2. Definition and Implementation of a Small Language

2.1 Syntax and Semantics

The syntax of a language defines a correct ordering of symbols to form a correct sentence of the language. The semantics of the language defines the meaning of a sentence. A rigorous formalism is possible for syntax through the concept of context-free grammars, whereas a complete formalism is still difficult for semantics. There are three approaches to semantics; operational, axiomatic and denotational. We take an operational approach in this note, which describes the meaning of the language using a hypothetical machine and its behaviour corresponding to each symbol in the language. An axiomatic semantics uses logical axioms in mathematical logic, whereas a denotational semantics uses the concept of functions in mathematics.

2.2 Context-Free Grammar (CFG) and Backus-Naur Form (BNF)

A context-free grammar G=(N, T, P, S) is defined as follows:

N : a finite set of non-terminal symbols (or simply non-terminals)
T : a finite set of terminal symbols (or simply terminals)
P : a finite set of rewriting rules (or simply rules or productions)
S : the starting symbol in N.

Here we define P in more detail. Let $V = N \cup T$ be the set of vocabulary. Let V^* be the set of all finite strings made from V including the empty string e. The length of a string x is denoted by lg(x). The concatenation of two strings x and y is denoted by xy. Note that lg(xy) = lg(x) + lg(y).

Example x=ab, y=bab. Then xy = abbbab. Note that lg(xy) = 5 = 2 + 3 = lg(x) + lg(y).

Strings are sometimes called words. The elements of each set follow some conventions as follows:

Elements in N : A, B, C, ...	Upper-case letters in the alphabet
Elements in T : a, b, c, ...	Lower-case letters in the first part of the alphabet
Elements in V^* : α , β , γ , ...	Lower-case letters in the Greek alphabet
Elements in T^* : x, y, z, ...	Lower-case letters in the last part of the alphabet

The set P consists of rules of the form $A \rightarrow \alpha$, meaning that A produces (or is rewritten by) α .

Example G = (N, T, P, S)

N = {S}, T = {a, b}, P = {S $\rightarrow$ aSb, S $\rightarrow$ ab}, S = S

We write $\alpha \Rightarrow \beta$ for α and β in V^* if an application of a rule in α produces β. We also say that α generates β. We write $\alpha \ ^*\Rightarrow \beta$ if there are $\alpha_0, ..., \alpha_n$ such that $\alpha_0=\alpha$, $\alpha_n=\beta$ and $\alpha_i \Rightarrow \alpha_{i+1}$ for i=0, ..., n-1. In this case also we say that α generates or produces β. The language generated by G, denoted by L(G), is defined by

$$L(G) = \{ x \text{ in } T^* \mid S \ ^*\Rightarrow x \}.$$

We say such a language is a context-free language (CFL). We sometimes give a CFG by just rewriting rules whenever it is clear from context.

Example

1. $S \rightarrow aSb$
2. $S \rightarrow ab$

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow ... \Rightarrow a^{n-1}Sb^{n-1} \Rightarrow a^nb^n$$

We can generate a^nb^n by using the first rules n-1 times and then the second rule once. Thus we have $L(G) = \{ a^nb^n \mid n \geq 1 \}$.

A generation process can be visualised by a tree, called a syntax tree or derivation tree.

Example. Non-terminals are enclosed in angular brackets.

