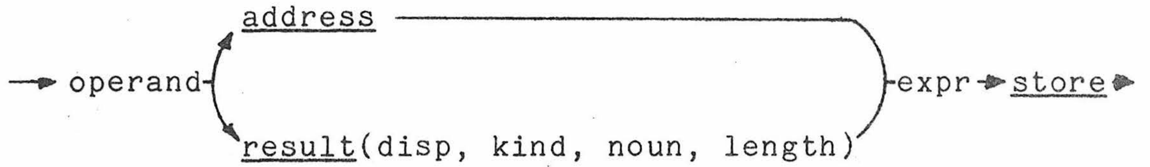
20. body

body(mode, label, parm length, var length, stack length) → stat → body end →

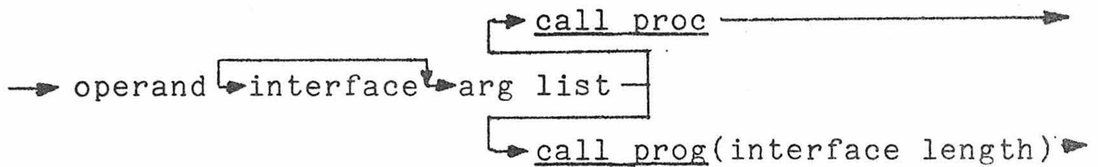
22. stat



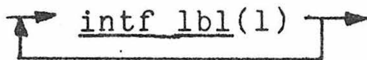
23. assignment



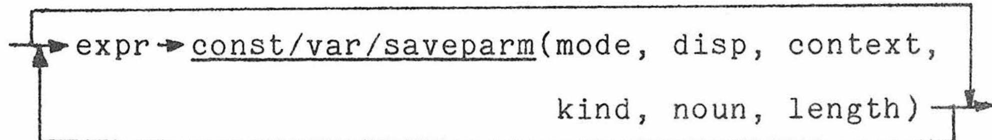
24. proc call



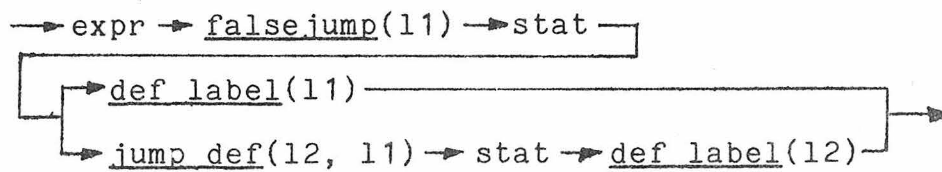
24.1 interface



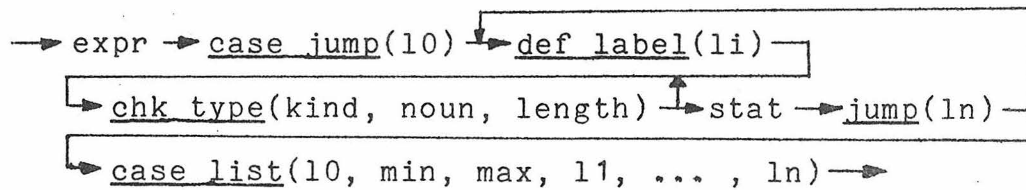
25. arg list



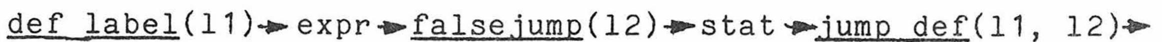
if stat



28. case stat

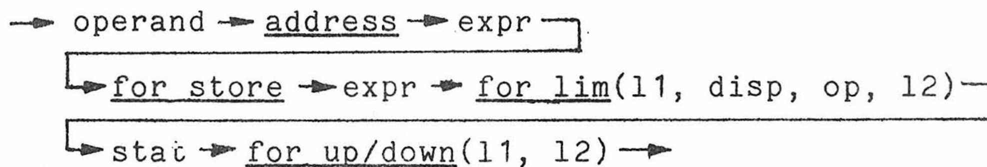


29. while stat



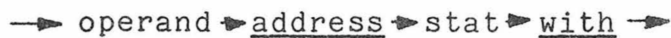
30. repeat stat

31. for stat

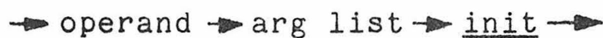


32. cycle stat

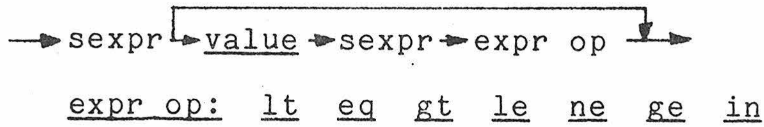
33. with stat



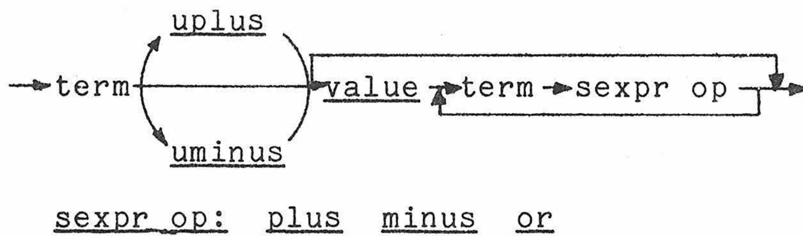
