Chapitre 2

Parsers

2.1 Language

An *alphabet*, in the context of formal languages can be any set Σ. The elements of an alphabet Σ are called *letters*. A *word* over an alphabet can be any finite sequence of letters. The set of all words over an alphabet Σ is usually denoted by Σ^* (using the Kleene star). For any alphabet Σ there is only one word of length 0, the empty word, which is denoted by ε.

A *language* $\mathcal{L}$ over an alphabet Σ is a subset of Σ^*.

Example 1 *The following rules describe a formal language $\mathcal{L}$ over the alphabet*

$$\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, =\}.$$

(i) *Every nonempty word that does not contain + or = is in $\mathcal{L}$.*
(ii) *The word* 0 *is in $\mathcal{L}$.*
(iii) *A word containing = is in $\mathcal{L}$ if there is exactly one =, and it separates two valid words in $\mathcal{L}$.*
(iv) *A word containing + but not = is in $\mathcal{L}$ if every + in the word separates two valid words in $\mathcal{L}$.*
(v) *No word is in $\mathcal{L}$ other than those implied by the previous rules.*
Under these rules, the word "$100+1=200$" is in $\mathcal{L}$, but the word "$=+123$" is not. This language expresses natural numbers, well-formed addition statements, and well-formed addition equalities, but it expresses only what they look like (their syntax), not what they mean (semantics). ■

Application. A compiler usually has two distinct components. A lexical analyzer or a scanner, generated by a tool like FLEX, identifies the tokens of the programming language grammar, e.g. identifiers or keywords, which are themselves expressed in a simpler language, usually by means of regular expressions. At the most basic conceptual level, a parser, usually generated by a parser generator like BISON, attempts to decide if the source program is valid, that is if it belongs to the programming language for which the compiler was built. Of course, compilers do more than just parse the source code—they usually translate it into some executable format. Because of this, a parser usually outputs an abstract syntax tree.

2.2 Grammars

A *grammar* is a set of rules for forming words in a language. *Parsing* is the process of recognizing a word by breaking it down to a set of symbols and analyzing each one.

Example 2 *Assume the alphabet is $\Sigma = \{a, b, c\}$, the start symbol is S, and we have the following production rules :*

$$\begin{array}{llll} (1) & S & \rightarrow & aSb \\ (2) & S & \rightarrow & c \end{array}$$

We have to start with S. If we choose rule (1), we obtain the word aSb. If we choose rule (1) again, we replace S with aSb and obtain the word $aaSbb$. If we now choose rule (2), we replace S with c and obtain the word $aacbb$, and are done. This series of choices is

$$S \stackrel{(1)}{\rightarrow} aSb \stackrel{(1)}{\rightarrow} aaSbb \stackrel{(2)}{\rightarrow} aacbb.$$

In the classic formalization of generative grammars first proposed by Noam Chomsky in the 1950s, a grammar G consists of the following components :

- A finite set N of non terminal symbols.
- A finite set Σ of terminal symbols that is disjoint from N.
- A finite set P of production rules, each rule of the form

$$(\Sigma \cup N)^* \, N \, (\Sigma \cup N)^* \rightarrow (\Sigma \cup N)^* \, .$$

- A distinguished symbol $S \in N$ that is the start symbol.

A grammar is formally defined as the tuple (N, Σ, P, S).

Example 3 *Consider the grammar G where $N = \{S, B\}$, $\Sigma = \{a, b, c\}$,*

$$P = \left\{ \begin{array}{llll} (1) & S & \rightarrow & aBSc \\ (2) & S & \rightarrow & abc \\ (3) & Ba & \rightarrow & aB \\ (4) & Bb & \rightarrow & bb \end{array} \right.$$

Checking whether a given word, say $aaabbbccc$, belongs to $L(G)$ is easy if we know the sequence of rules

$$\begin{array}{l} \mathbf{S} \stackrel{(1)}{\rightarrow} aB\mathbf{S}c \stackrel{(1)}{\rightarrow} aBaB\mathbf{S}cc \stackrel{(2)}{\rightarrow} a\mathbf{Ba}Babccc \stackrel{(3)}{\rightarrow} aaB\mathbf{Ba}bccc \stackrel{(3)}{\rightarrow} aa\mathbf{Ba}Bbccc \\ \stackrel{(3)}{\rightarrow} aaaB\mathbf{Bb}ccc \stackrel{(4)}{\rightarrow} aaa\mathbf{Bb}bccc \stackrel{(4)}{\rightarrow} aaabbbccc \end{array}$$

but when this sequence is unknown, the problem is difficult is the general case. Note that the grammar above defines the language $L(G) = \{a^n b^n c^n\}$ where a^n denotes a word of n consecutive a's.