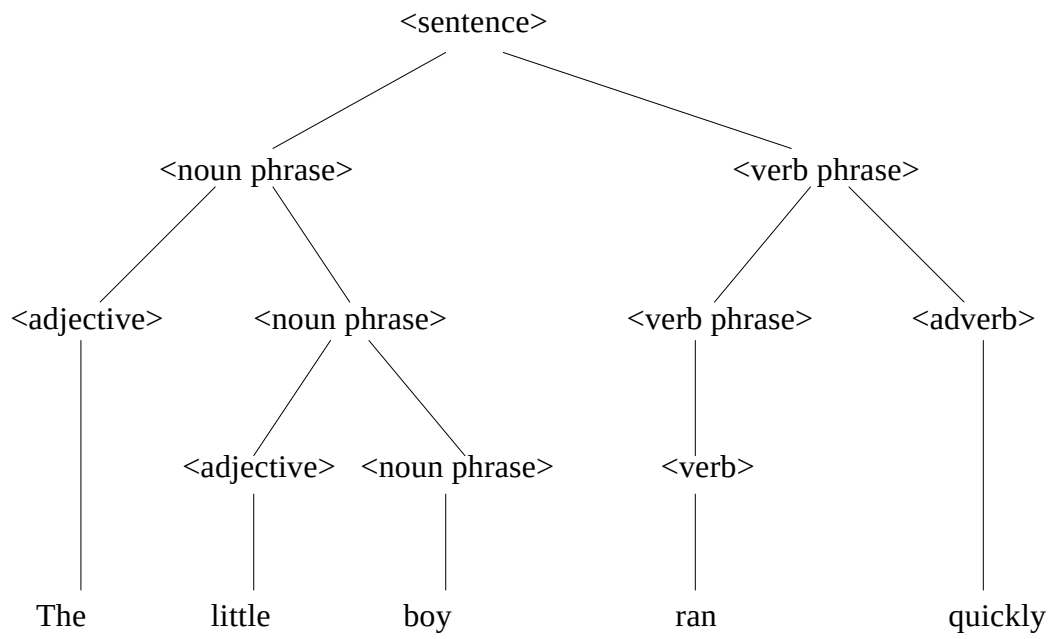
<sentence> → <noun phrase> < verb phrase>
<noun phrase> → <adjective> <noun phrase>
<verb phrase> → <verb phrase> <adverb>
<noun phrase> → <noun>
<verb phrase> → <verb>
<noun> → boy
<adjective> → The
<adjective> → little
<verb> → ran
<adverb> → quickly

The sentence “The little boy ran quickly” can be generated as follows:

<sentence> ⇒ <noun phrase> <verb phrase>
⇒ <adjective> <noun phrase> <verb phrase>
⇒ The <noun phrase> <verb phrase>
⇒ The <adjective> <noun phrase> <verb phrase>
⇒ The little <noun phrase> <verb phrase>
⇒ The little <noun> <verb phrase>

⇒ The little boy <verb phrase>
⇒ The little boy <verb phrase> <adverb>
⇒ The little boy <verb> <adverb>
⇒ The little boy ran <adverb>
⇒ The little boy ran quickly

In the above generation, we applied a rule in each intermediate form, which is also called a sentential form, to the leftmost non-terminal. This kind of derivation is called a leftmost derivation. A rightmost derivation is similarly defined. A leftmost derivation or right most derivation correspond to a derivation tree one-to-one. See below.



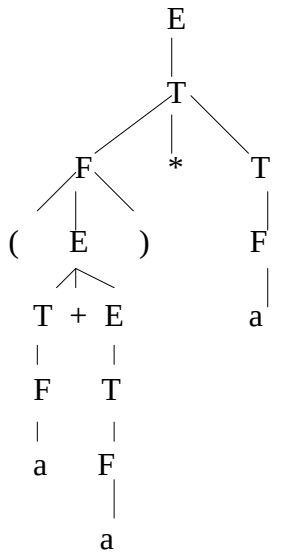
The leftmost derivation corresponds to the leftmost depth-first traversal of the tree. To find a derivation tree from the given sentence (or a string) is called **syntax analysis**. To know whether a given string is in the language L(G) for a given grammar is called recognition. To recognise a string often needs syntax analysis. The identification of each word such as “The little”, “little”, etc. is called **lexical analysis**. In programming languages, these lexical entities correspond to reserved words such as “if”, “begin”, etc.

The definition of the syntax of a programming language by a context-free grammar is called a Backus-Naur form, in which the symbol “::=“ is often used instead of “→ “ to rewrite the left-hand side by the right hand side.

Example Arithmetic expressions with “*” and “+” are generated by the following.

E → T + E | T
T → F * T | F
F → (E) | a

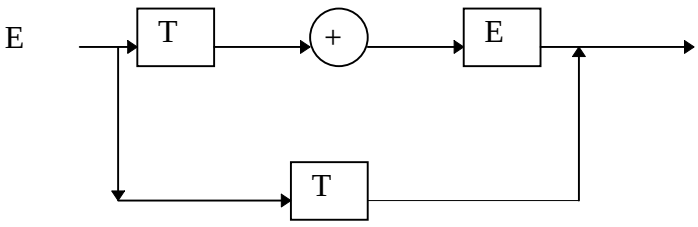
In these rules, we used simplified notations with vertical bars "|". The notation, for example, E → T + E | T stands for two rules E → T + E and E → T. The syntax tree for the string (a+a)*a is given below.

