Information Technology Experiment and Exercise II
System Program

Index

Chapter 1 Designing and Creating Language Processor

1.1 Introduction

The aim of this experiment is to create a language processor by using a simple procedural programming language, the Kyoto University Programming Language (KPL). This language processor presupposes a carefully structured calculator called "stack calculator", which I will explain later, and translates a source program written in KPL into object codes applicable to the calculator. KPL is a programming language of the same sort as ALGOL and PASCAL. It is especially close to PASCAL and is like a simplified form of PASCAL. With this experiment, I hope you will learn under what design concept such a programming language is designed, and how closely the idea of stack calculator is related to language processor, by finding out many interesting things by yourself. Although it is not easy to expand this processor to the normal compiler, it is still possible for you to learn the basic techniques to design programming languages and to create language processor in practice with this experiment. [1]

The processor in this experiment is a simple one as a language processor. But it has a considerable "complexity" and "size" as a real program. When developing such a program, it is important to avoid programming without planning and to systematically conduct 1) program designing, 2) error correction (debug), 3) documentation, and 4) changing and saving. It is another aim of this experiment to conduct a systematically and structurally well-regulated program development.

1.2 Functions of language processor

"Language processor" denotes software which translates a program written in a certain programming language into an executable form, or even itself undertakes its execution. It is classified variously according to its object and method of processing. For a typical example, the program which translates a source program expressed in programming language L (source language) into semantically equivalent object codes (program written in the object language) is called "L compiler" or "translator". Especially, a translator which has assembly language as its source language and machine language as its object language is known as "assembler". A program which input syntax description and semantics description of programming language L (ideally, the arbitrary chosen one) and which output L compiler, is called "compiler-compiler" or "compiler generator". The one which analyses source program each time by its basic structuring unit (generally, 1 statement) and executes it, is called "interpreter".

Language processor usually possesses a functional structure illustrated in Figure 1.1. A source program is analyzed as a "string" on word analysis part and divided into "words". That is, it is divided from a string into several parts including "bound symbols", "identifier", and "constant". It, in turn, finds out syntax and semantics of statements by analyzing words and passes them to "code generator". The function of code generator is to generate object codes which correspond to the source program.

As an actual program, in some cases, each three components mentioned above forms independent phases, and in other cases, those parts work as simple sub-programs constituting a single program as a whole.

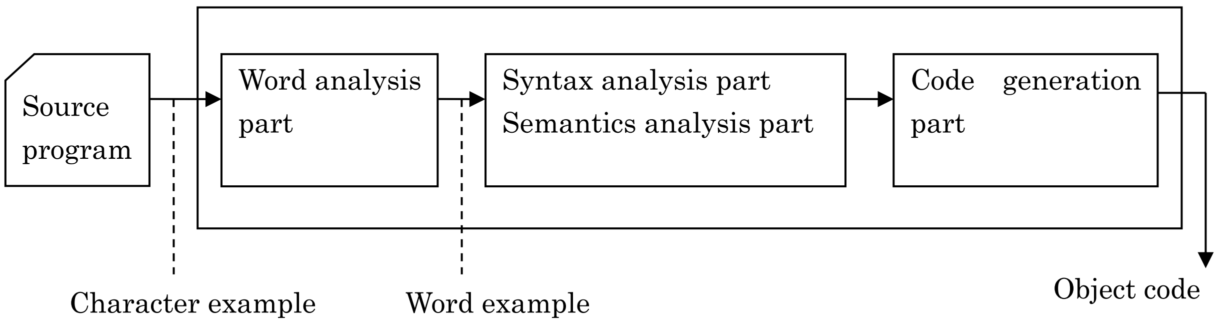


Figure 1.1 Function of language processor

1.3 Programming Language KPL

In order to create a language processor, it is necessary to understand the language of the object. Thus, you are expected to learn properly the "syntax" and "semantics" of the programming language KPL in this section.

Rather than getting into details of the language right from the beginning, it may be better to start with a bit intuitive introduction. I suppose you won't feel uncomfortable with KPL as you are already familiar with some other programming languages, or you will even be so interested and enjoy executing a processor written in this language. Here is an example of programming by KPL.

[Example.1.1] Calculation of factorial

```
PROGRAM   EXAMPLE1; (* FACTORIAL *)
     VAR   N:INTEGER;

     FUNCTION   F(N;INTEGER):INTEGER;
          BEGIN
               IF   N=0   THEN   F:=1   ELSE   F:=N*F(N-1)
          END;

     BEGIN
          FOR   N:=1   TO   7   DO
```

```
            BEGIN   CALL   WRITELN;   CALL   WRITEI(F(N))   END
        END.   (* EXAMPLE1 *)
```

Output
```
   1
   2
   6
  24
 120
 720
5040
```

Line 1 Program name is “EXAMPLE1”. Inside (* *) is a comment.
Line 2 Variable N is an integer.
Line 3 Definition of Function (F) which is an integer. Parameter (N) is also an integer.
Lines 4-6 Block structure. Recursively definable. Be careful to = and :=.
Lines 7-10 Execution statements.
Line 9 Input and output is executed with a basic procedure. WRITELN is break. WRITEI is integer output.
Line 10 Don’t forget a period after END.

A program has to start with a word “PROGRAM”. A program consists of several lines, and each line characters and spaces. You can start writing from anywhere in a line, and there can be blank lines. Words like PROGRAM are keywords of KPL. It is necessary for KPL to declare all identifiers like variables in advance.

(Exercise 1.1) What is the “recursive program”? Analyze how the program in the Example.1.1 executes calculation.

(Exercise 1.2) What are merits of declaring all identifiers in advance? Has declaring ever become a limitation for functions of KPL defined in this chapter? (cf. “Forward declaration” in PASCAL)

[Example.1.2] A calculation program to solve the puzzle “Tower of Hanoi”
Read the program below. READC and WRITEC are basic procedures of one-character input-output.

```
PROGRAM   EXAMPLE2;   (* TOWER OF HANOI *)
    VAR   I:INTEGER;   N:INTEGER;   P:INTEGER;   Q:INTEGER;
    C:CHAR;
```

```
        PROCEDURE   HANOI(N:INTEGER;   S:INTEGER;   Z:INTEGER);
            BEGIN
                IF   N┐=0   THEN
                   BEGIN
                       CALL   HANOI(N-1,S,6-S-Z);
                       I:=I+1;   CALL   WRITELN;
                       CALL   WRITEI(I);   CALL   WRITE(N);
                       CALL   WRITE(S);   CALL   WRITE(Z);
                       CALL   HANOI(N-1,6-S-Z,Z)
                   END
            END;   (*END OF HANOI*)

        BEGIN
            FOR   N;=1   TO   4   DO
                BEGIN
                    FOR   I:=1   TO   4   DO   CALL   WRITEC(‘   ‘);
                    CALL   READC(C);   CALL   WRITEC(C)
                END;
            P:=1;   Q:=2;
            FOR   N:=2   TO   4   DO
                BEGIN   I:=0;   CALL   HANOI(N,P,Q);   CALL   WRITELN
    END
        END.   (* EXAMPLE2 *)
```

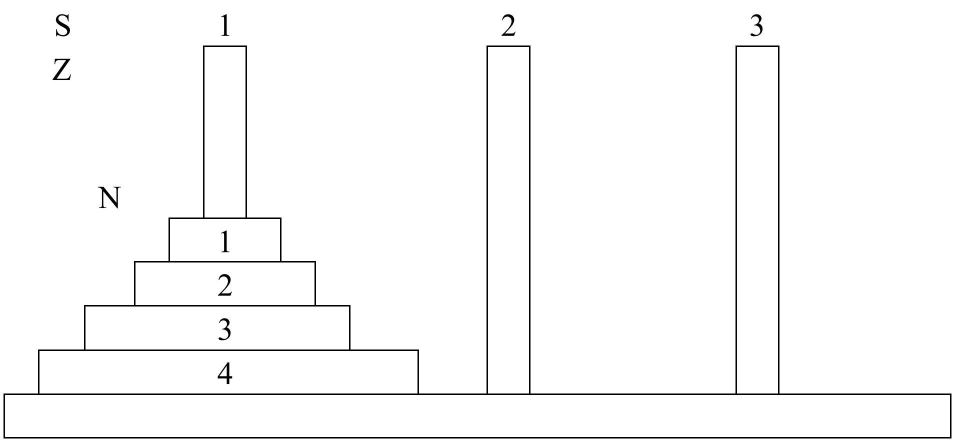
Input data INSZ

Output

```
 I  N  S  Z
 1  1  1  3

 2  2  1  2
 3  1  3  2

 1  1  1  2
 2  2  1  3
 3  1  2  3
 4  3  1  2
 5  1  3  1
 6  2  3  2
 7  1  1  2
```



```
 1  1  1  3
 2  2  1  2
 3  1  3  2
 4  3  1  3
 5  1  2  1
 6  2  2  3
 7  1  1  3
 8  4  1  2
 9  1  3  2
10  2  3  1
11  1  2  1
12  3  3  2
13  1  1  3
14  2  1  2
15  1  3  2
```

◎In this program, solutions are determined sequentially according to the number of dishes (2, 3, or 4).
◎Procedure HANOI(N,S,Z) prints out the procedure to remove Dish 1,…, N from Bar S to Bar Z.
◎Printing results I, N, S, Z indicates to remove Dish N from Bar S to Bar Z in step I.

1.3.1 Syntax of KPL

A program written in any language is seen as a string of symbols. Such a string is usually written in a certain grammatical manner, and a string not written in an appropriate grammar does not satisfy necessary conditions of the program written in that language. Such a relationship or a structure of symbols within strings constituting program is called “syntax”, and expression of it is called “syntax description.”

Let’s describe the syntax by using a notation known as “syntax chart”, rather than describing the character of KPL like a manual for normal calculator. Syntax chart is a notation first used for syntax description of PASCAL, and it is relatively easy to understand. Syntax chart is essential for syntactic analysis with the language processor. Please read carefully.

First of all, you will decide characters for KPL like this;

digit : 0, 1, ・・・・・・・, 9
letter : A, B, ・・・・・・・, Z, #, _, ¥
special character : +, -, *, /, <, >, =, ¬, [space] , , [comma], ., :, ;, ’, (,)

For convenience, I call line 1 “digit”, line 2 “letter”, and the whole of lines 1-3 “character”.

The whole of the KPL syntax chart is illustrated in Figure.1.2. I will describe more about it below.

For the notation of KPL, you use basic symbols fixed in advance. These are;

(character, digit, letter,)

(.,.), :=, ¬=, <=, >=,
PROGRAM, CONST, TYPE, VAR, PROCEDURE, FUNCTION,
BEGIN, END, ARRAY, OF, INTEGER, CHAR, CALL,
IF, THEN, ELSE, WHILE, DO, FOR, TO

Since characters like “PROGRAM” are reserved as basic symbols, you are prohibited to use them for other purposes such as for “procedure name”.

Basic symbols are circled in the syntax chart. Other symbols are in squares.

Let’s read the syntax chart. For the first part, “program” is the part starting from a basic symbol “PROGRAM” followed by identifier, ; [semi-colon], block, and . [period].

Identifier is the one which starts from “letter” followed by only one character or arbitrary numbers of letter or digit.

“Block” is much complex. Read carefully following arrows in branches. Either way will be fine when you come across branching points, and if the branch is looping back, you can pass through the branch any number of times. But remember each arrow has a particular direction. For example, it is not possible to go back from “typedcl” to “constdcl”.

In the following section about the semantics of KPL, I will discuss how to analyze a program written with such syntax. Until you define semantics, description following the syntax is seen only as a mere lining of symbols.

If you look only at the syntax, this syntax chart is actually insufficient. For instance, you may notice in the syntax chart in Figure. 1.2 that the structure of syntax principally allows four operations of character strings. However, other languages such as PASCAL too follow the manner of syntax notation like this, and they supplement, by simplifying the syntax chart, unexpressed restrictions on syntax from the side of semantics. Many of them can be expressed only with syntax by simply changing the syntax chart. Try to think about what sort of supplements will be necessary, though I won’t explain every detail of it in the section about the semantics of KPL.

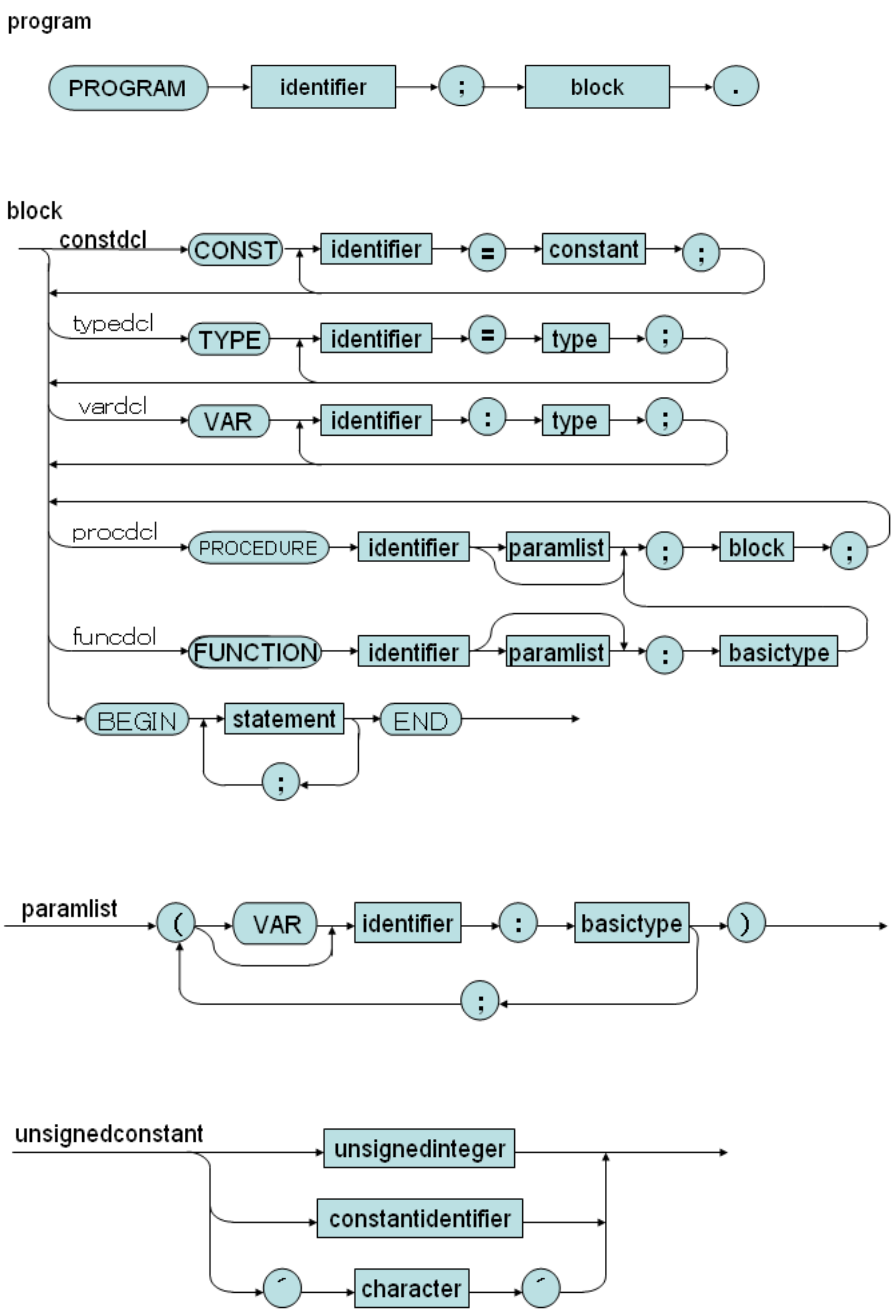


Figure 1.2 KPL syntax chart

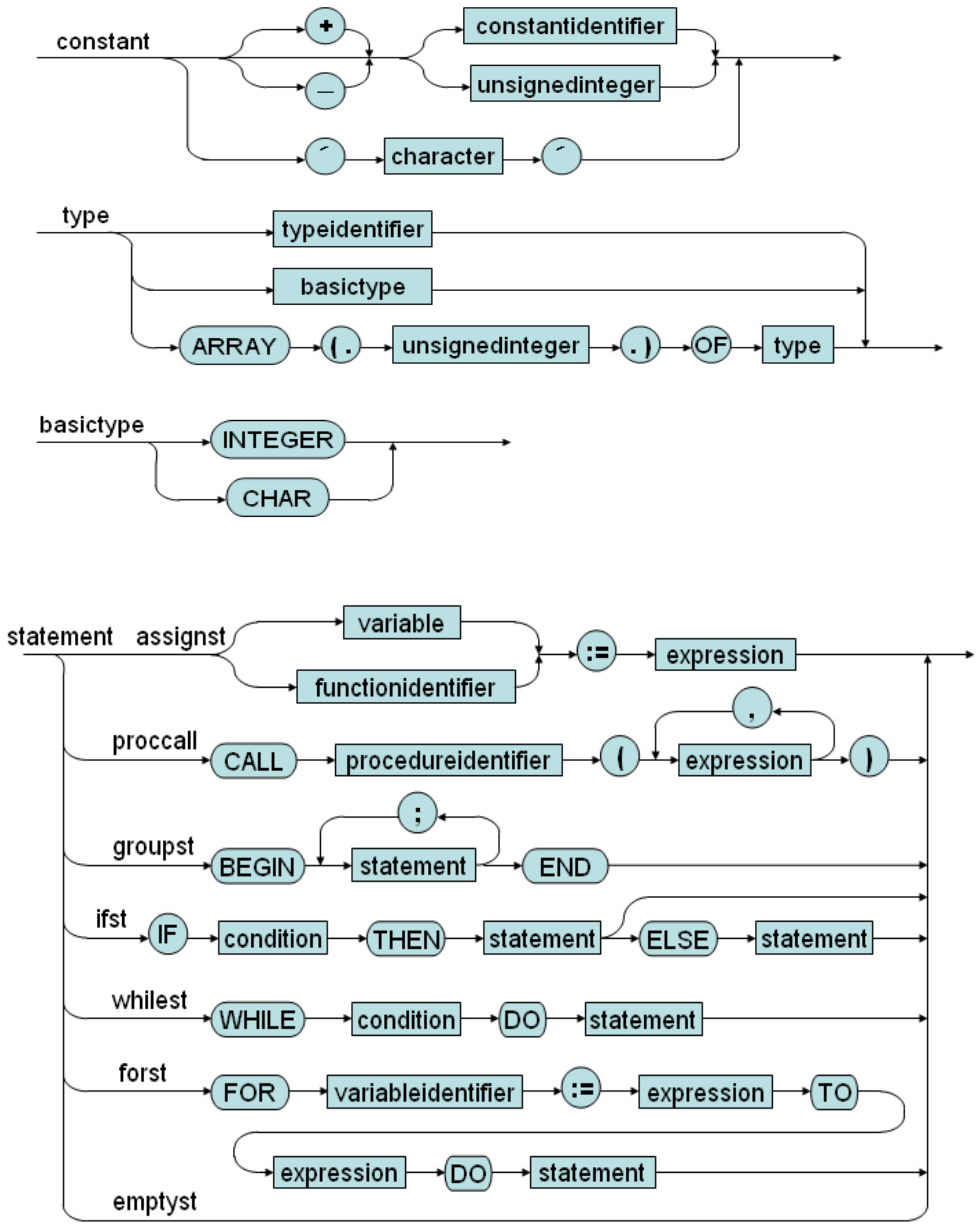


Figure 1.2 KPL syntax chart (cont.)