2.2.1 Context-free grammars

A *context-free grammar* is a grammar in which the left-hand side of each production rule consists of only a single non terminal symbol. The corresponding languages are called *context-free languages*. The language defined above $L = \{a^n b^n c^n\}$ is not a context-free language. The language $\{a^n b^n\}$ is. A context-free language can be recognized (or *parsed*) in $O(n^3)$ time by algorithms such as the Earley's algorithm.

Consider the grammar G where $N = \{S, E\}$, $\Sigma = \{a, b, +, (,), =\}$,

$$P = \begin{cases} (1) & S & \rightarrow & E = E \\ (2) & E & \rightarrow & (E) \\ (3) & E & \rightarrow & E + E \\ (4) & E & \rightarrow & a \\ (5) & E & \rightarrow & b \end{cases}$$

Let us check that the word $(a + b) + b = a$ can be build using this grammar

$$\begin{aligned} \mathbf{S} &\overset{(1)}{\rightarrow} \mathbf{E} = E \overset{(3)}{\rightarrow} \mathbf{E} + E = E \overset{(2)}{\rightarrow} (\mathbf{E}) + E = E \overset{(3)}{\rightarrow} (\mathbf{E} + E) + E = E \overset{(4)}{\rightarrow} (a + \mathbf{E}) + E = E \\ &\overset{(5)}{\rightarrow} (a + b) + \mathbf{E} = E \overset{(5)}{\rightarrow} (a + b) + b = \mathbf{E} \overset{(4)}{\rightarrow} (a + b) + b = a \end{aligned}$$

Note that other sequences could have yield the same word. From a sequence of the rules getting the corresponding word is an easy task. The inverse problem known as *parsing* is more difficult. It amounts to finding the sequence of choices that have to be done in order to generate a given word. We shall now give an algorithm to parse a word consistent with a context free grammar.

Shift-reduce parsing. The shift-reduce parsing uses a stack which memorizes some terminal and non terminal symbols. It alternates two steps :

– *Shift step*. Stack symbols of the word one by one from the left to the right.
– *Reduce step*. Every time the symbols on the top of the stack forms a right hand side of a production rule, reduce these symbols to the right hand side of the rule.

Let us illustrate the procedure on our example. We get the following table.

Stack	Word	Operation
$\emptyset$	$(a+b)+b=a$	
$($	$a+b)+b=a$	(shift)
$(a$	$+b)+b=a$	(shift)
$(E$	$+b)+b=a$	(reduce with rule (4))
$(E+$	$b)+b=a$	(shift)
$(E+b$	$)+b=a$	(shift)
$(E+E$	$)+b=a$	(reduce with rule (5))
$(E$	$)+b=a$	(reduce with rule (3))
(E)	$+b=a$	(shift)
E	$+b=a$	(reduce with rule (2))
$E+$	$b=a$	(shift)
$E+b$	$=a$	(shift)
$E+E$	$=a$	(reduce with rule (5))
E	$=a$	(reduce with rule (3))
$E=$	a	(shift)
$E=a$	$\emptyset$	(shift)
$E=E$	$\emptyset$	(reduce with rule (4))
S	$\emptyset$	(reduce with rule (1))

The sequence of rules to be applied to parse the word can be obtained from a backward reading of the shift-reduce table

$$\{1,4,3,5,2,3,5,4\}.$$

which is different from the previous one $\{1,3,2,3,4,5,5,4\}$. The word can thus be generated from the grammar as follows.

$$\begin{aligned}\mathbf{S} &\overset{(1)}{\to} E=\mathbf{E} \overset{(4)}{\to} \mathbf{E}=a \overset{(3)}{\to} E+\mathbf{E}=a \overset{(5)}{\to} \mathbf{E}+b=a \overset{(3)}{\to} E+\mathbf{E}+b=a \\ &\overset{(2)}{\to} E+(\mathbf{E})+b=a \overset{(4)}{\to} \mathbf{E}+(a)+b=a \overset{(2)}{\to} (\mathbf{E})+(a)+b=a \\ &\overset{(3)}{\to} (E+\mathbf{E})+(a)+b=a \overset{(5)}{\to} (\mathbf{E}+b)+(a)+b=a \overset{(4)}{\to} (a+b)+(a)+b=b.\end{aligned}$$

Abstract syntax tree. From a sequence of choices to get a given word, we can associate a tree named *abstract syntax tree*. The leaves of the tree are terminal symbols, other nodes are non terminal symbols, the root is the start symbol. For our previous example, the abstract syntax tree is represented on Figure 2.1.

2.2.2 Regular grammars

A *regular grammar* is a context free grammar for which the right side may be the empty word, a single terminal symbol, or a single terminal symbol followed by a non terminal symbol, but nothing else. The corresponding languages are called *regular languages*.