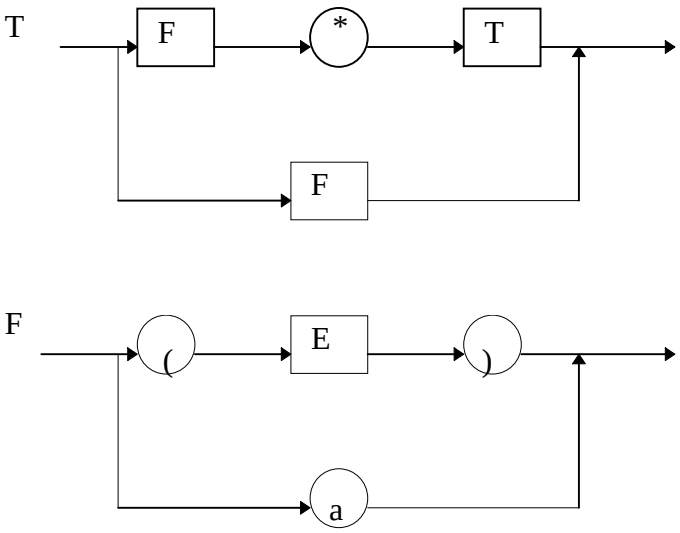


2.3 Syntax Chart and its Analysis

A syntax chart is equivalent to a context-free grammar. It is a collection of directed graphs, each corresponding to a non-terminal. There are two kinds of nodes in those graphs. One corresponds to a non-terminal expressed by a box, and the other to a terminal expressed by a circle. There is one entry arc and one exit arc in each graph. To enter a non-terminal node, that is a box, is to enter the graph corresponding to the non-terminal. If we go through a terminal node, that is a circle, is to generate the corresponding terminal symbol. The string generated by traversing the syntax chart from the graph corresponding to the starting symbol is a string in the language generated by the grammar given by the syntax chart. If there is a forking point, we can take any branch. The language L(A) generated from a non-terminal of the grammar is the set of strings generated by starting from the graph corresponding to non-terminal A.

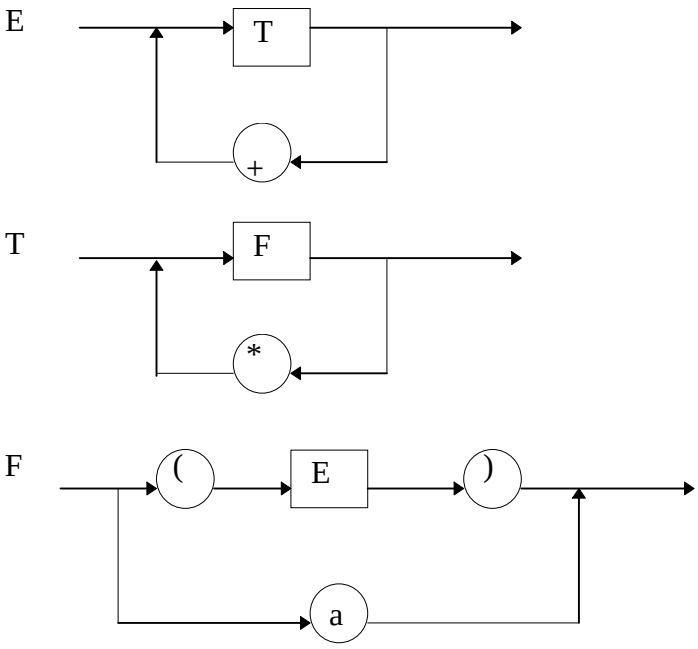
Example The previous example is shown by a syntax chart





By introducing loop structures, we can simplify syntax charts.

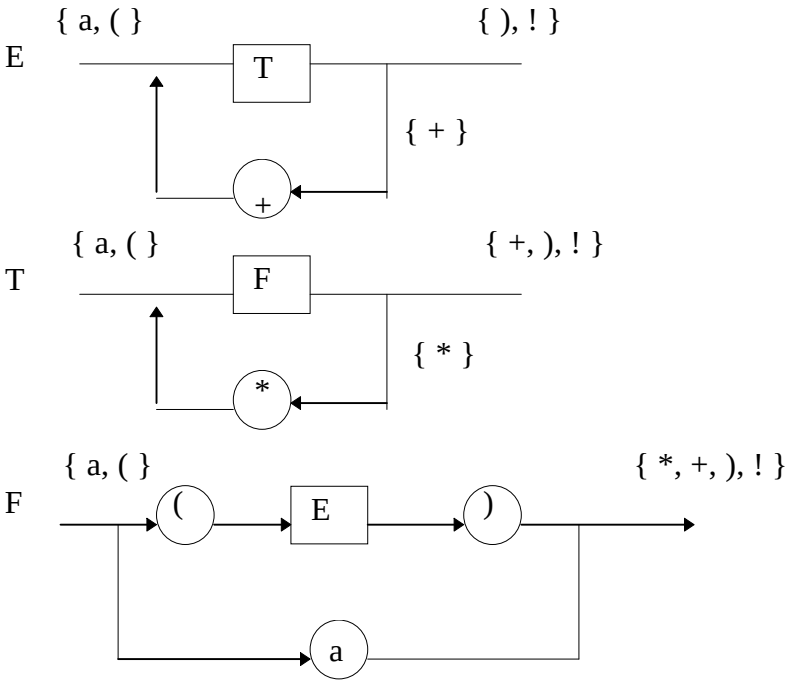
Example The previous example is simplified into the following.