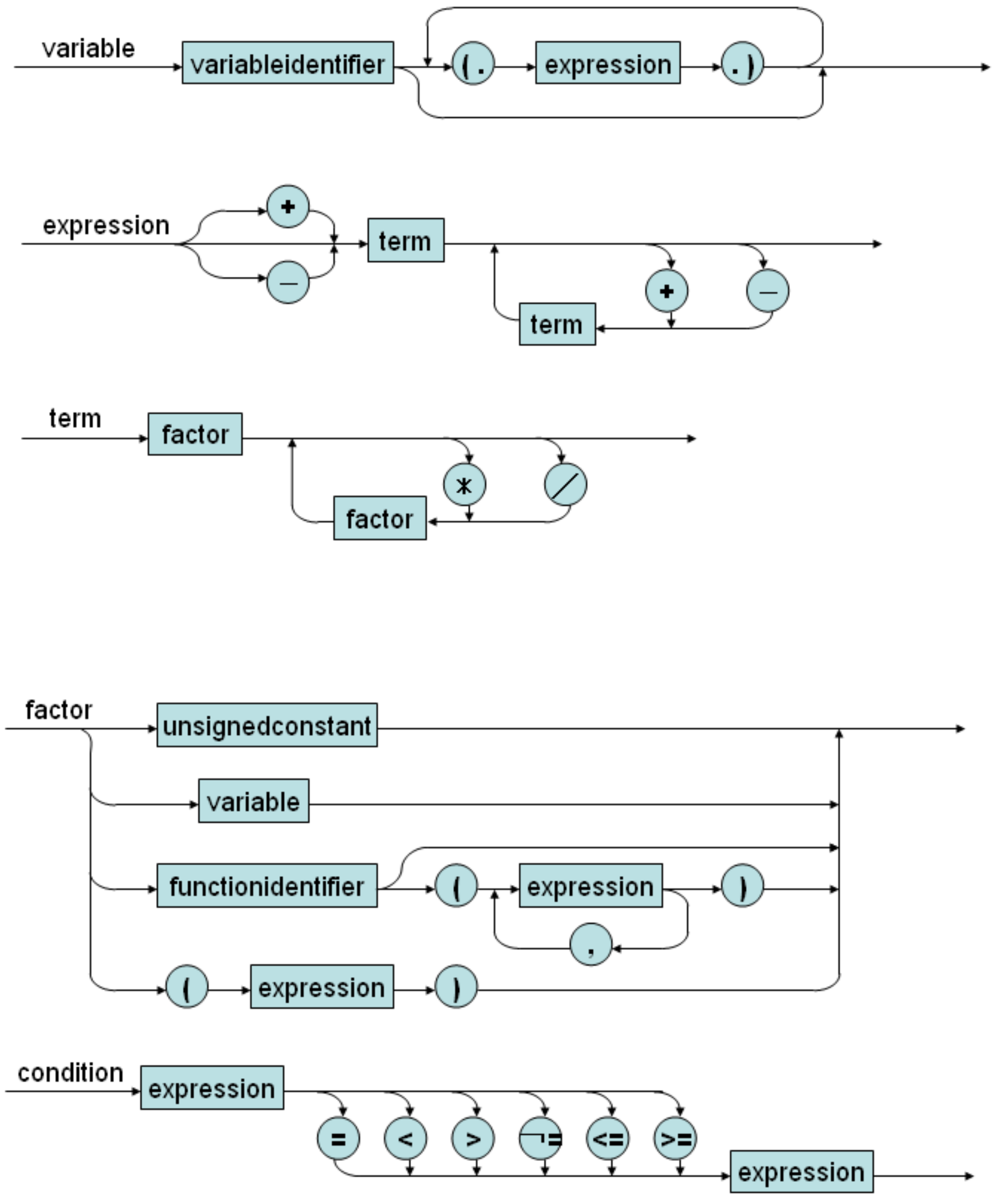


Figure 1.2 KPL syntax chart (cont.)

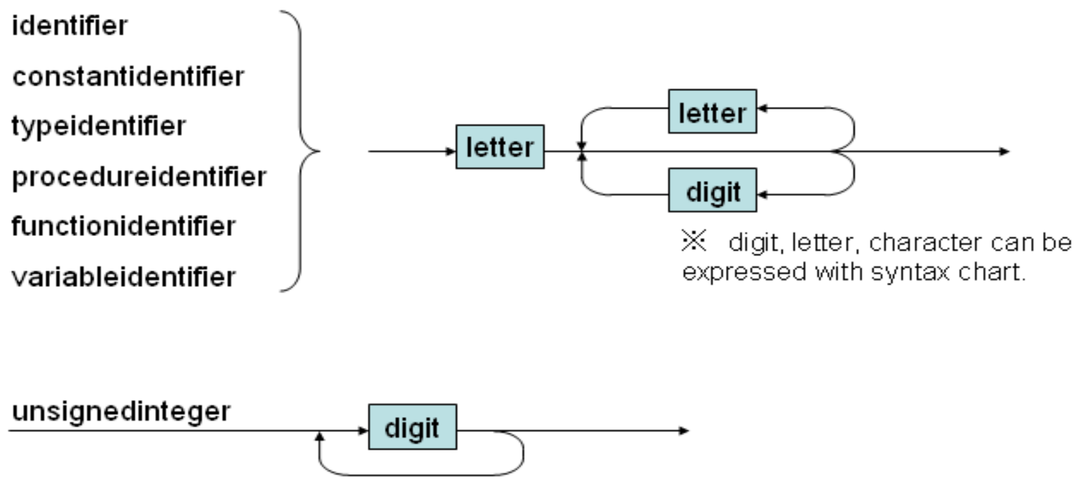


Figure 1.2 KPL syntax chart (cont.)

(Exercise 1.3) Point out syntax error in the following KPL program.

```
PROGRAM   QUICKSORT
     CONST   N:=20 ;
     VAR   A: ARRAY( .N. ) OF INTEGER ; I : INTGER ;

PROCEDURE   SORT( P, Q : INTEGER) ;
     VAR   I, J, X :INTGER ;
BEGIN
     I:=P;   J:=Q;   X:=A(( P+Q) / 2 ;
     REPART
          WHILE   A( .I. )< X DO I:=I+1 ;
          WHILE   X< A( .J. ) DO J:=J-1 ;
          IF   I<=J   THEN
          BEGIN
               W:=A( .I. );   A( .I. ):=A( .J. );   A( .J. ):=W;
               I:=I+1;   J:=J-1;
          END
     UNTIL   I   J ;
     IF   P< J   THEN   SORT( P, J) ;
     IF   I< Q   THEN   SORT( I, Q) ;
  END ;
BEGIN   FOR   I:=1   TO   N   DO   READI(A( .I. )) ;
           SORT(1, N) ;
           FOR   I:=1   TO   N   DO   WRITELN( A( .I. ))
END.
```

1.3.2 Semantics of KPL

There are many studies and suggestions about formal method to describe semantics of a programming language. For instance, English is used for ALGOL60, and Japanese for JIS. Since KPL has in the same linguistic system as ALGOL, I will address differences of two rather than explaining semantics one by one.

1) For block and effective range of name, the same concept as ALGOL is used.

2) Like PASCAL, KPL can define constant numbers and be referred by name. However, since it involves with constant, the value cannot be changed by substitution.

3) With KPL, type name can be defined. Integer (INTEGER) and character (CHAR) are two basic types. However, it is not possible to define complex type because types with structure are limited to array.
Declaration of type with six characters is like this;
```
TYPE    ALPHA=ARRAY(.6.) OF CHAR
```
Declaration of variable with the type ALPHA is described;
```
VAR    I:ALPHA;
```
Declaration of multi-dimensional array takes a form of repeating array declaration. For instance, a two-dimensional array is like this;
```
VAR    A:ARRAY(.10.) OF ARRAY(.5.) OF INTEGER;
```
The elements (I, J) are assigned by A(.I.)(.J.). ($1 \leqq I \leqq 10$, $1 \leqq J \leqq 5$)

4) Call of a procedure is done by call statement. Recursive calling is possible. Procedure and function need to be defined before referring.

5) There are two methods for passing ‘argument’ (ARG); i) “call by value” and ii) “call by reference”. A keyword ‘VAR’ is used to declare a parameter for call by reference. For call by value, the value of argument is calculated at first and then it is passed to procedure and function. It is prohibited to change the value of argument passed by this method within procedure and function. For call by reference, addresses of arguments are passed to procedure and function. When a formula is used instead of a variable as an argument, the processor will detect it as an error.

6) Analysis of IF statement
IF statements can be nested multiply. Although a program like below is ambiguous

in terms of syntax, the ELSE part is understood to form a pair with the IF statement closest to the ELSE part but not yet corresponding to it. Thus,

```
IF   A=B   THEN
IF   C=D   THEN   S2
ELSE   S3;
```

If written in ALGOL form,

```
if   A=B   then   begin   if   C=D   then   S2   else   S3   end;
```

but this doesn't mean

```
if   A=B   then   begin   if   C=D   then   S2   end
               else   S3;
```

7) The increment of control variable in FOR statement is always 1. Thus,

```
FOR   I:=1   TO   20   DO
```

In ALGOL form,

```
for   I:=1   step   1   until   20   do
```

8) It is not allowed to mix different types within one formula. In a substitution statement, types of both sides have to be identical. It is not possible to execute substitution on the whole statement at once by writing only an array name on the left-hand side of the substitute statement.

9) A blank works as a bound symbol.

10) A symbol string surrounded by (* and *) is a comment. Comment can be inserted wherever a blank can be inserted.

11) READC, READI, WRITEC, WRITEI, WRITELN are basic procedures for input-output. These procedures are regarded to be declaring that;

```
PROCEDURE   READC(VAR C: CHAR);
PROCEDURE   READI(VAR I: INTEGER);
PROCEDURE   WRITEC(C: CHAR);
PROCEDURE WRITEI(: INTEGER);
PROCEDURE   WRITELN;
```

These work for input and output of a character and integer, and WRITELN means to break at output.

(Exercise 1.4) Reexamine semantics for the Exercise 1.3. Also, consider what the "correct" program may be.

(Exercise 1.5) Create a program with KPL to solve following questions.

(1) Calculate the highest common factor for two positive integers using the Euclid mutual division.

(2) Calculate the product of two integer matrixes.

(3) Create a calculator of the reversed Polish notation. Input numbers are integers between -2 and 2.

(4) Convert inputted positive integers into Roman numerals.

1.4 Stack Calculator

We have seen the definition of KPL in the preceding section. A source program written in KPL is transformed to object codes by language processor. This object code belongs to the virtual stack calculator imagined here. If we actually had this stack calculator, it would be possible to execute object codes outputted by language processor. However, since we do not have it in this experiment, we will create an interpreter in order to execute instructions of the stack calculator. Figure 1.3 shows a sequence of program execution.

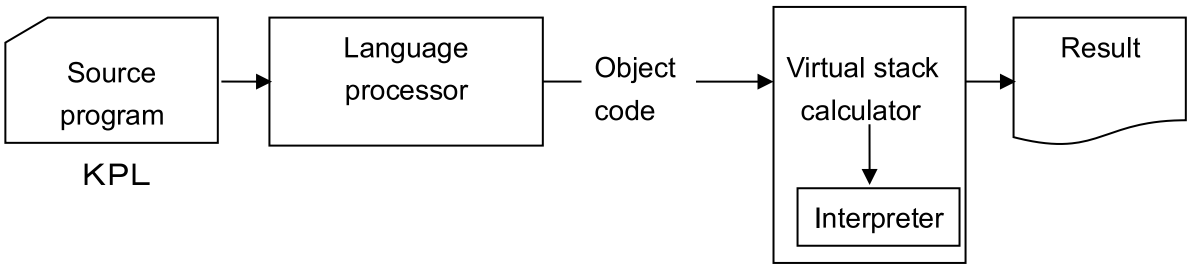


Figure 1.3 Program execution by KPL

1.4.1 Definition of stack calculator

By definition, a stack calculator consists of memory area to store instruction codes, stack calculator for operation, and four kinds of register including;

pc: program counter

t: stack top

b: base address of data area on the stack for active block

ps: status of active program

The structure is shown in Figure 1.4. In the calculation procedure, it shows a repetitive cycle with three procedures; 1) pulling out necessary numbers of operand from stack top, 2) execution of calculation, and 3) storing the result to stack top.

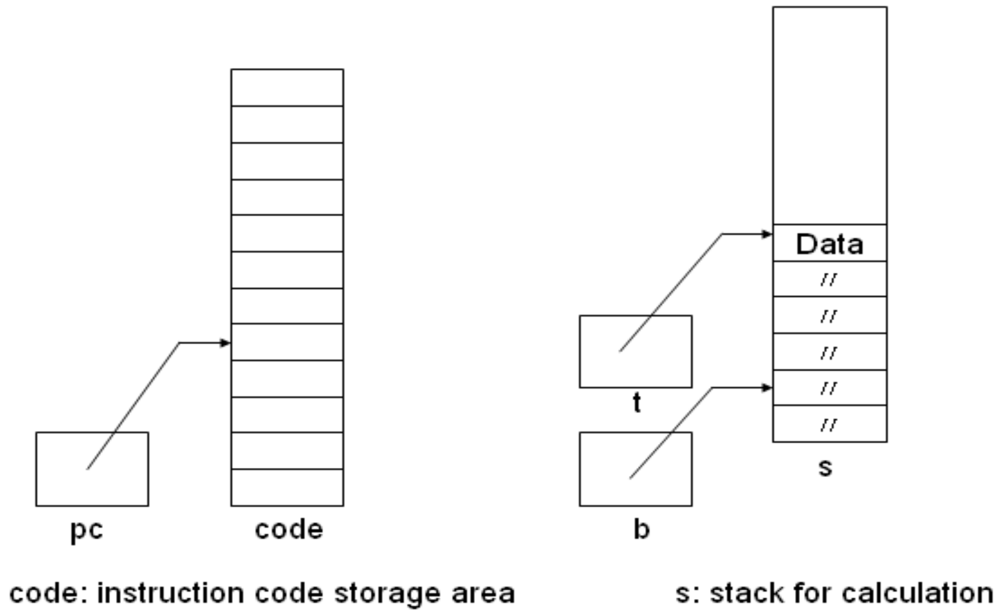


Figure 1.4 Structure of stack calculator

Program is supposed to be stored in the instruction code storage area in order from Address 0. The character ‘t’ refers to the one on the top of the stack when there are data in stack. The character ‘b’ refers to, as I will explain later, an area to put function result (FR) for the data area on the stack for the active block.

