where as the following grammars are not S-Grammars.

$$\begin{aligned} S &\rightarrow aABb \\ A &\rightarrow aA \mid b \\ B &\rightarrow bB \mid b \end{aligned}$$

Not an S-Grammar because of terminal b in S-production

$$\begin{aligned} S &\rightarrow aABB \\ A &\rightarrow aA \mid a \\ B &\rightarrow bB \mid b \end{aligned}$$

Not an S-Grammar because of first symbol in both the A-productions is a

Example 5.24: Find a Simple Grammar (S-Grammar) for the regular expression aaa*b + b

For a grammar to be simple grammar, no two production should have the same variable A on the left and same terminal *a* as the first symbol on the right hand side of the production and this terminal should be followed by zero or more variables. So, the resulting grammar can take the form

$$\begin{aligned} S &\rightarrow aA \mid b \\ A &\rightarrow aB \\ B &\rightarrow aB \mid b \end{aligned}$$

If we apply the production

$$S \rightarrow b$$

then from S we get the string b and the derivation for this is

$$S \Rightarrow b$$

So, the string *b* can be obtained successfully from S. The first part of the regular expression aaa*b can be derived using the productions

$$\begin{aligned} S &\rightarrow aA \\ A &\rightarrow aB \\ B &\rightarrow aB \mid b \end{aligned}$$

Note that by applying the first two productions, we get the partial derivation

$$S \Rightarrow aA \Rightarrow aaB$$

using which we have obtained two *a*'s followed by a variable B. It is clear from the B-production that B can generate one or more a's or a *b*. Thus, the required language is generated by the grammar.

Note: The same regular expression can be represented using the following grammar also.

$$S \rightarrow aAB \mid b$$
$$A \rightarrow aA \mid a$$
$$B \rightarrow b$$

But, this is not an S-grammar because both the A-productions have *a* as the first symbol.

Example 5.25: Find a Simple Grammar (S-Grammar) to generate the language $L = \{a^n b^n \mid n \geq 1\}$

The solution to this problem is very easy and the grammar is shown below:

$$S \rightarrow aA$$
$$A \rightarrow aAB \mid b$$
$$B \rightarrow b$$

Note that for the first time the partial string aA is obtained by applying the production

$$S \rightarrow aA$$

Now, the non-terminal A can be replaced by *b* giving the string *ab* or it can be replaced by aAB producing two *a*'s followed by AB as shown:

$$S \Rightarrow aA \Rightarrow aaAB$$

If we apply the A production again we get the derivation of the form

$$S \Rightarrow aA \Rightarrow aaAB \Rightarrow aaaABB$$

It is clear from the above derivation that if *n* number of a's are generated, they are followed by *n* number of non-terminals where the first non terminal is A and the rest are B's. Finally to terminate, A is replaced by b and B's are also replaced by b's generating *n* number of *a*'s followed by equal number of *b*'s. Thus the given grammar produces the language

$$L = \{a^n b^n \mid n \geq 1\}$$

5.8 Application of context free grammars

The various applications of context free grammars are:

Parsers: In the design of programming languages such as the interpreters or compilers, the formal language such as context free language (obtained from CFG) plays a very important role using which very efficient translators can be build. Typical languages uses balanced parentheses which can be easily expressed using CFG (Note: It is not possible to represent using regular expression or FA), The arithmetic and conditional expressions along with various operators can be easily expressed using CFG (see the example 5.22). For example, to define the programming languages, unlike the notations we used earlier to define the productions, a special notation called

BNF (Backus Naur Form) is used. Only the difference is that, all the variables using BNF notations are written within angular brackets '<' and '>' and the terminals without angular brackets. The arrow mark in the production is replaced by the symbols :=. For example, the productions

$$\begin{aligned} E &\rightarrow E+T \mid T \\ T &\rightarrow T*F \mid F \\ F &\rightarrow (E) \mid \mathbf{id} \end{aligned}$$

can be written using BNF notation as shown below:

```
<exression> := <expression> + <term> | <term>
<term>      := <term> * <factor> | factor
<factor>    := (<expression>) | id
```

Let us take some real examples in programming languages where the BNF notations are extensively used.

Example 5.26: Use BNF notation and describe the while statement in C language. Assume that assignment statement and condition for while are defined already.

We know that the syntax of while statement is:

```
while (condition)
{
        statements;
}
```

or

```
while (condition) statement;
```

The BNF notation for the C-while statement is shown below:

```
<whl_stmt>    := while ( <exp> ) <stat_list>
<stat_list>   := <stat>; | { <stat>;<stat_list> }
<stat>        := <assign_stat>
```

Example 5.27: Give the BNF notation to write a C program. Provide 6 or 7 productions generally describing the main program. Assume the rest are defined.