34. init stat



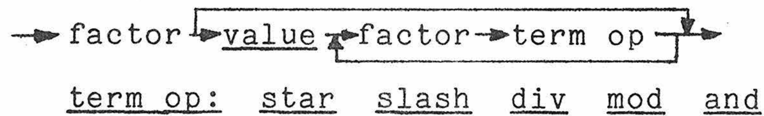
35. expr



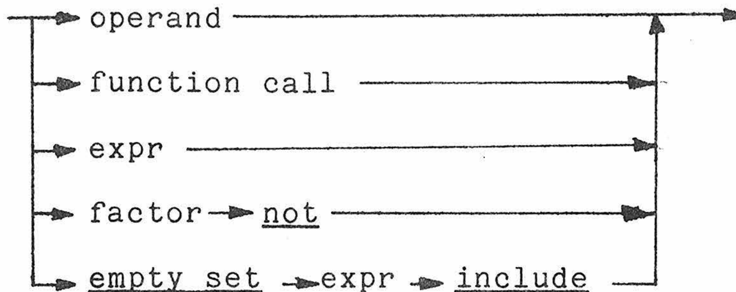
36. sexpr



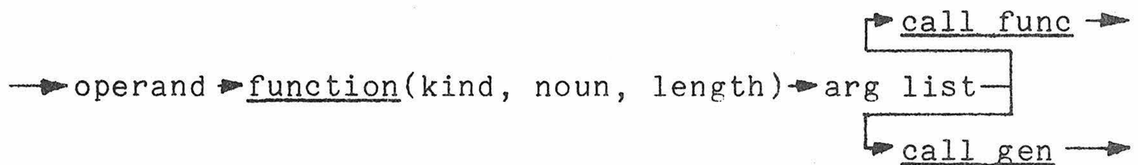
37. term



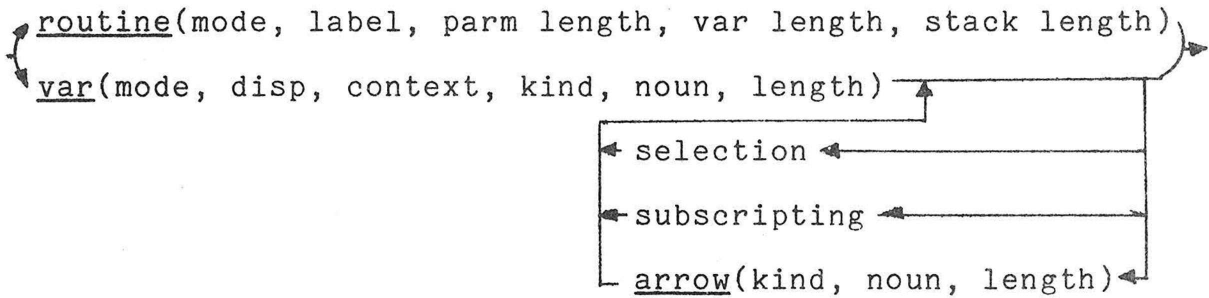
38. factor



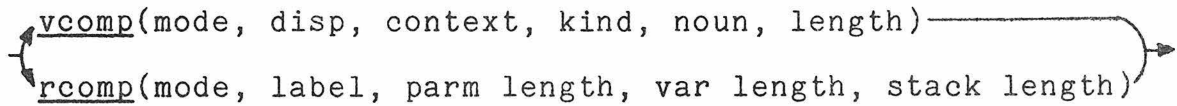
39. function call



40. <u>operand</u>



40.1 <u>selection</u>



40.2 <u>subscripting</u>

→ <u>address</u> → expr → <u>sub</u>(min, max, length,
"index" kind, noun, length,
"element" kind, noun, length) →

Pass 6 - input syntax description

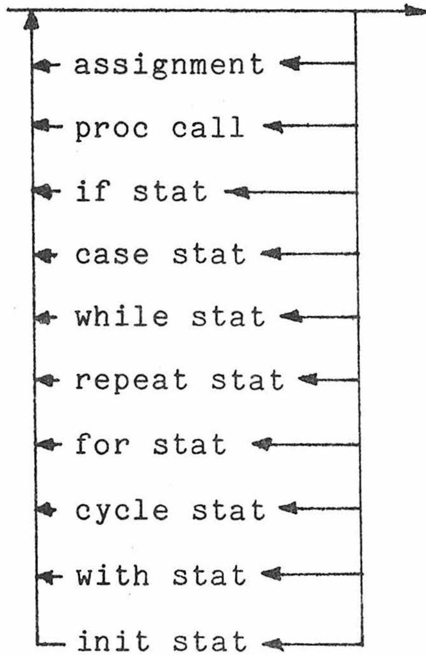
1. program

→ jump(1) → body → eom(var length) →

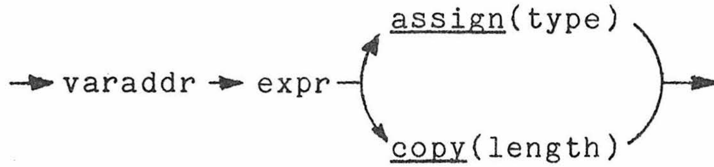
20. body

enter(mode, label, parm length, var length, temp length) ┐
└→ stat → return(mode) →