The status of ‘ps’ is one of following (Other definitions are also possible).

ps: 0: active
1: normal end (end with Halt instruction)
2: abnormal end due to devide error
3: abnormal end due to stack overflow
4: abnormal end due to input error
5: abnormal end due to output error

Form of instruction word for this stack calculator is as following;

op	p	q

Here, ‘op’ is an instruction code for the instruction word, and ‘p’ and ‘q’ are operand. Content of each instruction word is shown in Table 1.1.

‘Base’ in the explanation of instruction word is a function to track back the static link, and is expressed with function when described with PL/I. ‘Base(i)’ requires a base address at the block shallower by (i) levels than the current block. I will explain below about the static link in the block management section.

```
BASE: PROCEDURE(I)    RETUENS(FIXED    BIN);
          DCL    (I,I1,B1)    FIXED    BIN;
          /* FIND BASE I LEVELS DOWN */
          I1=I;
          B1=B;
          DO    WHILE(I1>0);
               B1=S(B1+3);
               I1=I1-1;
          END;
          RETURN(B1);
        END    BASE;
```

(Exercise 1.6) Create a calculator of the reversed Polish notation by using this stack calculator. Input numbers are integers between -2 and 2. Compare to the one written in KPL.

1.4.2 Block management by stack calculator

In KPL, bodies of program, function and procedure form so-called ‘block’. Block can have a nest structure. The identifier claimed at the beginning of a block is effective in the block, and for blocks within that block, it is effective until the same identifier is claimed again. Variables claimed in one block are local to the block, but it is impossible to statically allocate any area for variables because function and procedure are often recursively called out. As for stack calculator, it will suffice to secure dynamically the variable area local to function and procedure on the stack every time the function or procedure is called out, and to release it when returning. Hence, save the base stack address for the variable area to ‘b’, which is local to active function and procedure, and update it whenever you call out and return.

Instruction code	Instruction word	Content of instruction word [= (.,] = .)

LA	Load Address	t:=t+1; s[t]:=base(p)+q;
LV	Load Value	t:=t+1; s[t]:=s[base(p)+q];
LC	Load Constant	t:=t+1; s[t]:=q;
LI	Load Indirect	s[t]:=s[s[t]];
INT	Increment T	t:=t+q;
DCT	Decrement T	t:=t-q;
J	Jump	pc:=q;
FJ	False Jump	if s[t]=0 then pc:=q; t:=t-1;
HL	Halt	Halt
ST	Store	s[s[t-1]]:=s[t]; t:=t-2;
CALL	Call	s[t+2]:=b; s[t+3]:=pc; s[t+4]:=base(p); b:=t+1; pc:=q;
EP	Exit Procedure	t:=b-1; pc:=s[b+2]; b:=s[b+1];
EF	Exit Function	t:=b; pc:=s[b+2]; b:=s[b+1];
RC	Read Character	read one character into s[s[t]]; t:=t-1;
RI	Read Integer	read integer to s[s[t]]; t:=t-1;
WRC	Write Character	write one character from s[t]; t:=t-1;
WRI	Write Integer	write integer from s[t]; t:=t-1;
WLN	New Line	CR & LF
AD	Add	t:=t-1; s[t]:=s[t]+s[t+1];
SB	Subtract	t:=t-1; s[t]:=s[t]-s[t+1];
ML	Multiply	t:=t-1; s[t]:=s[t]*s[t+1];
DV	Divide	t:=t-1; s[t]:=s[t]/s[t+1];
NEG	Negative	s[t]:=-s[t];
CV	Copy Top of Stack	s[t+1]:=s[t]; t:=t+1;
EQ	Equal	t:=t-1; if s[t]=s[t+1] then s[t]:=1 else s[t]:=0;
NE	Not Equal	t:=t-1; if s[t]¬=s[t+1] then s[t]:=1 else s[t]:=0;
GT	Greater Than	t:=t-1; if s[t]>s[t+1] then s[t]:=1 else s[t]:=0;
LT	Less Than	t:=t-1; if s[t]<s[t+1] then s[t]:=1 else s[t]:=0;
GE	Greater or Equal	t:=t-1; if s[t]>=s[t+1] then s[t]:=1 else s[t]:=0;
LE	Less or Equal	t:=t-1; if s[t]<=s[t+1] then s[t]:=1 else s[t]:=0;

Table 1.1 Contents of instruction words for Stack calculator

Let's save the following items to a local variable area as information about function and procedure for KPL.

(1) The area to store results of function (FR); it is secured in case of procedure in order for unified treatment

(2) Dynamic link (DL)

(3) Returning address (RA)

(4) Static link (SL)

To understand why these should be secured is quite important to understand language processor later on. ‘RA’ is a returning address from function or procedure. ‘DL’ is necessary to update base address within b-register when returning. Addresses of variables are described by the block level (depth) – specified by ‘i’ as the block which is i levels shallower than the currently active block – and by relative addresses within block. Then, the actual stack address is decided. As for variables of a block other than currently active block, the base address of the block is necessary and SL is used for this purpose. The reason why SL is necessary along with DL can be understood if the example in Figure 1.5 is considered.

```
PROGRAM   P;
      VAR   J:INTEGER;

      PROCEDURE   Q;

PROCEDURE   R;
              VAR   J:INTEGER;
              BEGIN   (＊ R ＊)

                   CALL   Q;

              END   (＊ R ＊);

        BEGIN   (＊ Q ＊)

              CALL   R;   J:=J＋1;

        END   (＊ Q ＊);

    BEGIN   (＊ P ＊)

        CALL   Q;

    END   (＊ P ＊)
```

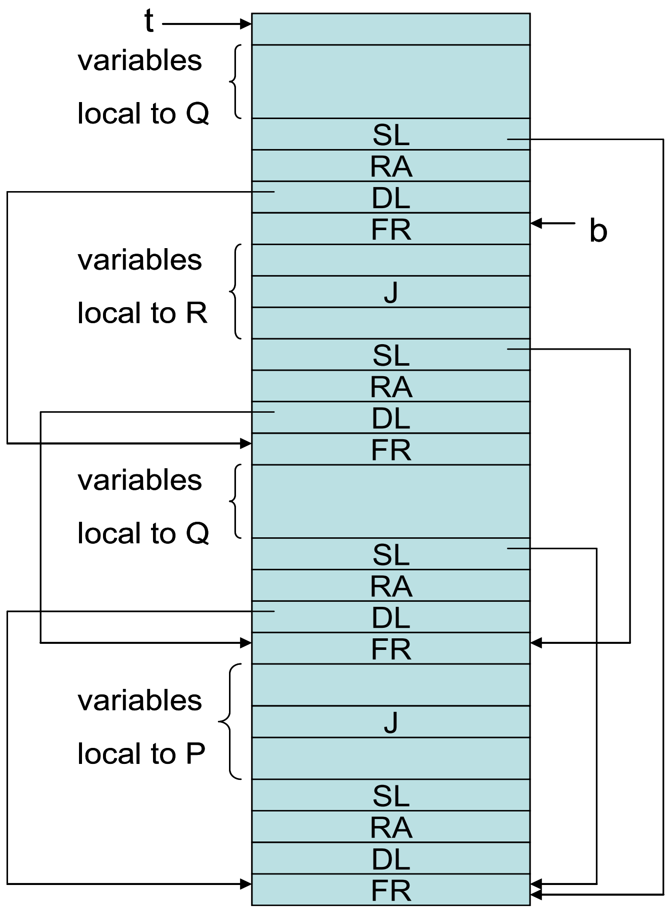


Figure 1.5 Block management by stack

Thus, when Q is called within P, when R is called within Q, and when Q is called within R, the local variable area on the stack becomes like the figure above. Suppose a variable of P within Q, the variable J for instance, is referred when Q is called out, the variable is in the variable area one level below it. But it is obtained not by tracing DL but only by tracing SL. DL is linked to the variable area local to R, and SL is to the one local to P. Look carefully the figure to see these relations.

1.4.3 Allocation of memory area

As we have seen above, memory area is allocated dynamically to the stack every time a block is activated. In a process of code generation, it is better to specify storage addresses of arguments and variables from relative addresses for block base. For that purpose, it is useful to use a new counter ‘dx’ whenever entering into a new block and allocate the current value of ‘dx’ as an address respectively to argument and variable. Then increase the value of ‘dx’ by the necessary amount of memory. In addition, the initial value of ‘dx’ becomes 4 as the information storage area is essential to block management for each block as we have seen earlier.

1.4.4 Code generation

In order to output generated codes to code sequences by a language processor, prepare firstly a program counter ‘pc’ for code generation and increase the value of ‘pc’ for each code generation. If you have understood Table 1.1 well, you may be fairly good at generating codes by now.

Following are prototypes of instruction codes for various typical statements. A language processor has to generate these instruction codes automatically for each statement of program by KPL.

```
1) Substitute statement        for   A:=E
                                       LA       A
                                       code for (E)              A:identifier
                                       ST                        E:expression

2) IF statement (1)        for   IF   C   THEN   S;
                                       code for (C)              C:condition
                                       FJ       L1               S:statement
                                       code for (S)
                                 L1:


3) IF statement (2)        for   IF   C   THEN   S1   ELSE   S2;
                                       code for (C)
```

```
              FJ      L1
              code for (S1)                C:condition
              J       L2                   S1, S2:statement
          L1: code for (S2)
          L2:
```

4) FOR statement   for FOR I:A TO B DO S;

```
              LA      I ---------  ⎧ When 'I' is a local variable.       ⎫
              code for (A)         ⎪ If 'I' is a parameter:              ⎪
              ST                   ⎪       LV (address of 'I' )          ⎪
              code for (B)         ⎩ Other codes must be changed.        ⎭
          L1: CV
              LV      I
              GE
              FJ      L2
              code for (S)                 I:identifier
              LA      I                    A, B:expression
              LV      I                    S:statement
              LC      I
              AD
              ST
              J       L1
          L2: DCT     1
```

5) WHILE statement   for WHILE C DO S;

```
          L1: code for (C)
              FJ      L2
              code for (S)                 C:condition
              J       L1                   S:statement
          L2:
```

6) Treatment of ARRAY

For VAR A:ARRAY(.Ih.) OF ARRAY(.Jh.) OF INTEGER;

the address of A(.I.)(.J.) is A11+(I-1)*elsize1+(J-1)*elsize2

For this, elsize2=1, elsize1=Jh*elsize2

```
          LA      A11

          code for (I)
          LC      1
          SB
```

```
LC      elsize1                 I, J:expression
ML
AD
code for (J)
LC      1
SB
LC      elsize2
ML
AD
```

Add 'Load Indirect' instruction in order to derive a value.

7) Call for PROCEDURE

```
For   PROCEDURE P(VAR N:INTEGER;   K:INTEGER);
      CALL P(I, E)   is
          INT     4
          LA      I
          code for (E)
          DCT     4 + Number of parameters       I:identirier
          CALL    P                              E:expression
```

(Think about call of function individually)

8) The beginning and ending of PROCEDURE

```
For   PROCEDURE P;
          BEGIN
            S;
          END (* P *);
            INT     4 + Number of parameters + Size of local area
            code for (S);                  S:statement
            EP
```

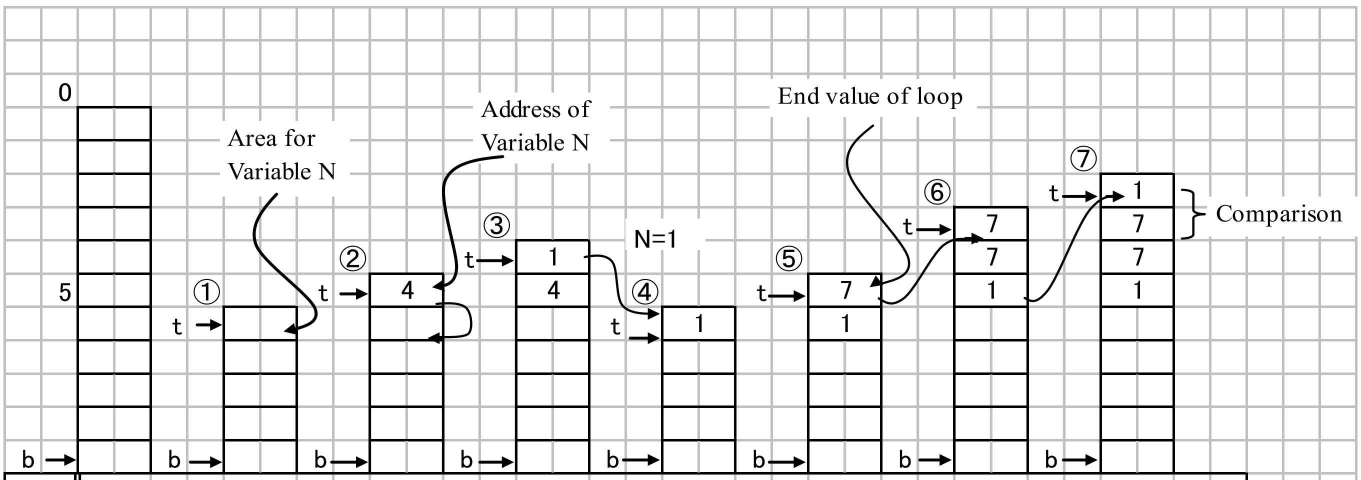
(Think about beginning and ending of function individually)

Note: For actual object codes, in the prototype above, operand identifiers (I and A) such as LA and LV have to be given as groups of 'p' and 'q'.

[Example 1.3]

The following is an example of object code generation for KPL program for factorial calculation in the Example 1.1. In this example, execution starts from Address 20. But it is still possible to generate codes to place Instruction J on Address 0 while in principle starting execution always from Address 0. By comparing object codes below and the source program in the Example 1.1, consider carefully why you generate object codes like above. I also illustrate a part of execution in the Figure 1.6.

Address	op	p	q												
0	INT	0	5	11	INT	0	4	22	LC	0	1	33	CALL	0	0
1	LV	0	4	12	LV	0	4	23	ST	0	0	34	WRI	0	0
2	LC	0	0	13	LC	0	1	24	LC	0	7	35	LA	0	4
3	EQ	0	0	14	SB	0	0	25	CV	0	0	36	LV	0	4
4	FJ	0	9	15	DCT	0	5	26	LV	0	4	37	LC	0	1
5	LA	0	0	16	CALL	1	0	27	GE	0	0	38	AD	0	0
6	LC	0	1	17	ML	0	0	28	FJ	0	41	39	ST	0	0
7	ST	0	0	18	ST	0	0	29	WLN	0	0	40	J	0	25
8	J	0	19	19	EF	0	0	30	INT	0	4	41	DCT	0	1
9	LA	0	0	20	INT	0	5	31	LV	0	4	42	HL	0	0
10	LV	0	4	21	LA	0	4	32	DCT	0	5				