This simplification is done by observing that E generates T + T + ... + T and T generates F * F * ... * F.

Now how can we know that (a+a)*a is generated by the above grammar given by a syntax chart? For simplicity we give an end marker "!" at the end of each string. The set first(A) of non-terminal A is the set of terminals at the beginning of any terminal strings generated from A, that is, a string in L(A). The set follow(A) is the set of terminals in a string in the language that may follow any string in L(A). The selection set of a branch at a forking point is the set of terminals that are obtained from the downstream. Selection sets are calculated by tracing back each arrow in the chart . When we enter a box backward, we get to the exit of the corresponding graph with the set at hand. When we get out of a graph for A backward through the entry point, we bring the set first(A) at each occurrence of A in the chart. When we trace back through a forking point, we take a union of selection sets associated with the branches.

Example. The syntax chart in the previous example is now enhanced with selection sets, first sets and follow sets given below.



first(E) = first(T) = first(F) = { a, (}
follow(E) = {), ! }
follow(T) = { +,), ! }
follow(F) = { *, +,), ! }

Now we describe recursive descent parsing. As we read the given input string, we traverse the syntax chart from the starting graph. If we have a match between the current input symbol and the terminal symbol encountered in the chart, we consume the input symbol and go through the circle in the chart. If we come to a box, we enter the graph for the corresponding non-terminal. If we come to a forking point, we go to a branch with

the selection set to which the current input symbol belongs. If we can not proceed any more, we signal an error message.

If the selection sets at any forking point are disjoint, a recursive descent parsing is possible. It is not possible to parse all context-free grammars in a recursive descent manner, but many programming languages are designed in such a way that recursive descent parsing is possible.

Example The string (a+a)*a is analysed through the syntax chart with the histories of entered non-terminals and current input strings. Consumed parts of input strings are not shown.

History	Input	Comments
E	(a+a)*a!	enter T
ET	(a+a)*a!	enter F
ETF	(a+a)*a!	consume (
ETF	a+a)*a!	enter E
ETFE	a+a)*a!	enter T
ETFET	a+a)*a!	enter F
ETFETF	a+a)*a!	consume a
ETFETF	+a)*a!	exit from F
ETFET	+a)*a!	exit from T
ETFE	+a)*a!	consume +
ETFE	a)*a!	enter T
ETFET	a)*a!	enter F
ETFETF	a)*a!	consume a
ETFETF	)*a!	exit from F
ETFET	)*a!	exit from T
ETFE	)*a!	exit from E
ETF	)*a!	consume)
ETF	*a!	exit from F
ET	*a!	consume *
ET	a!	enter F
ETF	a!	consume a
ETF	!	exit F
ET	!	exit from T
E	!	exit from E
null	!	signal “no error”

For a syntax chart that can be analysed in a recursive descent manner, we can write a recursive program for syntax analysis. The graph for each non-terminal corresponds to a procedure and a box corresponds to a procedure call. A circle corresponds to the read operation.

Example The recursive program for the previous syntax chart is shown below. The variable current shows the current input symbol. The procedure getsym reads the next input symbol. The procedure error detects an error and abandon the processing after signalling an error.

```
procedure E;
begin
  T;
  while current = "+" do begin
     getsym;
     T
  end;
  if current is in not { ")", "!" } then error
end;

procedure T;
begin
  F;
  while current = "*" do begin
     getsym;
     F
  end;
  if current is not in { "+", ")", "!" } then error
end;

procedure F;
begin
  if current = "(" then begin
    getsym;
    E;
    if current = ")" then getsym else error
  end
  else if current = "a" then getsym else error
end
begun { main program }
  set up the input string;
  getsym;
  E
end.
```