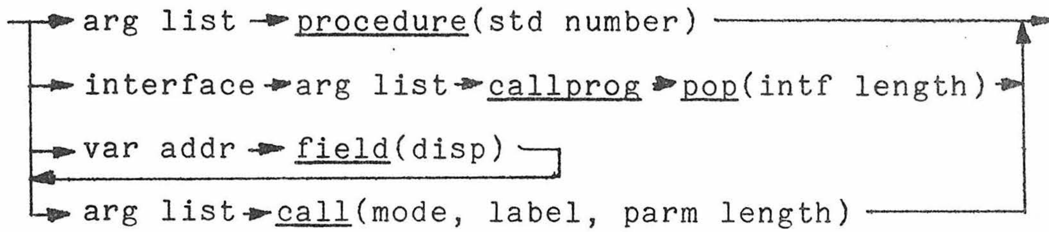
22. stat



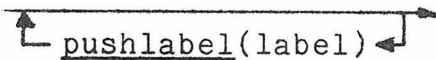
23. assignment



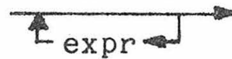
24. proc call



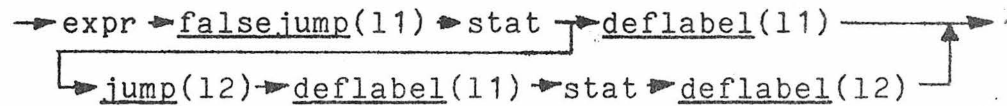
24.1 interface



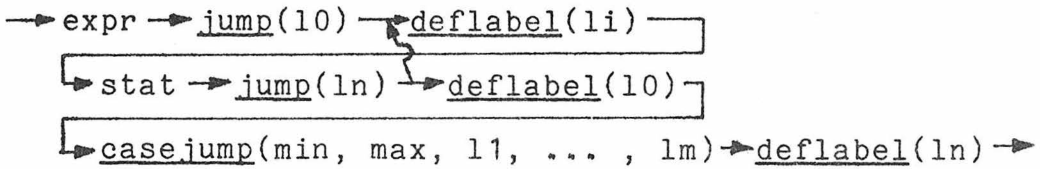
25. arg list



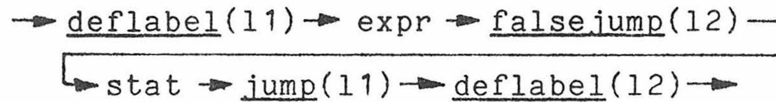
27. if stat



28. case stat

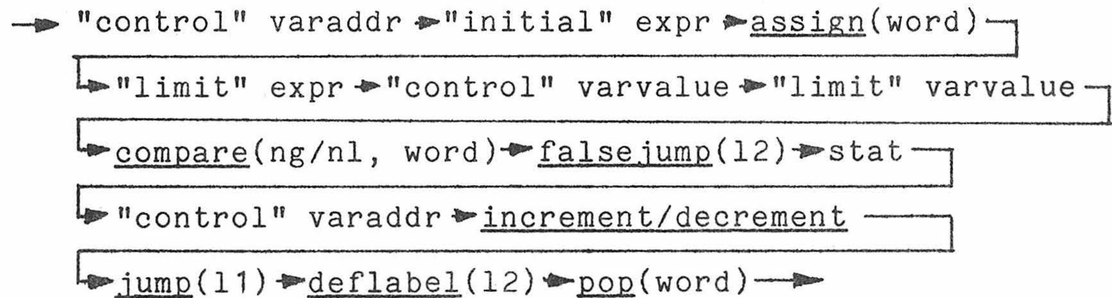


29. while stat



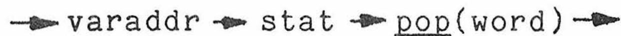
30. repeat stat

31. for stat

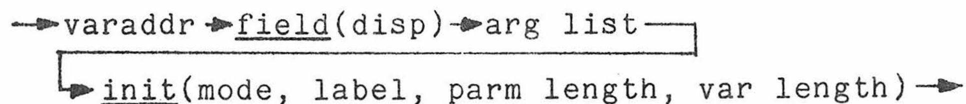


32. cycle stat

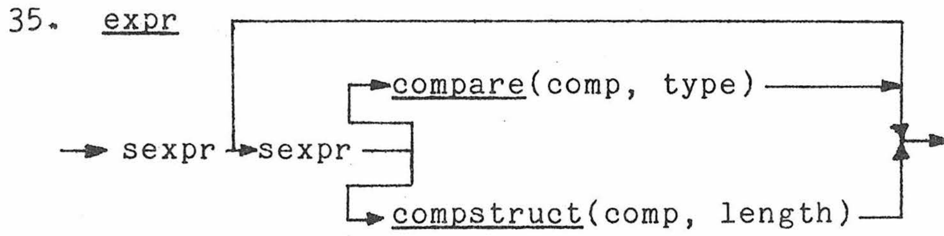
33. with stat



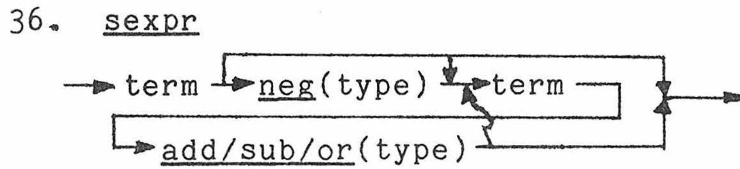
34. init stat



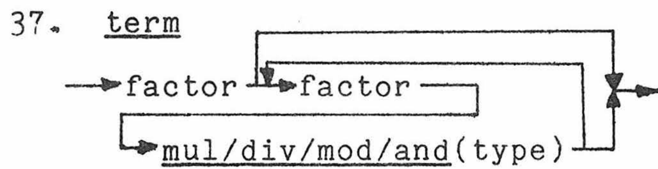
35. expr



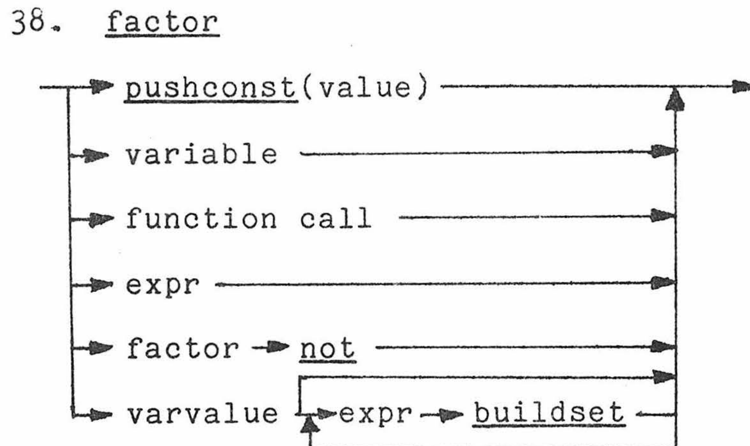
36. sexpr



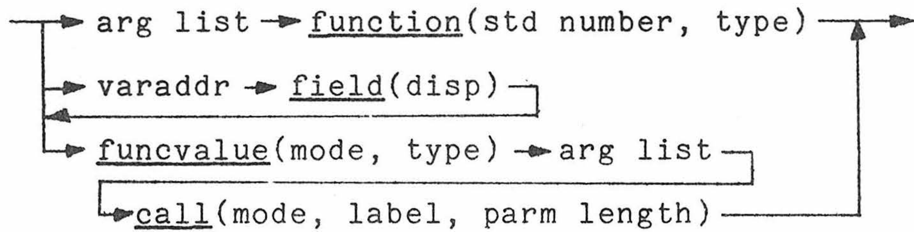