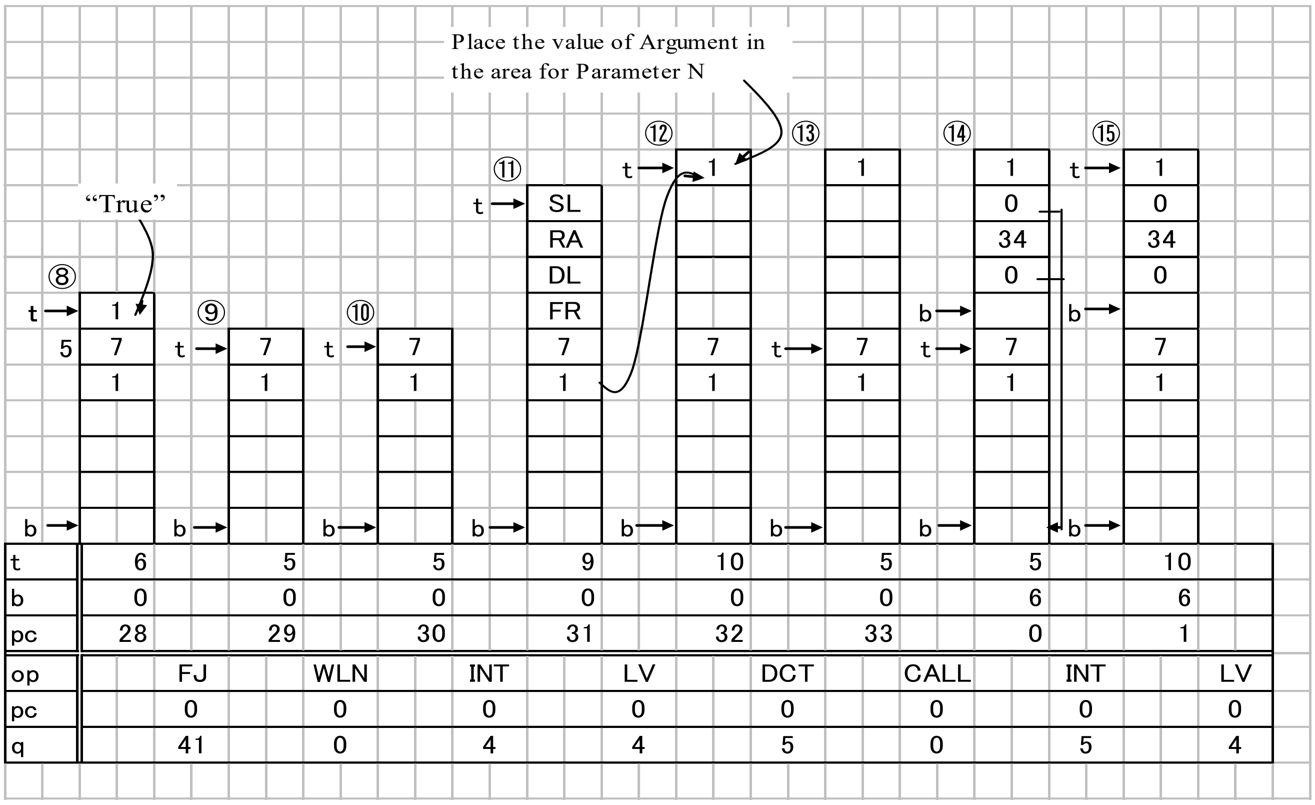


t	−1	4	5	6	4	5	6	7
b	0	0	0	0	0	0	0	0
pc	20 ※	21	22	23	24	25	26	27

op	INT	LA	LC	ST	LC	CV	LV	GE
pc	0	0	0	0	0	0	0	0
q	5	4	1	0	7	0	4	0

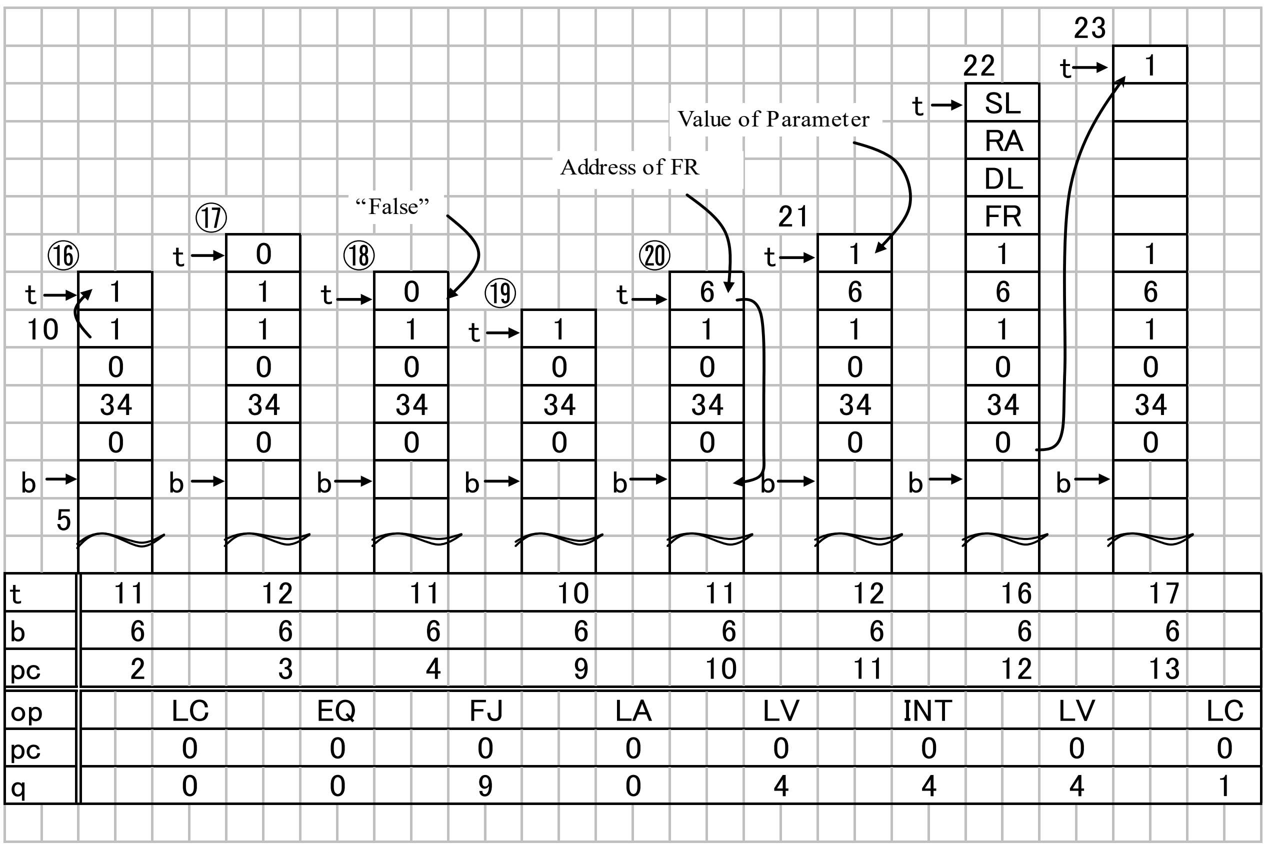
※ Initialize “pc” to the address of the first object code of the main block (body of the PROGRAM level 1) of user program. Increase by 1 after instruction fetching.

Figure 1.6 Operation of stack



t	6	5	5	9	10	5	5	10
b	0	0	0	0	0	0	6	6
pc	28	29	30	31	32	33	0	1

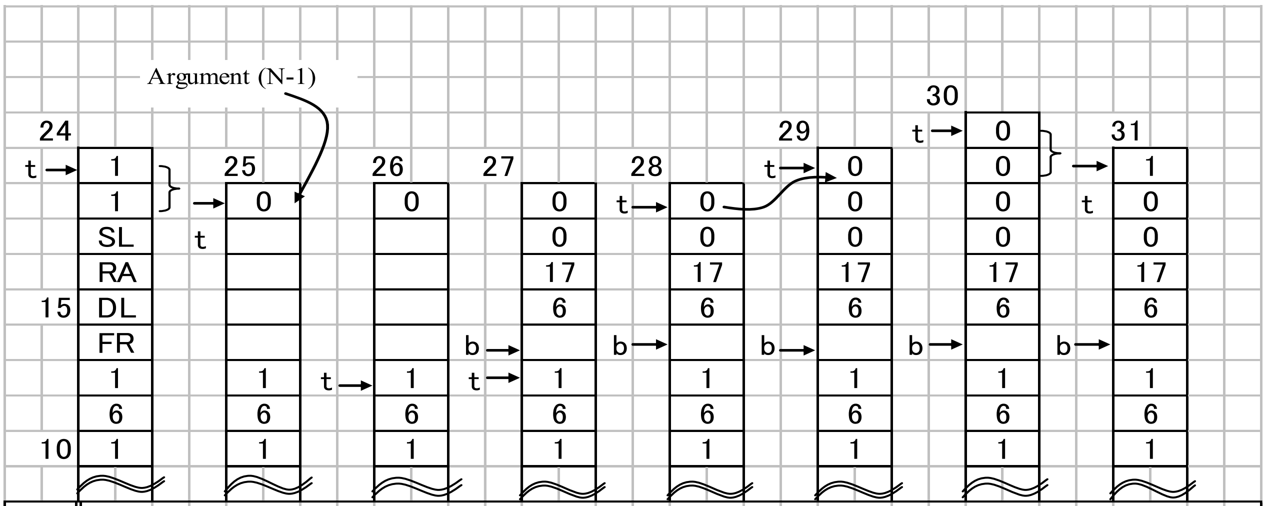
op	FJ	WLN	INT	LV	DCT	CALL	INT	LV
pc	0	0	0	0	0	0	0	0
q	41	0	4	4	5	0	5	4



t	11	12	11	10	11	12	16	17
b	6	6	6	6	6	6	6	6
pc	2	3	4	9	10	11	12	13

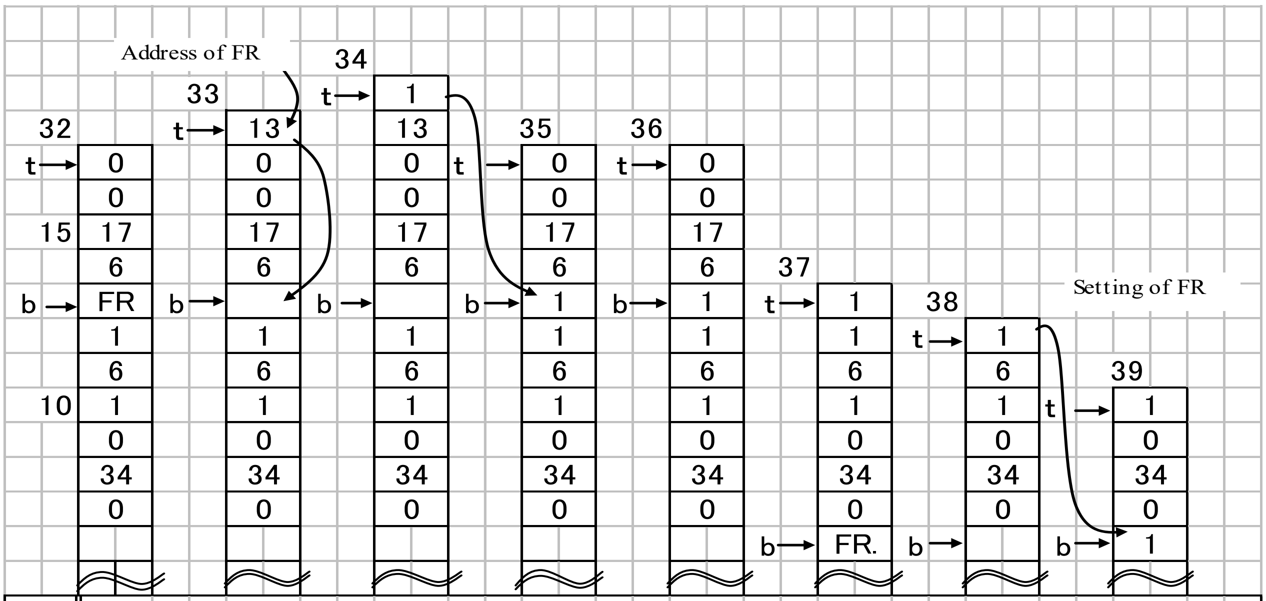
op	LC	EQ	FJ	LA	LV	INT	LV	LC
pc	0	0	0	0	0	0	0	0
q	0	0	9	0	4	4	4	1

Figure 1.6 Operation of stack (cont.)



	24	25	26	27	28	29	30	31
t	18	17	12	12	17	18	19	18
b	6	6	6	13	13	13	13	13
pc	14	15	16	0	1	2	3	4

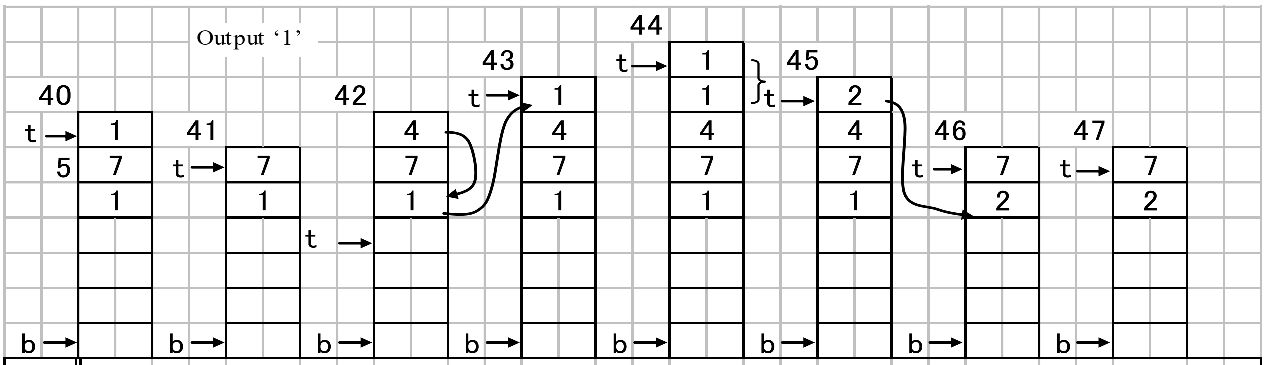
op	SB	DCT	CALL	INT	LV	LC	EQ	FJ
pc	0	0	1	0	0	0	0	0
q	0	5	0	5	4	0	0	9



	32	33	34	35	36	37	38	39
t	17	18	19	17	17	13	12	10
b	13	13	13	13	13	6	6	6
pc	5	6	7	8	19	17	18	19

op	LA	LC	ST	J	EF	ML	ST	EF
pc	0	0	0	0	0	0	0	0
q	0	1	0	19	0	0	0	0

Figure 1.6 Operation of stack (cont.)



t	6	5	6	7	8	7	5	5
b	0	0	0	0	0	0	0	0
pc	34	35	36	37	38	39	40	25

op	WRI	LA	LV	LC	AD	ST	J	CV
pc	0	0	0	0	0	0	0	0
q	0	4	4	1	0	0	25	0

Figure 1.6 Operation of stack (cont.)

(Exercise 1.7) Run a check on execution of this object code by a stack calculator up to N=3 on the desk.

1.5 Syntactic analysis algorithm

We have studied so far KPL and its object codes. Some of you may be confident by now to design this language processor. However, as I have mentioned in the beginning, the aim of this experiment is to create a systematically and structurally organized program. In reality, the basis for it has already been prepared within the definition of KPL. In this section, we will study two methods of syntactic analysis; “LL(1) grammar” and “recursive descent.”

1.5.1 Generative grammar

In order to talk about syntactic analysis algorithm, it is necessary to understand terms of the “generative grammar” by N. Chomsky. Let me start from a little mathematical explanation.

Suppose the grammar G is expressed in this way;

G=(VN, VT, P, S)

VN: finite set of non-terminal symbols (metalinguistic variable) [element is generally expressed in a capital letter]

VT: finite set of terminal symbols [element is generally expressed in a small letter]

P: finite set of generation rules [element generally has a form $\alpha \to \beta$]
S: start symbol ($S \in VN$)

Provided VN and VT do not have a common part. That is, $VN \cap VT = \phi$. Also, α and β are symbol strings consist of elements of $V=VN \cup VT$ with length longer than 1 and 0 respectively. These are expressed as $\alpha \in V+$ and $\beta \in V*$. V+ and V* are sets of all symbols consist of symbols within V with length longer than 1 and 0 respectively. The string with length 0 is generally expressed with a symbol ε.
When $\alpha \to \beta$ is to be a generation rule of P and γ and δ to be arbitrary symbol strings of V*, the relationship between a string $\gamma\alpha\delta$ and a string $\gamma\beta\delta$ that is obtained by replacing α with β in the former string, is written like this;

$$\gamma\alpha\delta \underset{G}{\Longrightarrow} \gamma\beta\delta$$

For grammar G, $\gamma\alpha\delta$ "directly derives" $\gamma\beta\delta$, and $\gamma\beta\delta$ is "directly resolved" to $\gamma\alpha\delta$. G is often unnecessary to write when grammar is obvious. When α_1, …, αm are symbol strings within V* and $\alpha i \Rightarrow \alpha i+1$ (i=1, … , m-1), the relation is expressed in this way;

$$\alpha 1 \overset{*}{\underset{G}{\Longrightarrow}} \alpha m$$

For grammar G, $\alpha 1$ "derives" αm, and αm is "resolved" to α_1. The language L(G) generated by the grammar G is defined like this;

$$L(G)=\{W \mid S \overset{*}{\underset{G}{\Longrightarrow}} W, W \in VT^*\}$$

When $L(G_1)=L(G_2)$, G_1 and G_2 have an equal value. Symbol strings, which are derived from the start symbol S and consist of terminal and non-terminal symbols, are called "statement form." If it only consists of terminal symbols, it is called "statement."

[Example 1.4] The grammar of operation formulas including addition and multiplication

$G_1= (\{E, T, F \},\{ a, b, +, *, (,)\},P1, E)$
$P_1=:$ $E \to E + T \mid T$
$T \to T * F \mid F$
$F \to (E) \mid a \mid b$

Here, " | " is an abbreviation of the generative grammar, and the first line of P_1 is a combined form of two generation rules "E→E+T" and "E→T".

(Exercise 1.8) Give an example of formula derived from the grammar of Example. 1.4.

1.5.2 The context-free grammar and the parse tree

I have briefly explained in the last section what the generative grammar is. Class of grammar expressed with generative grammar is so extensive that, for the widest class, it will be necessary to consider every possibilities of examining whether the grammar fits to today's calculator (or every possible calculator in the future). The grammar of KPL is actually designed suitable to effective syntactic analysis on a calculator. It belongs to the class called the "context-free grammar."

The context-free grammar is a grammar which has generation rules like this;

$A \rightarrow \alpha, \quad A \in VN, \quad \alpha \in V^*$

For a form of generation rule, the left-hand side consists of only one non-terminal symbol. A language generated with context-free grammar is called "context-free language."

Check that the grammar of operation formula in Exercise 1.4 is actually written in context-free grammar.

Then, why is it important? This feature of the context-free grammar means that, in the process of deriving statement form, generation rules are applicable whenever non-terminal symbol exists within the statement form regardless of its surroundings since its generation rule is that the left-hand side has only one non-terminal symbol. This is why it is called "context-free." Therefore, you may intuitively notice that it is inadequate for grammars of natural languages we use in our everyday life such as Japanese and English. However, it is quite often sufficient for artificial languages like programming language. You will see later that KPL can indeed be expressed in the context-free grammar.