The very abstract BNF notation for C program is shown below:

```
<program>     := main () <block>
<block>       := { <stat_list> }
<stat_list>   :=<stat> | <stat><stat_list> | <dclr> | <dclr><stat_list>
<dclr>        := int <identifier> | char <identifier>
<stat>        := <assign_stat> | <cntrl_stat>
```

```
<assign_stat>   := <identifier> = <exp>
<exp>           := <exp> + <exp> | <exp> * <exp> | ( <exp> )
<exp>           := <exp> - <exp> | <exp> / <exp> | (identifer)
                .
                .
                .
```

and so on.

Using the CFG, it is possible to check only the syntax of a language but, not the semantics of a language. While passing the parameters, we know that the type of *actual parameters* should match with type of *formal parameters.* If the types are different, the syntax still may be correct, but the semantics are wrong. Using the CFG's it is not possible to check whether the statement is semantically correct or not. It is the responsibility of semantic phase to check for the correctness. The details of the compiler construction is not the scope of this book and the reader is recommended to refer the compiler construction for the details. In C language various block structures which are nested using { and } can be easily implemented. The various control statements such as for, if, while etc, can be represented using CFG.

YACC Parser-generator: The UNIX system provides a YACC command using which efficient parsers can be generated. The input to this command is a CFG but represented using different notations. In the notation used which is slightly different from that of notations used in CFG, each production is associated with an *action* which is the C code to be executed when the parse tree is created. For example, the grammar

$$
\begin{aligned}
E &\rightarrow E+E \mid E-E \\
E &\rightarrow E*E \mid E/E \\
E &\rightarrow (E) \mid I \\
I &\rightarrow a \mid b \mid c
\end{aligned}
$$

using YACC notation can be written as shown below:

```
E : I           {.....}
  | E '+' E     {.....}
  | E '-' E     {.....}
  | E '*' E     {.....}
  | E '/' E     {.....}
  | E '^' E     {.....}
  | '(' E ')'   {.....}
  ;
I : 'a'         {.....}
  | 'b'         {.....}
  | 'c'         {.....}
  ;
```

Note the notations used in this representation when compared with notations used in CFG:

1. The symbol '→' is replaced by the symbol ':'
2. All the productions with a given head (i.e., the non-terminals towards left of → is called a head) are grouped together and the body of the corresponding productions are separated by vertical bars and ends with terminator ';'
3. The terminals are enclosed within single quotes and the variables are not enclosed within single quotes.

Markup languages: The most familiar markup language is HTML ((Hyper Text Markup Language). HTML is used to create Hypertext documents for use on the World Wide Web. In HTML a block of text is surrounded with codes that indicate how it should appear on the computer screen. In HTML we can specify that a block of text, or a word, is linked to another file on the Internet. Hypertext Markup Language is the code used to write most documents on the World Wide Web. We can write the code using any text editor.

XML: XML stands for Extensible Markup Language. XML is a system for defining, validating, and sharing document formats. XML uses tags to distinguish document structures, and **attributes**. Instead of concentrating on formatting the text, XML is used to define the semantics.

Exercises:

1. What are the limitations of regular languages?
2. What is a context free grammar? Explain with example.
3. Let G = (V,T,P,S) be a CFG where

   V = { S }
   T = {a, b}
   P = {
   S → aSa | bSb
   }
   S is the start symbol.
   What is the language generated by this grammar?

4. Show that the language $L = \{ b^n \mid m \neq n \}$ is context free.
5. Draw a CFG to generate a language consisting of equal number of a's and b's
6. Draw a CFG on {a, b} to generate a language $L = \{ a^n w w^R b^n \mid w \in \Sigma^*, n \geq 1 \}$.
7. Obtain a context free grammar to generate properly nested parentheses structures involving three kinds of parentheses ( ), [ ] and { }.
8. Obtain a context free grammar to generate the following language
   $L = \{w \mid w \in \{a, b\}^*, n_a(v) \neq n_b(v)$ where v is any prefix of w$\}$
9. Obtain a context free grammar to generate the following language
   $L = \{01(1100)^n 110(10)^n \geq 0\}$
10. Is the following language Context free?
    $L = \{a^n b^n \mid n \geq 0\}$
11. Obtain a context free grammar to generate the following language
    $L = \{a^n b^m : m > n$ and $n \geq 0\}$