37. term



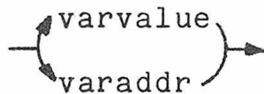
38. factor



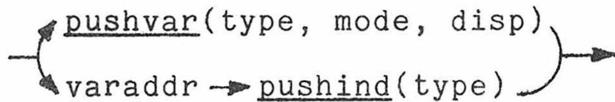
39. function call



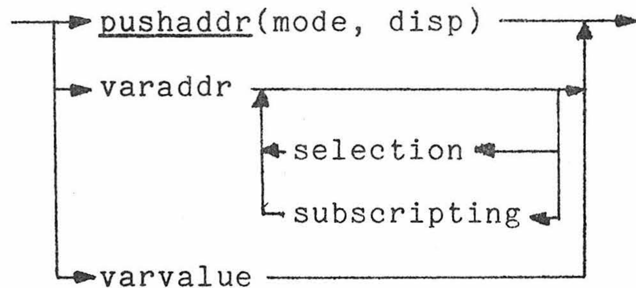
40. variable



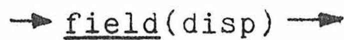
40.1 varvalue



40.2 varaddr



40.3 selection



40.4 <u>subscripting</u>

⟶ expr ⟶ <u>index</u>(min, max, length) ⟶

Pass 7 - input syntax description

1. program

→ jump(loc, label) → body → eom →

20. body

→ enter → stat → return →

20.1 enter

enter(block, pop length, line, var length)

enterprog(pop length, line, block, var length)

enterproc(block, pop length, line, var length)

enterclas(block, poplength + wordlength, line, var length)

entermon(block, poplength + wordlength, line, var length)

beginproc(line)

beginclas(block, fivewords, line, 0)