The procedure to derive statements with the context-free grammar can be expressed in a tree-form. It is called the "derivation tree", or the "parse tree" in terms of syntactic analysis. This is an important point because at this point you are about to see how "context-free grammar", "tree expression", "recursive program", and "stack calculator" come to be related with each other. Let me give you an example of the parse tree.

[Example 1.5] Deriving ‘a+b*a’ with the grammar in Example. 1.4.

$E \Rightarrow E + T$ (apply $E \to E+T$)
$\Rightarrow T + T$ (apply $E \to T$)
$\Rightarrow F + T$ (apply $T \to F$)
$\Rightarrow a + T$ (apply $F \to a$)
$\Rightarrow a + T * F$ (apply $T \to T*F$)
$\Rightarrow a + F * F$ (apply $T \to F$)
$\Rightarrow a + b * F$ (apply $F \to b$)
$\Rightarrow a + b * a$ (apply $F \to a$)

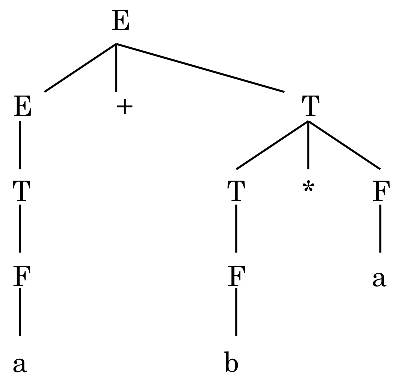


Figure 1.7 Parse tree for a+b*a

This parse tree is shown in Figure 1.7. Look carefully correlations.

(Exercise 1.9) Obtain a parse tree for the operation formula created in the Exercise 1.8.

1.5.3 Syntactic analysis

It is necessary for a language processor, not only to check whether symbol strings inputted as a source program are grammatically correct or not, but also to generate object codes. Therefore, we have to analyze the structure of the source program, in other words, how each symbol is in relation with one another to form a program. Such an analysis is called “syntactic analysis (syntactic analysis, parsing)”.

Syntactic analysis for the context-free grammar is thought as obtaining a parse tree. In this section, I will explain important points to notice concerning rules of syntactic analysis for the context-free grammar. Let’s start the discussion first putting aside for now the order of application and way of choosing generation rules.

[Example 1.6] Analyze a+b*a with the grammar G_2 below.

$G_2 = (\{E\}, \{a, b, +, *, (,)\}, P_2, E)$

$P_2 : E \to E + E \mid E * E \mid (E) \mid a \mid b$

The ‘a+b*a’ can also be obtained as a statement of this grammar. In reality, $L(G_1) = L(G_2)$. However, it is noticeable that there are two kinds of parse tree for G_2 as shown in Figure 1.8. In regard of the object code generation for operation formula, (a) in the figure should be seen as a+(b*a), while (b) as (a+b)*a. Then, we will end up with having different results from calculation.

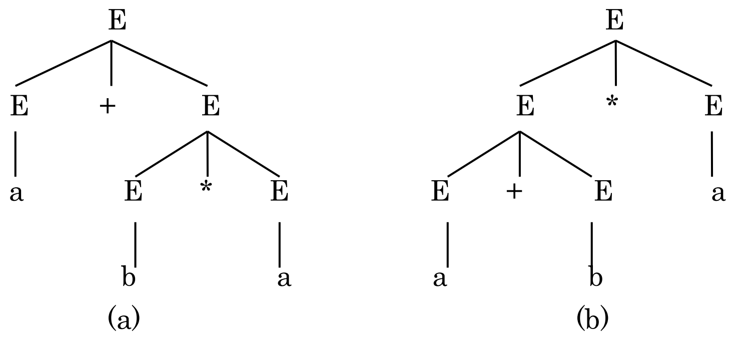


Figure 1.8 Two kinds of parse tree for a+b*a

It will be a problem when generating object codes with language processor if parse tree is not uniquely determined like above.

The context-free grammar G is seen to be ambiguous when there is a statement with more than two different parse trees in L(G). Among context-free languages, there is the one seen as "inherently ambiguous" because any grammar that generates it is ambiguous.

Programming languages are normally defined not to be ambiguous. However in some cases, language is defined semantically unambiguous even though syntactically ambiguous.

IF-statement may be an example for KPL. Although it is still possible to design a grammar for IF-statement which is not ambiguous, it will make the language rather complex. And, in calculation, such an ambiguity is easily removed in the process of analysis.

(Exercise 1.10) Design an unambiguous grammar for four operations by using a, b, c, and (,). Conduct syntactic analysis on 'a-b-c' with the grammar. What may be the grammar which has more than one parse tree for 'a-b-c'?

There are various procedures to derive statements even if the grammar is not ambiguous. It is a problem of application order of generation rules. When there are numbers of non-terminal symbols within a statement form, it is possible to apply generation rules to any of them. The result is the same parse tree.

There are two ways to derive statements by starting from a start symbol. When applying generation rules always to the non-terminal symbol on the left end, the procedure is called "leftmost derivation". Thus, in the case of $\beta \in VT$, direct derivation $\beta A\gamma \Rightarrow \beta\alpha\gamma$ is possible by a generation rule $A \rightarrow \alpha$. On the contrary, if $\beta A\gamma \Rightarrow \beta\alpha\gamma$ and $\gamma \in VT$, it is called "rightmost derivation".

[Example 1.7] The derivation in Exercise 1.5 was the leftmost derivation. The rightmost derivation of the same a+b*a by G_1 is like this;

$$
\begin{aligned}
E &\Rightarrow E+T \\
&\Rightarrow E+T*F \\
&\Rightarrow E+T*a \\
&\Rightarrow E+F*a \\
&\Rightarrow E+b*a \\
&\Rightarrow T+b*a \\
&\Rightarrow F+b*a \\
&\Rightarrow a+b*a
\end{aligned}
$$

In any case, the parse tree is as in the Figure 1.7.

The "top-down" syntactic analysis is the method to analyze the grammatical structure of given statements by starting from a start symbol following the leftmost derivation. On the other hand, the "bottom-up" syntactic analysis is the one to analyze the grammatical structure by scanning a given statement from left to right following the rightmost derivation backward sequentially to the start symbol.

The top-down syntactic analysis is convenient to write a program, but has a weakness on analytic capacity. In reality, the bottom-up syntactic analysis is more capable, that is to say, has a wider range of applicable grammar. We do not go into the bottom-up syntactic analysis in this experiment, but it will be beneficial for you to study further about it if interested. Also, there is a method which uses "precedence grammar", and it is effective for syntactic analysis of operation formula. [4][5]

[Example 1.8] Let's go back to the top-down syntactic analysis of a+b*a by the grammar G_1. I put generation rules of G_1 again here for your convenience. Underlined parts of 'a+b*a' are important points in the below because it is necessary to decide which one to choose when there are several generation rules with the same non-terminal symbol on the left-hand side.

$$
\begin{aligned}
P_1 :\ & E \rightarrow E + T \mid T \\
& T \rightarrow T * F \mid F \\
& F \rightarrow (E) \mid a \mid b
\end{aligned}
$$

a+b*a	$E \Rightarrow E+T$	Read up to '+', it's noticeable to apply E→E+T
a+b*a [space]	$\Rightarrow T+T$	Read to the end, it is an addition of two terms
a+b*a	$\Rightarrow F+T$	Choose T→F from '+'
a+b*a	$\Rightarrow a+T$	Choose F→a from 'a'
a+b*a	$\Rightarrow a+T*F$	Choose T→T*F from '*'
a+b*a [space]	$\Rightarrow a+F*F$	Read to the end, it is a multiplication of two factors

a+<u>b</u>*a ⇒ a+b*F Choose F→b from ‘b’
a+b*<u>a</u> ⇒ a+b*a Choose F→a from ‘a’

As in this example, it is usually a time-consuming task to decide which one to choose when there are several generation rules with the same non-terminal symbol on the left.

There is another method which process syntactic analysis by applying any generation rule and “back-track” the string when it comes to an impasse. But this is not an efficient way as it requires tree-search task. For the context-free grammar, there is actually a slightly more efficient method for general grammar. It is called “Earley’s algorithm” (by J. Earley). With this method, it is possible to conduct syntactic analysis on the length of input symbol string n, by the order of n^2 (calculation time proportionate to n^2) if the grammar is not ambiguous, or by the order of n^3 for any context-free grammar. There also is a study of method slightly faster than this as an order.

However, because it is nevertheless inefficient, a programming language is usually designed possible to conduct syntactic analysis by the order of n, by imposing restriction on grammar. KPL is the grammar of this kind. I will explain it in the next section.

1.5.4 LL(1) grammar

Among context-free grammar, the grammar of KPL belongs to the class called the “LL(1) grammar.” Before explaining it, let us revisit problems of syntactic analysis in Example 1.8.

The biggest problem lies in the second and the sixth lines of analysis. That is because generation rules cannot be decided without foreseeing processes to a greater degree since it is necessary to consider the possibility of calculation with more than three terms. If I use more intuitive language, it occurs because a terminal symbol on the left-hand side of generation rule is included as well on the left end of the right-hand side, as in a example of E→E+T. The cause of such a difficulty comes from that if you try to find out if the left-hand side of symbol string to be analyzed corresponds to the form of E, in some cases you have to continue checking forever whether or not the left-hand side is E. Such a generative rule is said to be “left-recursive”.

It is possible, by changing generation rules, to remove the left-recursive generative rule without changing the language generated. There are several ways for it. I adopt a notation of generative grammar expanded within a range suitable for calculator, because language processor is not fixed on a certain notation for already defined generation rules. Following the notation, E→E+T is written as E→T{+T}. Here, the item inside { } indicates that any number of repetition (more than 0) is allowed.

[Example 1.9] Define the grammar G_3, which has an equal value with G_1, like below by using the expanded notation, and try again the syntactic analysis of a+b*a.

G_3 = ({E, T, F}, {a, b, +, *, (,)}, P_3, E)
P_3 = E → T {+T}
T → F {*F}
F → (E) | a | b

a+b*a	E ⇒ T{+T}	Leave the item within { } yet as a possibility
a+b*a	⇒ F{*F}{+T}	
a+b*a	⇒ a{*F}{+T}	
a+b*a	⇒ a{+T}	
a+b*a	⇒ a+T{+T}	
a+b*a	⇒ a+F{*F}{+T}	
a+b*a	⇒ a+b{*F}{+T}	
a+b*a	⇒ a+b*F{*F}{+T}	
a+b*a	⇒ a+b*a{*F}{+T}	
a+b*a [space]	⇒ a+b*a{+T}	
a+b*a [space]	⇒ a+b*a	

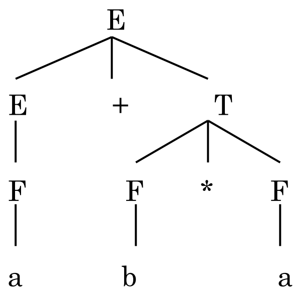


Figure 1.9 Parse tree of a+b*a by G_3

The parse tree is shown in Figure 1.9.

In this example, syntactic analysis seems quite easy as it can be completed by reading only one character without back-tracking.

The LL(1) grammar is a context-free grammar of this kind which allows top-down syntactic analysis by one-character reading without back-track. In general, reading ahead k characters, it is called LL(k) grammar Other than this, there is a class called LR(k) grammar which reads k characters ahead. You can work by yourself on it if you are interested.

I expanded the notation of generation rules here, but it is also possible to express such grammars without using { } (You can use a string ε with length 0.).

The following rules are conditions for the LL(1) grammar to be able to do top-down syntactic analysis by one-character reading without back-track.

(Rule 1)

When there is a generative rule in a form of "A→$\xi_1|\xi_2|\ldots|\xi m$", there must not be a common part for the class of the first symbols of all statements derived from ξi . That is;

first (ξi) ∩ first (ξj) = ϕ for all i≠j

Class 'first (i)' is the class of the first symbols of all statements derived from ξ, and is calculated following next rules.

When 1. first $(a\xi) = \{a\}$

2. $A \to \alpha_1 | \alpha_2 | \ldots | \alpha m$

first $(A\xi)$ = first $(\alpha_1) \cup$ first $(\alpha_2) \cup$ --------- $\cup$ first (αn)

(Rule 2)

For all non-terminal symbols which generate a symbol string ε with length 0, following conditions need to be satisfied.

first (A) ∩ follow (A) = ϕ

'Follow (A)' is a set of possible symbols which follow A and it can be calculated by the following manner.

For all the generation rules Pi which include A on the right, "$xi \to \alpha iA\beta i$", 'follow (A)' is calculated by obtaining a joint class of 'first(βi)'. When βi derives ε, you will also calculate 'follow (xi)' and add it to the class above to obtain 'follow (A)'.

The following is a procedure of top-down syntactic analysis. Suppose the grammar is not ambiguous.

Apply generation rules sequentially with the leftmost derivation starting from a start symbol. Suppose you have to choose one from generation rules with more than one right-hand side. That is,

$A \to \alpha_1 | \alpha_2 | \ldots | \alpha m$

and you have to apply generation rules to A. Then, find out one input symbol. It is seen to belong to the class 'first (αi)'. There won't be no more than one 'i' for it as 'first (αi)' does not have any common part from Rule 1. When 'i' is fixed, adopt $A \to \alpha i$.

Supposing that the above 'i' does not exist, there is an input error if A does not derive ε. Or supposing αj derives ε, adopt $A \to \alpha j$ when one symbol of the next input belongs to 'follow (A)'. If it does not belong to 'follow (A)', there may be an input error. There won't be more than one 'j' if the grammar is not ambiguous.

(Exercise 1.11) In the notation for expanded generation rules, how do we think about application of above generation rules concerning the item within { }?

At this point, you can go back to KPL again now. The syntax chart of KPL given in the Figure 1.2 is actually equivalent to the notation of expanded generation rules. Also, KPL belongs to the LL(1) grammar.

In the Figure 1.2, non-terminal symbol is written on the upper left of each part of the syntax chart, and 'program' is a start symbol. Non-terminal symbol appears within the square in the chart. Regard a symbol inside circle as terminal symbol. Branches correspond to '|', and loop to '{ }'.

(Exercise 1.12) What is the Backus notation (Backus-Naur form, Backus normal form, BNF)? Write syntax of KPL with the expanded Backus notation. As it may be difficult to find one-by-one correspondence with the syntax chart, you can divide it arbitrarily. It is also convenient if you can generate a symbol string with length 0.

The following is conditions for syntactic analysis without back-tracking.

1. For each branch on the chart, any two branches must not start with the same symbol.
2. For the chart A which can be passed without reading input symbol, label all symbols which can follow branches passed without reading the input symbol. With this process, which branch you choose should be decided by input symbol.

It is possible, like above, to conduct syntactic analysis of the LL(1) grammar by using syntax chart rather than generation rules.

(Exercise 1.13) Check these conditions on the syntax chart of KPL.

1.5.5 Recursive descent

Program becomes easy to create and understand if you use the “recursive descent” method when creating a top-down syntactic analysis program. To adopt this method, it has to be possible to use recursive procedure for programming language to write a syntactic analysis program.

In simple terms, this method is to correlate each non-terminal symbol to one procedure. And you should make the form of the right-hand side of generation rules with that non-terminal symbol on the left directly into a program. If non-terminal symbol appear on the right-hand side, you only need to call a procedure which corresponds to it.

Let me explain general principles to create a syntactic analysis program from syntax. S* indicates a program translated from figure S.

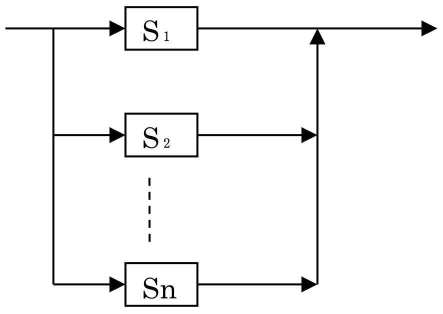
R1. Syntax chart should be compiled into as small number of (independent) charts as possible by substitution.

R2. Translate each independent chart to a program with following principles, and add appropriate declaration to create ‘PROCEDURE’.

R3. $\rightarrow\boxed{S_1}\rightarrow\boxed{S_2}\rightarrow$ ------ $\rightarrow\boxed{Sn}\rightarrow$ is translated into a series of statement groups,

S_1*; S_2*; ------ Sn*

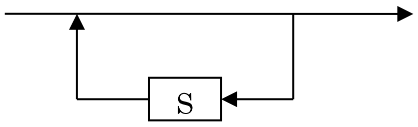
R4. When there are branches.



```
IF   IN(SY,L1)   THEN   S*1 ;
ELSE   IF   IN(SY,L2)   THEN S*2 ;
                  ⋮
ELSE   IF   IN(SY,Ln)   THEN   S*n ;
ELSE   CALL   ERROR ;
```

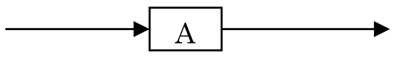
SY here indicates the next symbol, and Li indicates ‘first (Si)’. PROCEDURE IN which gives a theoretical figure becomes 1 when SY is included to Li, and it becomes 0 in other cases. This means to create efficient programs in individual cases.