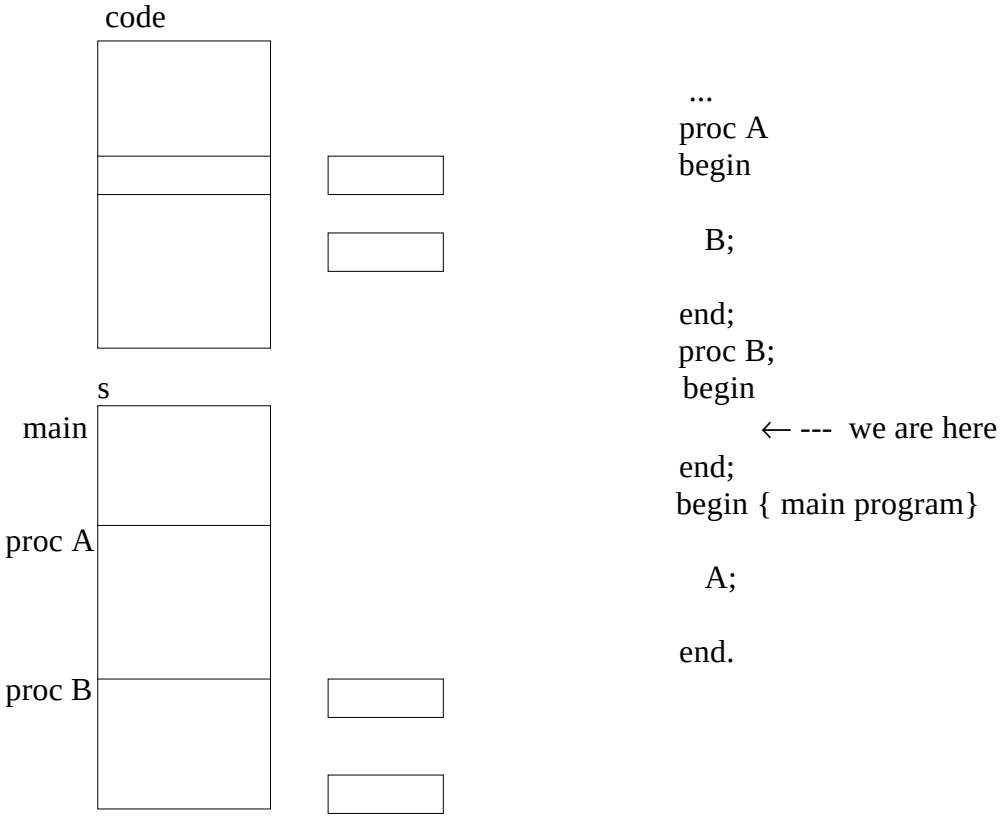
A Pascal code is attached in the next page.

2.4 Definition of a Simple Programming Language -- PL/0

A small language, called PL/0, was developed by N. Wirth in Algorithms + Data-Structures = Programs, Prentice-Hall, 1976, as a subset of Pascal. The following syntax chart allows a recursive descent parsing. The parser follows the syntax chart. The parser does syntax analysis as well as static check of objects declaration and usage. In PL/0, an object is a constant, variable, or procedure. When an object is declared, it is entered into a table with its name and kind. If an object ,which is not declared, that is not in the table, is used in a PL/0 program, an error message is signalled This is called a static check because we can check at compile time before we run the object code. In a higher level language, the type of an object is also checked at compile time. If a variable of integer type is assigned with a floating number, for example, an error is signalled. On the other hand, we can not know if an array index goes beyond the boundaries of the declared array until we actually run the object code. This type of check is called a run time or dynamic check.

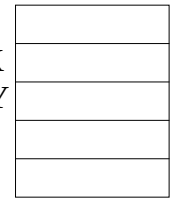
2.5 PL/0 Machine

We generate machine instructions for a hypothetical computer called PL/0 machine when we compile a PL/0 program. It has a program store named “code” and a data store named “s” organised as a stack. These are represented by one-dimensional arrays in the interpreting program (simulator in out terminology). Also there are three registers; the base register b, the top stack register t, and the instruction register i. The program counter is represented by p. See blow.



The data store is divided into data segments, each corresponding to an activation of a block. Each data segment consists of the dynamic link, the static link, the return address, and the area for local variables. If an arithmetic/logical or relational expression is evaluated. the area on top of this data segment is used. For example, if the following program is executed and the right-hand side of the arithmetic expression is being evaluated, we have the following picture.

```
var X,Y;
begin
  X:=10; Y:=20;
  Y:=X+Y;
end.
```



After this 30 is placed to the position of 10, and further in the next step, 30 will be placed in the store for Y

After we finish each execution of procedure, we need to go back to the calling point and recover the environment for the calling block.