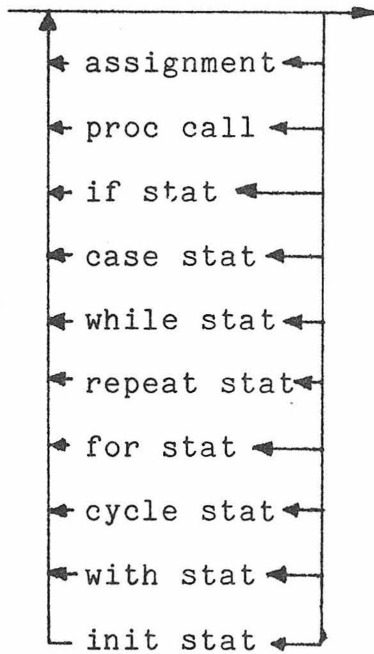
beginmon(block, fivewords, line, 0)

20.2 return

exit exitprog exitproc exitclass

exitmon endproc endclass endmon

22. stat



23. assignment

→ varaddr → expr → assign →

23.1 assign

copybyte

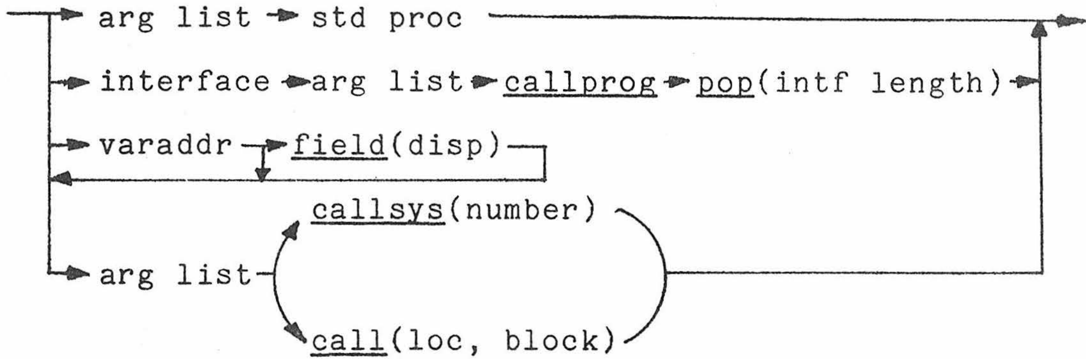
copyword

copyreal

copyset

copystruct(length in words)

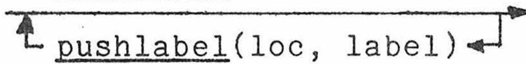
24. proc call



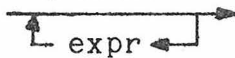
24.1 std proc

delay continue io start stop setheap wait

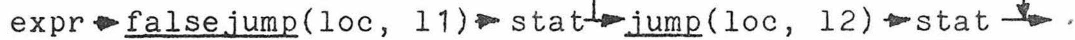
24.2 interface



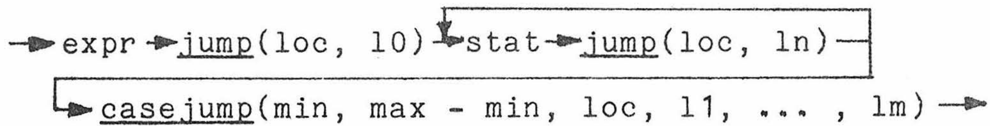
25. arg list



27. if stat



28. case stat



29. <u>while</u> <u>stat</u>

expr → <u>falsejump</u>(loc, l2) → stat → <u>jump</u>(loc, l1) →

30. <u>repeat</u> <u>stat</u>

→ stat → expr → <u>falsejump</u>(loc, l) →

31. <u>for</u> <u>stat</u>

→ "control" varaddr → "initial" expr → <u>copyword</u> ┐

└→ "limit" expr → "control" varvalue → "limit" varvalue ┐

└→ <u>ngword/nlword</u> → <u>falsejump</u>(loc, l2) → stat ┐

└→ "control" varaddr → <u>incrword/decrword</u> ┐

└→ <u>jump</u>(loc, l1) — <u>pop</u>(word) →

32. <u>cycle</u> <u>stat</u>

→ stat → <u>jump</u>(loc, l) →

33. <u>with</u> <u>stat</u>

→ varaddr → stat → <u>pop</u>(word) →

34. <u>init</u> <u>stat</u>

→ varaddr └→ <u>field</u>(disp) ┘→ arg list → init →

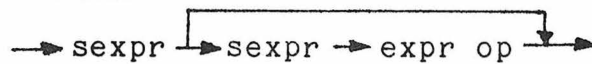
34.1 <u>init</u>

<u>initproc</u>(parm length, var length, block, loc, block)

<u>initclass</u>(parm length, loc, block)

<u>initmon</u>(parm length, loc, block)

35. <u>expr</u>



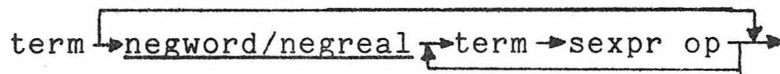
<u>expr op:</u> <u>lsword</u> <u>eqword</u> <u>grword</u> <u>nlword</u>