R5. Translation when there is a loop



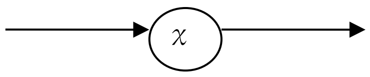
```
DO   WHILE(IN(SY,L)) ;
         S*n ;
END ;
```

R6. For an element of another chart A, translate it to a statement to call PROCEDURE which recognizes A.



```
CALL   A;
```

R7. Translation for an element of chart which indicates the terminal symbol x



```
IF   SY=x   THEN   CALL NEXTSYMB(SY);
ELSE   CALL   ERROR;
```

NEXTSYMB here is a procedure to substitute the next terminal symbol into a variable SY.

[Example 1.10] The following shows a syntactic analysis program written with PL/I concerning only ‘term’ within the syntax chart in the Figure 1.2. Look carefully correlations with the chart.

```
TERM:PROCEDURE;
```

```
  CALL   FACTOR;
  DO   WHILE(SY= '*'|SY= '/');
     IF   SY= '*'   THEN
        DO;
           CALL   NEXTSYMB(SY);
           CALL   FACTOR;
        END;
     ELSE
        DO;
           CALL   NEXTSYMB(SY);
           CALL   FACTOR;
        END;
   END;
 END   TERM;
```

The syntactic analysis program written with the recursive descent method is easy to insert programs for object code generation and semantics processing and to create a language processor.

[Example 1.11] Let's think about how you can incorporate an object code generation program into the syntactic analysis program for the grammar G_3 in the Example 1.9. Since object codes are for stack calculator, they should correspond to calculation with the reverse Polish notation.

In the reverse Polish notation, for instance, normal 'a+b' is expressed as 'ab+'. If you can generate object codes of stack calculator in the order of a, b, and +, the execution is directly possible. The following is the procedure.

Object code of an operand is outputted immediately after reading the operand. Since syntactic analysis program of the recursive descent method directly corresponds to generation rules, I illustrate types of object code generation using generation rules. For example, if a object code for 'a' is expressed as [a], object code generation for an operand goes like this;

$F \rightarrow a\ [a]$

Then, how about operator? It is only necessary to do it immediately after the subsequent operand analysis and object code generation. In a case of '+', for instance, it goes like this;

$E \rightarrow T\ \{+T[+]\}$

Although the original formula may have brackets, in a object code generation of the reverse Polish notation, there is no need to generate object codes but only to analyze these. In sum, object code generation for the entire generation rules of the grammar G_3 is like the following.

P: $E \rightarrow T\ \{+T[+]\}$

T → F {*F[*]}
F → (E) | a [a] | b [b]

Figure 1.10 shows the extended syntax tree in the case of a+b*a. Object code generation is also expressed in this tree. When you create a language processor with the recursive descent method, you can understand with what process of analysis object codes are generated if you follow the tree focusing on its depth.

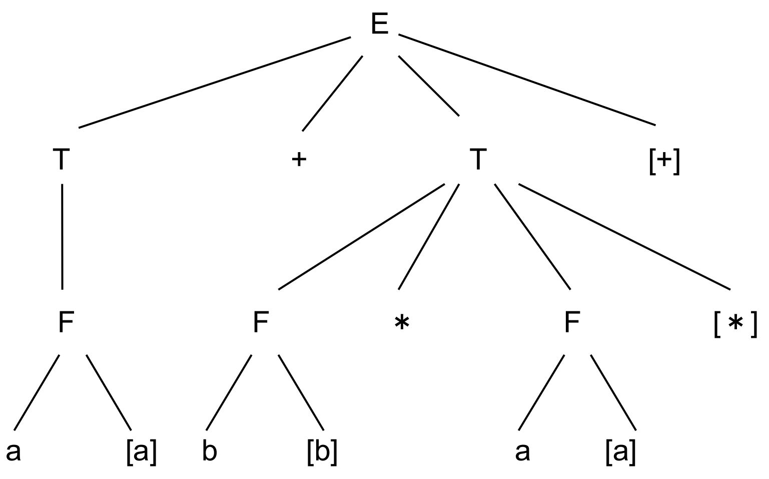


Figure 1.10 Object code generation in the syntactic analysis of a+b*a

(Exercise 1.14) How do object codes, which should generate at each object code generation point, exactly look like?

Some of you might have noticed when we extended notation of generation rules that, with such an extension, the binary operation is extended to general multinomial operation. Therefore, while it is fine in cases of operations in which associativity laws like '+' or '*' exist, there will be a problem of calculation order if there is no associativity law in calculation as in the cases of '-' or '/'. To use the method in the Example 1.11 is to add another condition that a priority is given to the left-hand side as meanings of arithmetic operation at the time of object code generation. Consider this fact.

(Exercise 1.15) How do you think about syntactic analysis and the system of object code generation like this? Write what you have thought and studied.

1.6 Design Principles for Processor.

This experiment aims to create a language processor by using the top-down syntactic analysis based on the LL(1) grammar. To create a program, we use the recursive descent method explained in the section 1.5.5. The skeleton framework of program module is as shown in the Figure 1.11. Here, arrows indicate call relations, and doubled-arrows indicate flows of information. I explain points of processor creation in the following.

1.6.1 Word analysis part

Word analysis program, as discussed in the section 1.2, requires dividing character strings into words. Types of word include; keyword, identifier, integer constant, character constant, special symbols (operator, special character, etc.).
You will conduct the decimal-to-binary encode for an integer constant and calculate its value. Also, you should be careful about that some special symbols consist of two special characters. Comments to make program reading easy will be removed in the process of word analysis. Generation rules related to word analysis can usually be rewritten into the following form;

$A \rightarrow ab$

$A \rightarrow a$

The grammar in which generation rules take the above form is called the 'Chomsky's third grammar', or the 'regular grammar'. Since the regular grammar is included into context-free grammar, it is possible to use syntactic analysis algorithm for context-free grammar. However, because the regular grammar as a class of grammar is smaller than context-free grammar, much more efficient methods are used in general. The regular grammar can be recognized by a machine with finite numbers of status, the finite automaton. Therefore, regarding the word analysis part as a finite automaton, it is often developed by a transition table. However, the program of word analysis part is relatively simple and it is quite often possible to directly create a program for the syntax chart about word analysis.

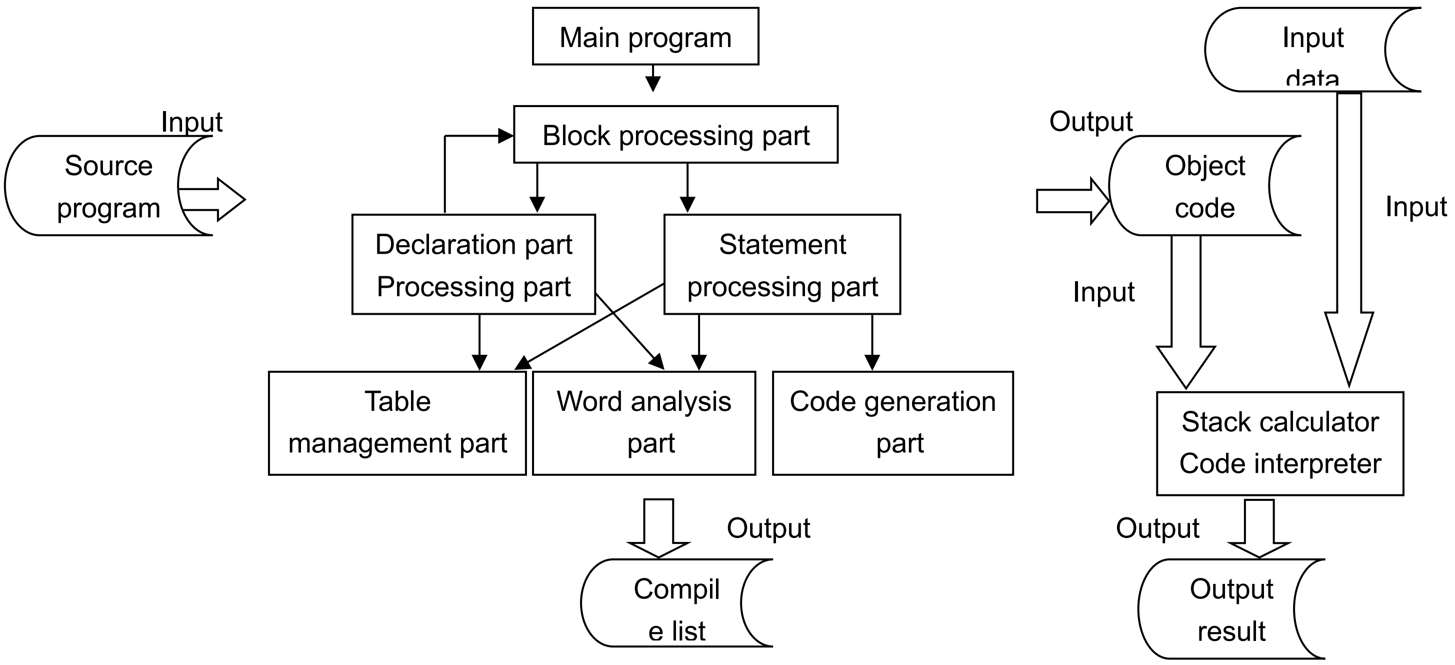


Figure 1.11 Module structure of language processor

I show in Figure 1.12 a flow chart of a program when word analysis part is created on the basis of finite automaton. Also, I show in the table 1.2 input and output of program of this word analysis part. This chart is always constructed under the principle of one-character reading-ahead. Moreover, just like the word analysis part does ‘one-character reading-ahead’, it is convenient to create the word analysis part regarding it does ‘one-character reading-ahead’ too. The way to output is strongly dependent on the way of designing various tables discussed in the next section.

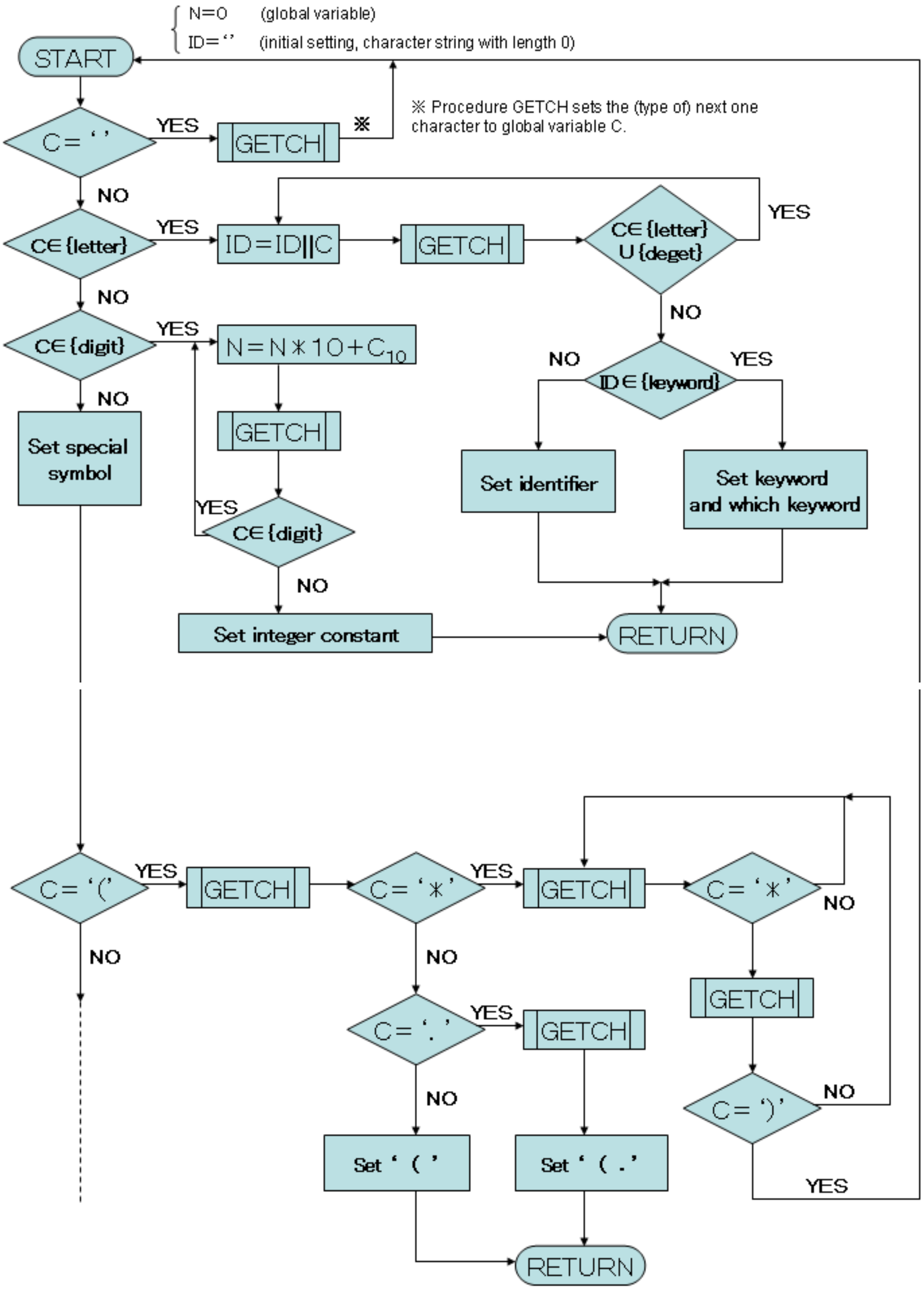


Figure 1.12 part of flow chart for word analysis part

Input	Next one character (Variable C in the flow vhart)	
Output	Class of clipped word	
	Class	Sub-class
	(1) Keyword (2) Identifier (3) Integer constant (4) Character constant (5) Special symbol	Which keyword? (What number?) Its identifier (Value of variable ID in the flow chart) Its value (Value of variable N in the flow chart) Its value Which special symbol? (What number?)

Note: (1) and (5) can be bundled together

Table 1.2 Input/output of word analysis part

1.6.2 Design and management of various tables

Various tables are used to design a language processor including keyword table, instruction code table, and identifier table. Tables include those with fixed entries (ex. keyword table), and those with increasing entries (ex. identifier table).

Because, on a practical processor, data structure and searching method of tables significantly affect performance of the processor, designing a processor should take into consideration natures of tables and given conditions (ex. storage capacity). Some of techniques used are 'hashing technique' and 'tree-searching.'

For the language in this experiment, we use simply-structured tables about keyword, identifier, array name, and block.

(1) Identifier table

Designing identifier table, which shows attributes of each identifier, is especially important because it is quite often referred during compiling. Data structure of identifier table can be constituted with an array of structures or a structure made into a tree-form by pointers. Here I illustrate an example of simple array of structures for reference, but each of you can device your own design. Fields of each entry on an identifier table are;

name	link	object	type	reference	normal	lev	adr

- 'name' is an identifier name
- 'link' is involved in block processing, and it suggests to connect identifiers declared in the same block by pointer. It should indicate the previous variable declared in the same block, and the first variable should indicate a particular region. (Corresponds to NIL.)
- 'object' is a kind of identifier. It shows 0: Constant identifier, 1: Type identifier, 2: Variable identifier, 3: Procedure identifier, and 4: Function identifier.

- ‘type’ indicates kinds of types including 0: procedure, 1: integer, 2: character, and 3: array.
- ‘reference’ indexes array table when an identifier is array identifier, and indexes block table when procedure identifier or function identifier.
- ‘normal’ distinguishes either 0: parameter to access indirectly, or 1: normal variable to access directly.
- ‘lev’ indicates the block level declared.
- ‘adr’ shows the value when it is integer, necessary storage capacity when type identifier, relative address within block when variable, and the first address of instruction codes of the routine when it is a procedure or function identifier.

(2) Array table has the following fields.

element type	element ref	high	elsize	vsize

- ‘element type’ indicates types of element.
- ‘element ref’ indicates index for the array table when element is array (array more than two dimensions).
- ‘high’ indicates the upper limit of subscripts.
- ‘elsize’ is the storage size of elements.
- ‘size’ is storage size for the whole of array. size = elsize * high.

(3) Block table is has the following fields.

last	last param	paramsize	vsize

- ‘last’ indicates the index of the last identifier on the identifier table declared in the block.
- ‘lastparam’ indicates the index of the last parameter on the identifier table.
- ‘paramsize indicates the storage size of parameter in the block.
- ‘vsize’ indicates the size +4 of the local variable area of the block.

(4) Other

Because a block has a nest structure, it is necessary to know to which block you can access from the current position during compiling. You should prepare an array called ‘blockindex’, and store the block table index of the ‘i’ level block accessible from the current position into the ‘i’th position of the array during compiling. With this blockindex and link information on the identifier table, the scope of identifiers will be processed correctly.