<u>neword</u> <u>ngword</u> <u>lsreal</u> <u>eqreal</u> <u>grreal</u> <u>nlreal</u>

<u>nereal</u> <u>ngreal</u> <u>eqset</u> <u>nlset</u> <u>neset</u>

<u>ngset</u> <u>inset</u>

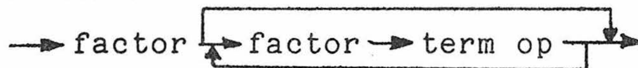
36. <u>sexpr</u>



<u>sexpr op:</u> <u>addword</u> <u>addreal</u> <u>subword</u>

<u>subreal</u> <u>subset</u> <u>orword</u> <u>orset</u>

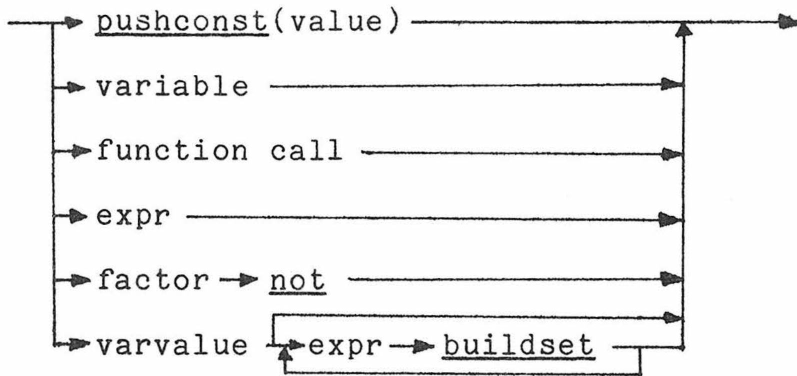
37. <u>term</u>



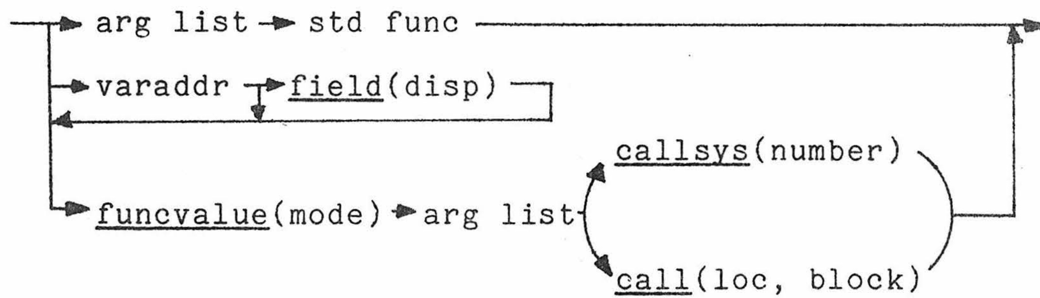
<u>term op:</u> <u>mulword</u> <u>mulreal</u> <u>divword</u>

<u>divreal</u> <u>modword</u> <u>andword</u> <u>andset</u>

38. factor



39. function call

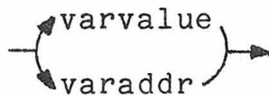


39.1 std func

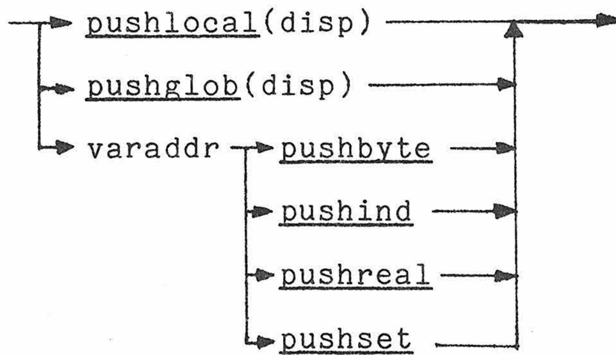
truncreal absword absreal succword predword

convword empty attribute realtime

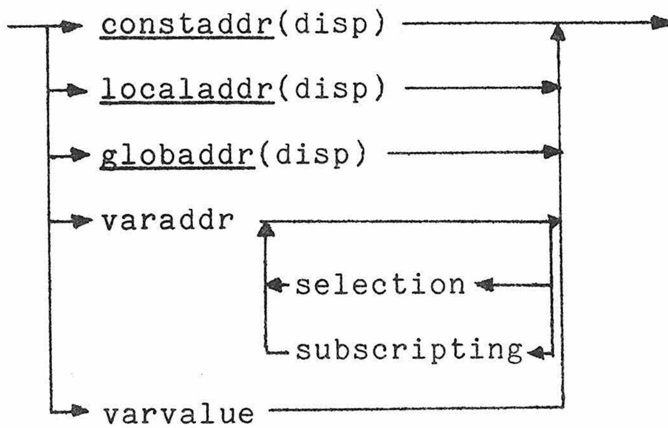
40. variable



40.1 varvalue



40.2 varaddr



40.3 selection

→ field(disp) →

40.4 subscripting

→ expr → index(min, dimension, length) →

Final code - syntax description

1. program

→ (prog length, code length, stack length, var length) → jump(disp) → body → (constants) →