[Exercise 1.12] Read the following program. This program outputs a given character string with the length 6 in a backward order.

```
PROGRAM    EXAMPLE3 ;
CONST    SIX=6;
TYPE    ID=ARRAY ( . 6 . ) OF CHAR;
VAR    NAME:ID;    I:INTEGER;    CH:CHAR;
BEGIN
    FOR    I:=1 TO SIX DO
            CALL    READC ( NAME ( . I . ) ) ;
    FOR    I:=1 TO 3 DO
            BEGIN    CH:=NAME ( . SIX+1-I ) ;
                     NAME ( . SIX+1-I . ) :=NAME ( . I . ) ;
                     NAME( . I . ) :=CH
            END;
    FOR    I:=1 TO SIX DO CALL WRITEC ( NAME ( . I . ) ) ;
    CALL WRITELN
END.
```

I illustrate below how identifier table, array table, and block table will look like in this case. Read them carefully. Figure 1.13 shows how the stack looks like.

	name	link	object	type	reference	normal	lev	adr
1	EXAMPLE 3	0	3	0	1	1	0	0
2	SIX	0	0	1	0	1	1	6
3	ID	2	1	3	1	1	1	6
4	NAME	3	2	3	1	1	1	4
5	I	4	2	1	0	1	1	10
6	CH	5	2	2	0	1	1	11

array table

	element type	element ref	high	elsize	size
1	2	0	6	1	6

block table

	last	lastparam	paramsize	vsize
1	6	0	0	12

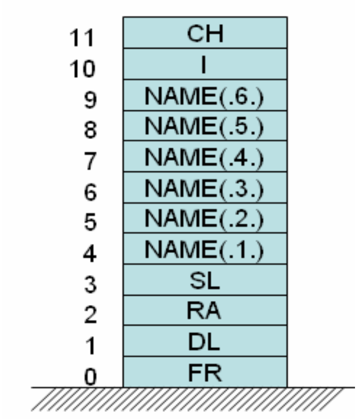


Figure 1.13 Memory allocation on a stack

(Exercise 1.16) Draw identifier table, array table, and block table respectively for programs in Example 1.1, Example 1.2 and Exercise 1.3. Also, illustrate stack storage allocation.

(Exercise 1.17) Is translation with the 1-pass method possible in the real sense? Consider the way to decide the value of 'adr field' on an identifier table concerning procedure identifier.

1.6.3 Syntactic analysis and object code generation

Because we are adopting the recursive descent method, it is enough here to do coding (see sections 1.5.5 and 1.4.3) by referring syntax chart and correspondence rules of syntactic analysis – code generation program.

In the following, I explain specifically the creation (addition, updating) of tables by language processor and code generation by referring the tables, by using the source program in the Example 1.1 as an example. I regard, in explanation and figure below, the existence of a procedure with the same name as the one which appeared in Figure 1.2 to process each non-terminal symbol. Also, there is a couple of rules fixed in the following examples of table;

wavy line: value has changed
(?): value is not determined
(-): never be referred

Let's start explaining generation of tables step by step.

(1) Status of tables at the point of registering 'EXAMPLE1' as a program identifier (block level = 0)

block table

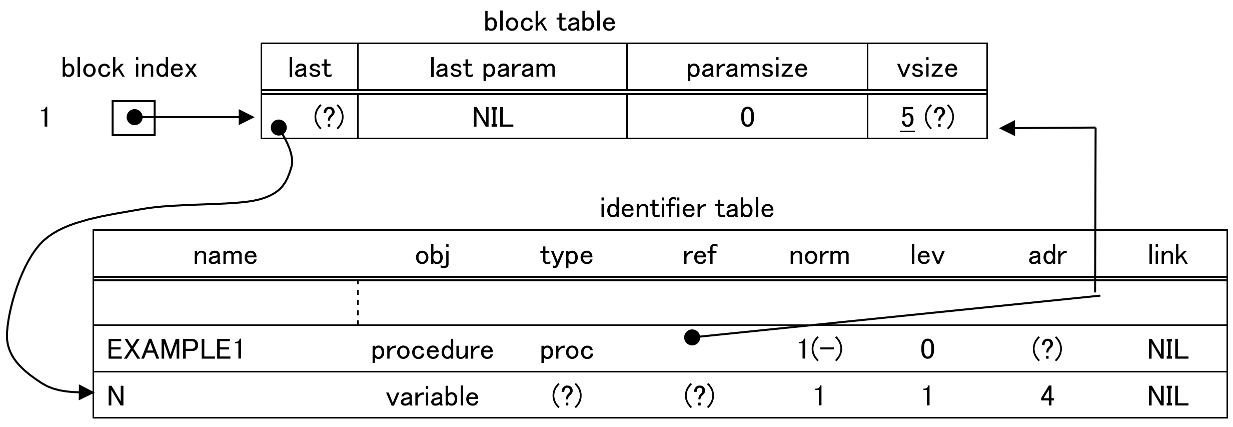
last	last param	paramsize	vsize
NIL(?)	NIL(?)	0 (?)	4 (?)

identifier table

name	obj	type	ref	norm	lev	adr	link
※							
EXAMPLE1	procedure	proc	●	1(–)	0	(?)	NIL

※ It is allowed to register basic identifiers (ex. basic procedural identifier) to the identifier table by default.

(2) Status of tables at the point when the procedure VARDCL registered ‘N’
(block level = 1)



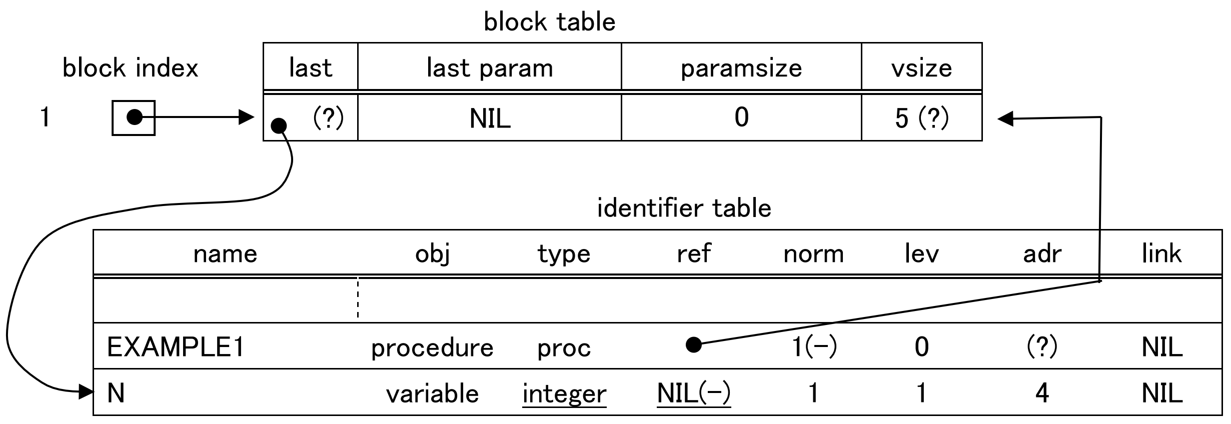
block table

block index	last	last param	paramsize	vsize
1 ●	● (?)	NIL	0	5 (?)

identifier table

name	obj	type	ref	norm	lev	adr	link
EXAMPLE1	procedure	proc	●	1(−)	0	(?)	NIL
N	variable	(?)	(?)	1	1	4	NIL

(3) Status of tables at the point when the procedure VARDCL registered ‘integer-form’ returned by TYPE (block level = 1)



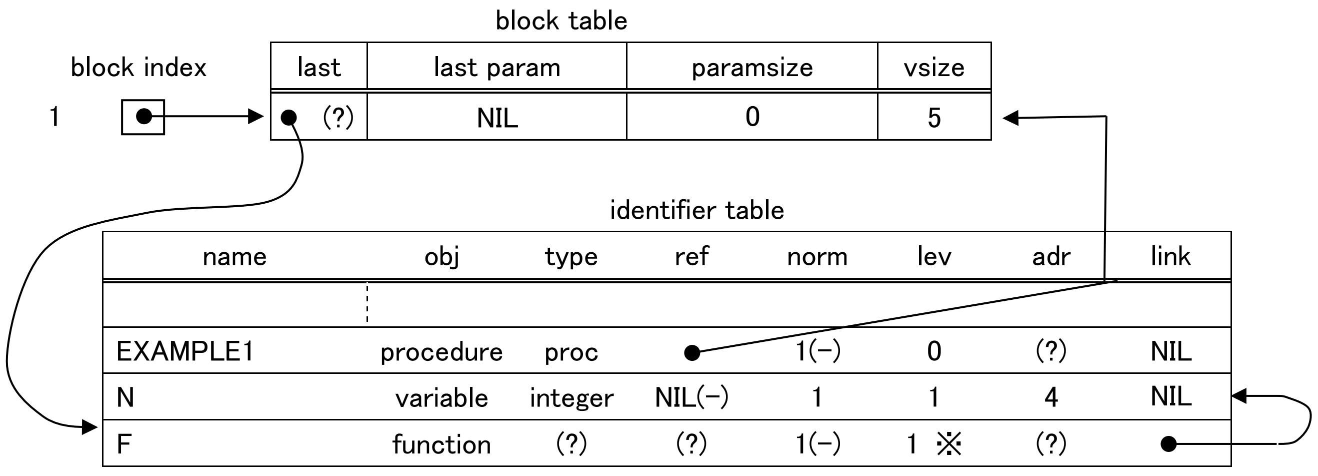
block table

block index	last	last param	paramsize	vsize
1 ●	● (?)	NIL	0	5 (?)

identifier table

name	obj	type	ref	norm	lev	adr	link
EXAMPLE1	procedure	proc	●	1(−)	0	(?)	NIL
N	variable	integer	NIL(−)	1	1	4	NIL

(4) Status of tables at the point when the procedure FUNCDCL registered ‘F’
(block level = 1)



block table

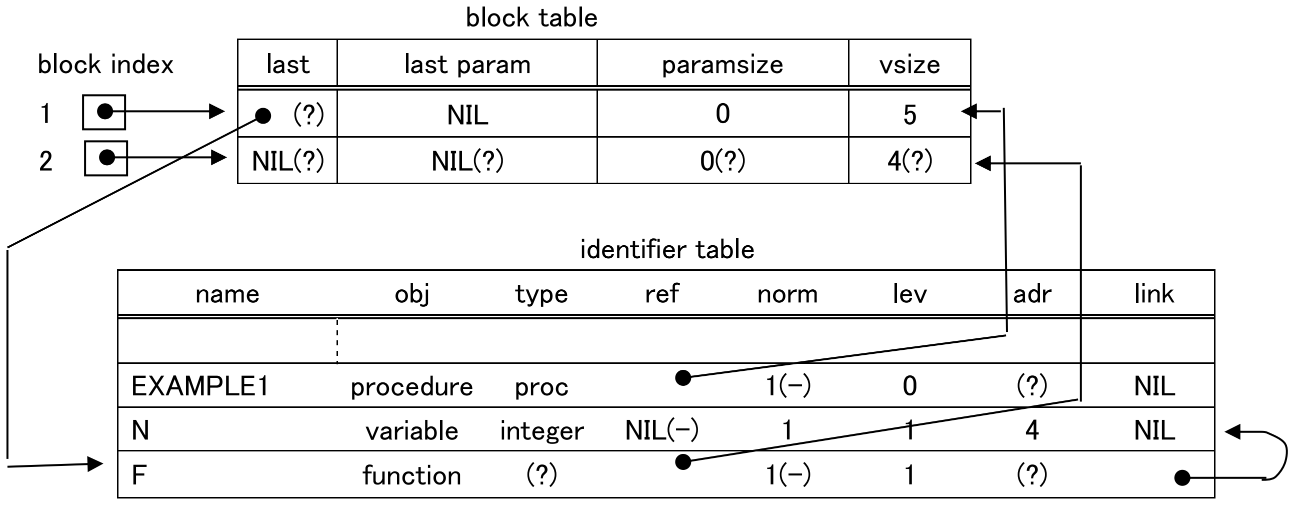
block index	last	last param	paramsize	vsize
1 ●	● (?)	NIL	0	5

identifier table

name	obj	type	ref	norm	lev	adr	link
EXAMPLE1	procedure	proc	●	1(−)	0	(?)	NIL
N	variable	integer	NIL(−)	1	1	4	NIL
F	function	(?)	(?)	1(−)	1 ※	(?)	●

※ In terms of the scope of identifier, the block level of function F is 1. In other words, it is linked to a variable ‘N’ as it belongs to the block of the program EXAMPLE1. In this sense, the level of function F becomes 1. However, the FR area to which the result of function F is stored can be regarded as a variable of block level 2, since it is reserved and released at the same time as the local area of function F. Then, from the latter’s standpoint, the level of function F is 2. In this example, I regard the level as 1 from the former reason.

(5) Status of tables at the point when the procedure FUNCDCL increased the block level by referring ‘ (’ (block level = 2)



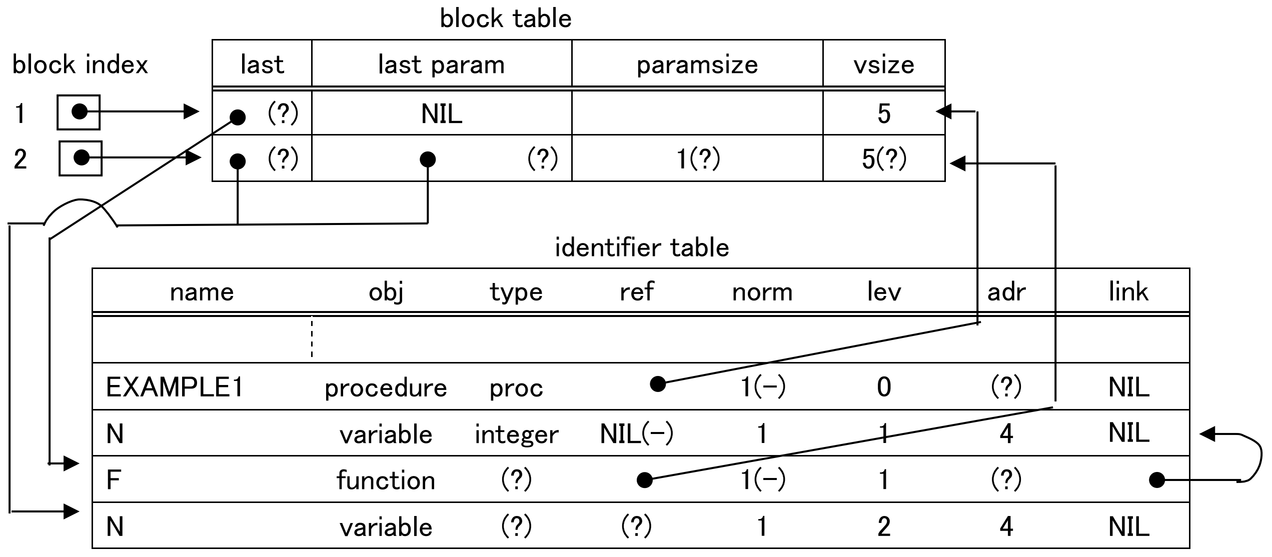
block table

block index	last	last param	paramsize	vsize
1 ●	● (?)	NIL	0	5
2 ●	NIL(?)	NIL(?)	0(?)	4(?)

identifier table

name	obj	type	ref	norm	lev	adr	link
EXAMPLE1	procedure	proc	●	1(−)	0	(?)	NIL
N	variable	integer	NIL(−)	1	1	4	NIL
F	function	(?)	●	1(−)	1	(?)	●

(6) Status of tables at the point when the procedure PARAMLIST registered ‘N’ (block level = 2)



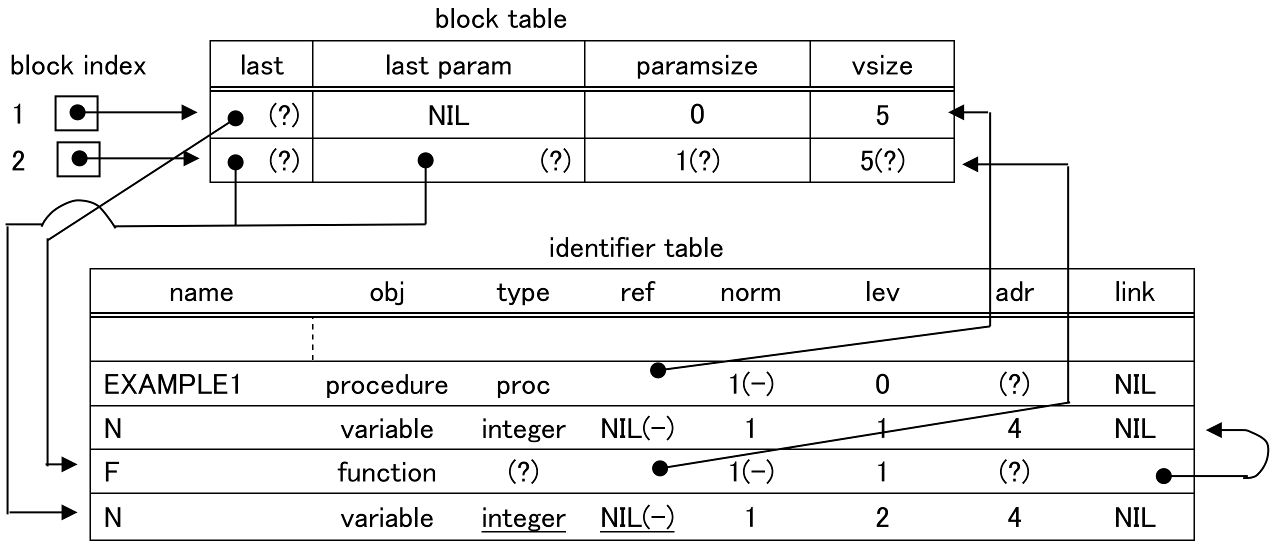
block table

block index	last	last param	paramsize	vsize
1 ●	● (?)	NIL		5
2 ●	● (?)	● (?)	1(?)	5(?)