20. body

→ enter → stat → return →

20.1 enter

enter(stack length, pop length, line, var length)

enterprog(pop length, line, stack length, var length)

enterproc(stack length, pop length, line, var length)

enterclas(stack length, pop length + word length,
line, var length)

entermon(stack length, pop length + word length,
line, var length)

beginproc(line)

beginclas(stack length, five words, line, 0)

beginmon(stack length, five words, line, 0)

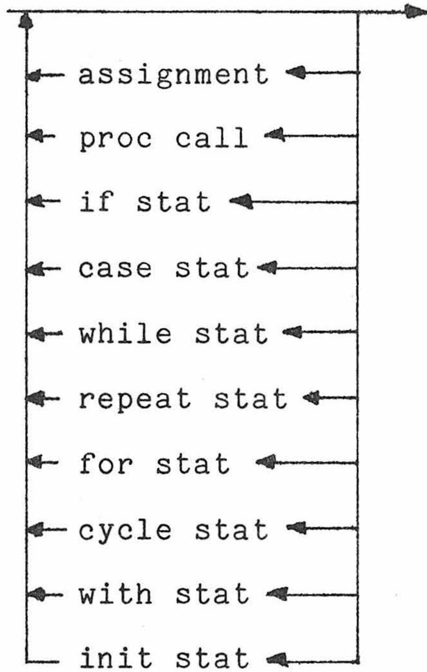
"pop length is parm length plus four words"

20.2 return

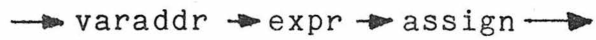
exit exitprog exitproc exitclass

exitmon endproc endclass endmon

22. stat



23. assignment



23.1 assign

copybyte

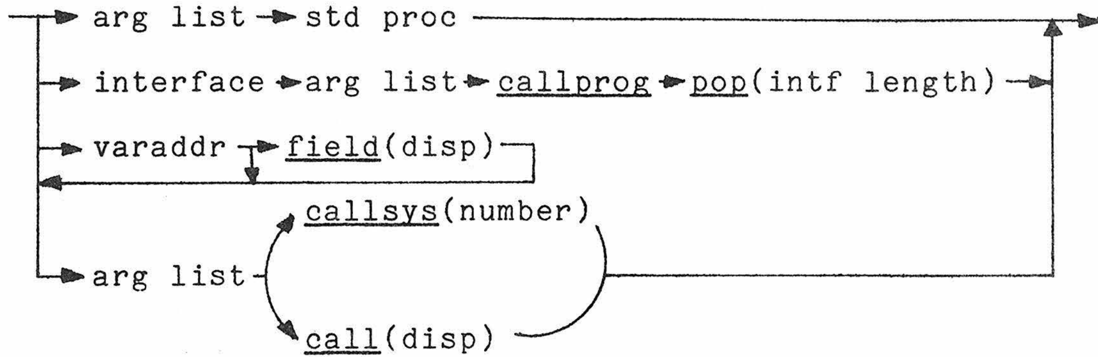
copyword

copyreal

copyset

copystruct(length in words)

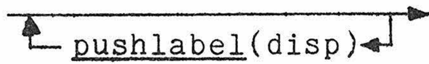
24. proc call



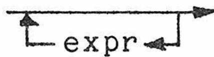
24.1 std proc

delay continue io start stop setheap wait

24.2 interface



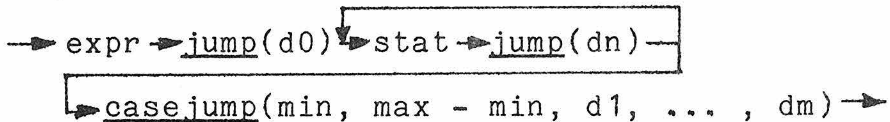
25. arg list



27. if stat

expr → falsejump(d1) → stat → jump(d2) → stat →

28. case stat



29. while stat

→ expr → falsejump(d2) → stat → jump(d1) →

30. repeat stat

→ stat → expr → falsejump(d) →

31. for stat

→ "control" varaddr → "initial" expr → copyword →
→ "limit" expr → "control" varvalue → "limit" varvalue →
→ ngword/nlword → falsejump(d2) → stat →
→ "control" varaddr → incrword/decrword → jump(d1) →
→ pop(word) →

32. cycle stat

→ stat → jump(d) →

33. with stat

→ varaddr → stat → pop(word) →

34. init stat

varaddr → field(disp) → arg list → init →

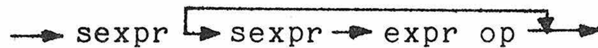
34.1 init

initproc(parm length, var length, stack length, disp)

initclass(parm length, disp)

initmon(parm length, disp)

35. expr



expr op: lsword eqword grword nlword

neword ngword lsreal eqreal grreal

nlreal nereal ngreal eqset nlset neset

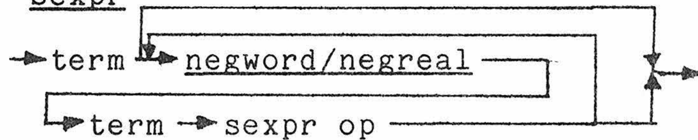
ngset inset lsstruct eqstruct grstruct

nlstruct nestruct ngstruct.

"struct operators take the struct length in

words as an argument"

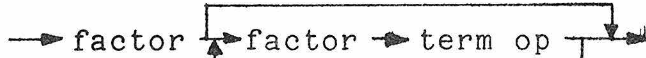
36. sexpr



sexpr op: addword addreal subword

subreal subset orword orset.

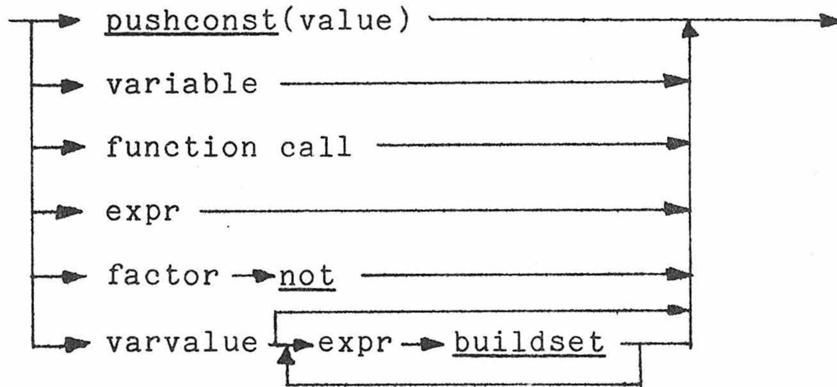
37. term



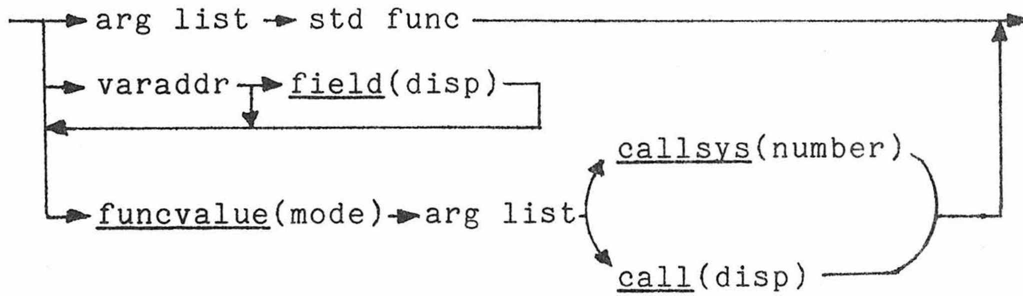
term op: mulword mulreal divword

divreal modword andword andset.

38. factor



39. function call

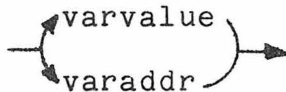


39.1 std func

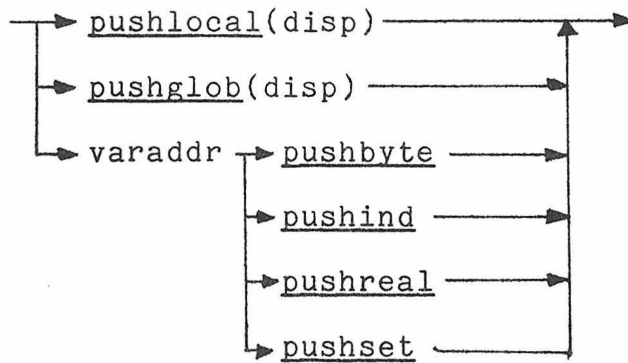
truncreal absword absreal succword predword

convword empty attribute realtime

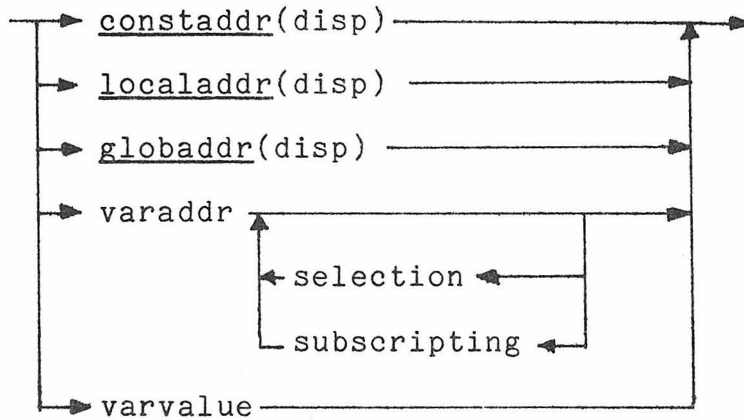
40. variable



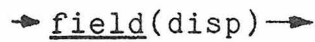
40.1 varvalue



40.2 varaddr



40.3 selection



40.4 subscripting