identifier table

name	obj	type	ref	norm	lev	adr	link
EXAMPLE1	procedure	proc	●	1(−)	0	(?)	NIL
N	variable	integer	NIL(−)	1	1	4	NIL
F	function	(?)	●	1(−)	1	(?)	●
N	variable	(?)	(?)	1	2	4	NIL

(7) Status of tables at the point when the procedure PARAMLIST registered ‘integer-form’ returned by BASICTYPE (block level = 2)



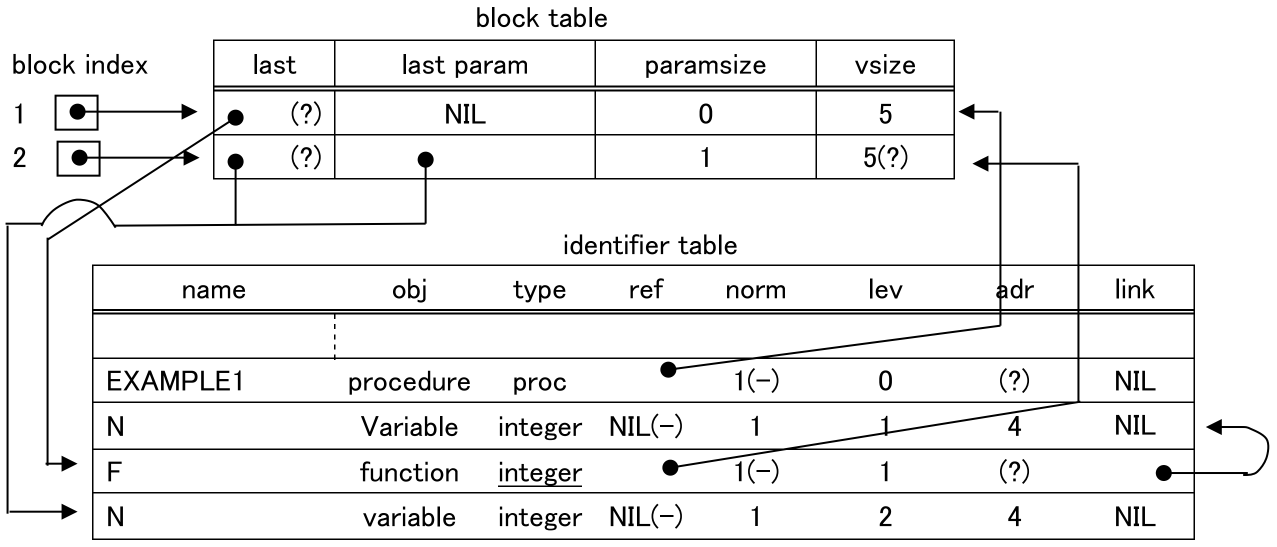
block table

last	last param	paramsize	vsize
● (?)	NIL	0	5
● (?)	● (?)	1(?)	5(?)

identifier table

name	obj	type	ref	norm	lev	adr	link
EXAMPLE1	procedure	proc	●	1(−)	0	(?)	NIL
N	variable	integer	NIL(−)	1	1	4	NIL
F	function	(?)	●	1(−)	1	(?)	●
N	variable	integer	NIL(−)	1	2	4	NIL

(8) Status of tables at the point when the procedure FUNCDCL registered ‘integer-form’ returned by BASICTYPE (block level = 2)



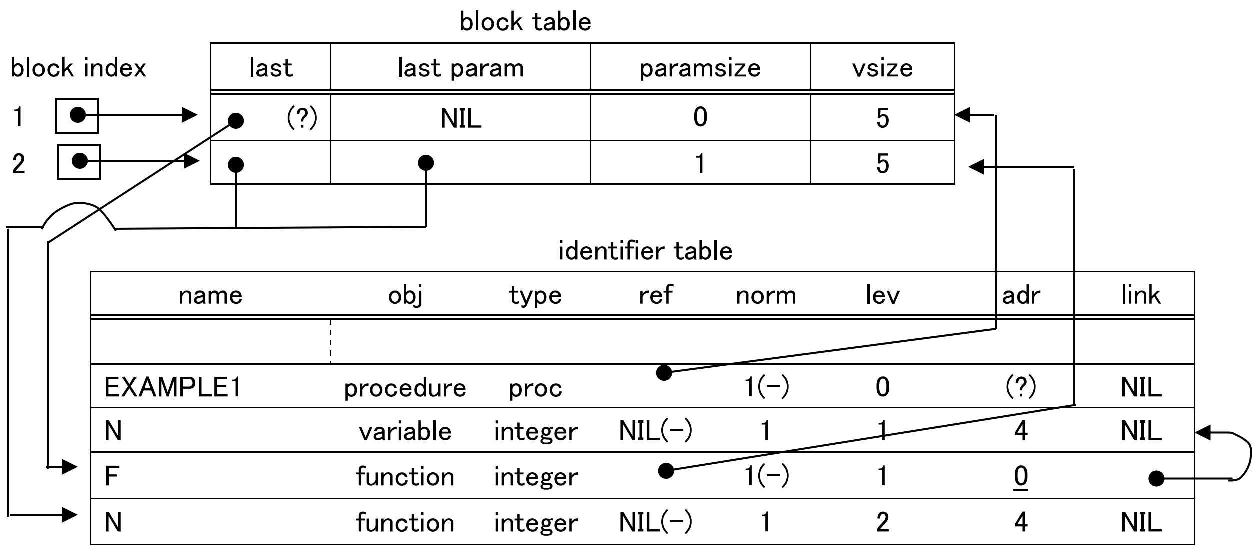
block table

last	last param	paramsize	vsize
● (?)	NIL	0	5
● (?)	●	1	5(?)

identifier table

name	obj	type	ref	norm	lev	adr	link
EXAMPLE1	procedure	proc	●	1(−)	0	(?)	NIL
N	Variable	integer	NIL(−)	1	1	4	NIL
F	function	integer	●	1(−)	1	(?)	●
N	variable	integer	NIL(−)	1	2	4	NIL

(9) Status of tables at the point when the procedure BLOCK registered the starting address of object code of function F by referring 'BEGIN' (block level = 2)



block table

block index	last	last param	paramsize	vsize
1 ●	● (?)	NIL	0	5
2 ●	●	●	1	5

identifier table

name	obj	type	ref	norm	lev	adr	link
EXAMPLE1	procedure	proc	●	1(−)	0	(?)	NIL
N	variable	integer	NIL(−)	1	1	4	NIL
F	function	integer	●	1(−)	1	0	●
N	function	integer	NIL(−)	1	2	4	NIL

You may have, from above examples, a much clear understanding about generation of each table and its usage by now. To give more details, in the following, I explain again works of fields in each table giving focus on their usage within a language processor.

(1) Identifier table

Identifier table is to store information about already-appeared identifiers

(a) 'name' field

Function: Stores names of identifier

Referred example: To check whether an identifier has already been declared or not (checking double-declaration)

(b) 'link' field

Function: Pointer field to form wave-form list of identifiers declared in the same block

Referred example: To check whether an identifier is declared in any block, or to track back to the entry of the object parameter through this link for enough times in order to obtain information of parameter

(c) 'object' field

Function: Indicates kinds of identifier. Semantics of each field on the identifier table changes slightly due to this field

Referred example: To check if the relevant identifier is 'constant identifier', 'type identifier', 'var identifier', 'procedure identifier', or 'function identifier'.

(d) ‘type’ field

Function: Indicates attribute of type of identifier

Referred example: To check types of function and variable

(e) ‘reference’ field

Function: Pointer for array table or block table

Referred example: To refer array table to conduct detailed type check for array identifier, or to provide sizes of elements needed for code generation to determine the address of an element of array. To access to block table in order to find out information of parameter for identifiers of procedure/function, or to find out sizes of local area and parameter area needed for (object) code generation.

(f) ‘normal’ field

Function: Indicates whether the relevant variable is storing addresses for indirect access or is retaining values for direct access

Referred example: To determine addresses of the relevant variable is determined by generating either ‘LA’ or ‘LV’ when you need them for code generation. Similarly, when you decide a needed value to generate codes is either of only ‘LV’ or of both ‘LV’ and ‘LI’.

(g) ‘lev’, ‘adr’ field

Function: Indicates the block level of the relevant variable and its relative address from base

Referred example: In case of code generation, operand p is calculated by (current block level) – (lev field of the relevant variable). Also, the operand q which is generated with it is given as (adr field of the relevant variable). In case of the area FR which retain results of function.

(operand p) = (current block level) – (<u>block level of the relevant variable</u>) – 1

//

(lev field of the relevant variable)

(operand q) = 0

See the note for (4) in the above explanation on status of tables.

(2) Array table

This table stores specific information about array.

(a) element type } same as type and ref (for array) on an identifier table
(b) element field

(c) high field }
(d) elsize field

Referred example: when generating codes to calculate addresses of array elements

(e) size field

Referred example: To determine the relative address of the variable declared right after this type of array by finding the size of the relevant array, or to determine the size of local area of the block which includes declaration of this type of array

(3) Block table

This table stores information about each block.

(a) 'last' field

Function: Creates a wave-line list of identifiers table within the relevant block, together with link field on the table

Referred example: To detect whether an identifier in the relevant block or not

(b) 'last param' field

Function: Creates a wave-line list of parameters though similar to the last field

Referred example: To track parameters of the relevant block in sequence

(c) 'param' field

Function: Indicates the number (size) of parameters of the relevant block

Referred example: To check its relation to the number of arguments

(d) 'vsize' field

Function: Indicates the storage area size of the relevant block which should be secured on the stack

Referred example: When generating codes for 'INT' order at the beginning and the end of the body of procedure/function

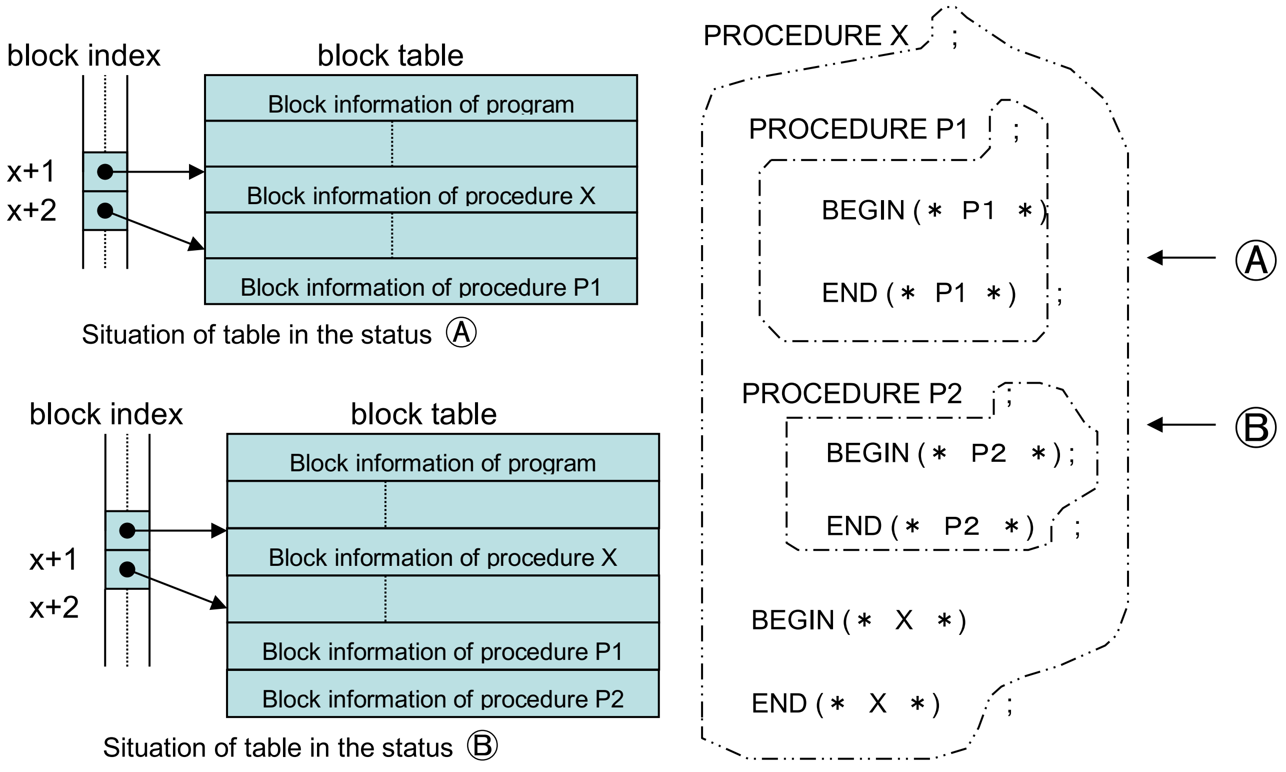
(4) Block index

This table identifies an entry of the block table which possesses information of referable blocks at statically shallower levels including the current block level in order to implement rules for scope of an identifier. (See Figure 1.14)

As shown in the figure, it is necessary to secure entry information for the procedure P1 on the block table even in the status Ⓑ. This is because, for instance, that information is necessary when P1 is called out within P2. Even when P1 does not have parameter, such information -- there is no parameter -- is still important.

Therefore, what is not needed in the status Ⓑ is information on the identifier table about identifiers which were declared within the block of the procedure P1, other than parameters of the procedure P1 (or after those parameters, in terms of the structure of table). In general, for any procedure/function, information about identifier declared within its own block (including information of parameters existing on the block table or on the identifier table of procedure/function declared within its own block) become unnecessary when compiling of the body of its own block finishes.

(Exercise 1.18) A source program should be thought to include various errors. Consider how to respond to errors on a language processor.



block index
block table
x+1
x+2
Block information of program
Block information of procedure X
Block information of procedure P1
Situation of table in the status Ⓐ
block index
block table
x+1
x+2
Block information of program
Block information of procedure X
Block information of procedure P1
Block information of procedure P2
Situation of table in the status Ⓑ
PROCEDURE X ;
PROCEDURE P1 ;
BEGIN (* P1 *)
END (* P1 *) ;
PROCEDURE P2 ;
BEGIN (* P2 *);
END (* P2 *) ;
BEGIN (* X *)
END (* X *) ;
Ⓐ
Ⓑ

Suppose the block level of procedure X is x, the block level inside double-dotted line is (x+1) and that inside dotted line is (x+2).

Figure 1.14 Change in block level and relations between block index and block table

1.6.4 Designing interpreter

The language processor in this experiment generates object codes for a stack calculator by reading a source program from files and by syntactic analysis. Object code is a machine language for virtual stack calculator. In order to process this language with an actual calculator, it is necessary to design an interpreter to simulate the machine language of stack calculator. It is enough for interpreter to simulate the stack calculator in the section 1.4 without any modification according to its specification. Creating an interpreter is quite simple. When there are problems during program execution like devide error and stack overflow, you have to output information. It is also useful as an option for debug to give a function to print running pc, instruction code, ‘t’, ‘s[t]’, and so on. However, adding the tracing facility significantly will lower the performance of program execution.

1.6.5 Important points for designing

In the process of actual programming, you have to start with as systematic procedure as possible and with well-considered designing before starting coding actually. It is important to keep close contact among programmers, and to habitually document interfaces between programs as specific as you can. Assignment of works has to be done carefully so that every member can deepen his/her understanding of important issues to study. This experiment also requires considering the process of program creation.

(Study 1.1) Try to extend the language specification of KPL referring other languages such as PASCAL.

(Study 1.2) Consider the optimization of object codes.

(Study 1.3) Consider designing by the micro-program method of stack calculator. It is possible to execute KPL by simulation of micro-codes.

References

[1] Wirth, N., *Arugorizumu + Deeta Kouzou*, Trans. by Takuya Katayama, Japan Computer Association, September 1979 (Wirth, Niklaus, *Algorithm + Data Structure = Programs*, NJ: Prentice Hall, 1976)

[2] Doushita, Shuuji (ed.), *Jouhoushoririron Senmonyougoshu* [Technical glossary for information processing theory], Denshi Tsuushin Gakkai, March 1982

[3] Hopcroft, J. E., and Ullman, J. D., *Gengoriron to Ootomaton*, Trans. by Akihiro Nozaki and Izumi Kimura, Saiensu-sha, June 1971 (Hopcroft, John. E., and Ullman, Jefery. D, *Formal Language and Their Relation to Automata*, Addison-Wesley Pubishing, 1969)

[4] Fujio, Nishida. *Gengojouhoushori* [Language information processing], Korona-sha, June 1981

[5] Ikuo, Nakata. *Konpaira* [Compiler], Sangyou-tosho, November 1981

[6] Earley, J., “An Efficient Context-free Parsing Algorith”, *Comm. ACM*, 13-2, Feb., 1970, 94-102