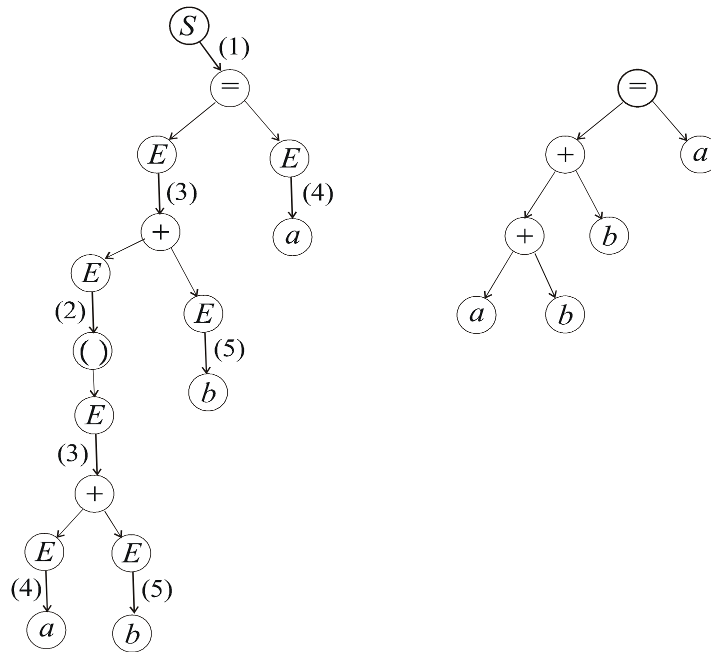

Figure 2.1 – Parse tree associated with the string $(a+b)+b=a$

The language $\{a^n b^m, n \geq 1, m \geq 1\}$ is a regular language, as it can be defined by the grammar with $N = \{S, A, B\}$, $\Sigma = \{a, b\}$ and the following production rules :

$$P = \begin{cases} S & \rightarrow & aA \\ A & \rightarrow & aA \\ A & \rightarrow & bB \\ B & \rightarrow & bB \\ B & \rightarrow & \varepsilon \end{cases}$$

or more concisely

$$P = \begin{cases} S & \rightarrow & aA \\ A & \rightarrow & aA|bB \\ B & \rightarrow & bB|\varepsilon. \end{cases}$$

All languages generated by a regular grammar can be recognized in linear time by a finite state machine. In practice, regular grammars are commonly expressed using regular expressions.

2.3 Regular expression

Regular expressions describe regular languages in compact way. Regular expressions consist of constants and operators that denote sets of words and operations over these sets, respectively. A *literal character* is a singleton of finite alphabet Σ. The empty word is denoted by ε. We define the following operations

$$\begin{array}{rcll} AB & = & \{ab, a \in A, b \in B\} & \text{(concatenation)} \\ A|B & = & A \cup B & \text{(alternation)} \end{array}$$

For example

$$\begin{array}{lcl}\{ab,c\}\{d,ef\} & = & \{abd,abef,cd,cef\}\\ \{ab,c\}|\{ab,d,ef\} & = & \{ab,c,d,ef\}.\end{array}$$

The *Kleene star*, of A denoted by A^* is the smallest superset of A that contains ε and is closed under word concatenation. For example, $\{0,1\}^*$ is the set of all finite binary words. It is assumed that the Kleene star has the highest priority, then concatenation and then set union. If there is no ambiguity then parentheses may be omitted. For instance

$$\begin{array}{lcl}a|b^* & = & \{\varepsilon,a,b,bb,bbb,\dots\}\\ (a|b)^* & = & \{\varepsilon,a,b,aa,ab,ba,bb,aaa,\dots\}\\ ab^*(c|\varepsilon) & = & \{a,ac,ab,abc,abb,abbc,...\}.\end{array}$$

We define also

$$\begin{array}{lcl}a^+ & = & aa^*\\ a? & = & (a|\varepsilon)\,.\end{array}$$

Example 4 *The languages L_k consisting of all words over the alphabet $\{a,b\}$ whose kth-last letter equals a. We have*

$$L_1 = (a|b)^*a(a|b).$$

A possible grammar is provided by the following rules

$$\left\{\begin{array}{llcl}\textit{(1)} & S & \rightarrow & aA|bA|aB\\ \textit{(2)} & A & \rightarrow & aA|bA|aB\\ \textit{(3)} & B & \rightarrow & aC|bC\\ \textit{(4)} & C & \rightarrow & \varepsilon\end{array}\right.$$

There is a simple mapping from regular expressions to the more general *nondeterministic finite automata* (NFAs). For this reason NFAs are often used as alternative representations of regular languages. Figure 2.2 gives an illustration of the NFA corresponding to the grammar of the example.

2.4 Flex and Bison

This section shows how to build a C++ program which builds the abstract syntax tree associated to word (a mathematical expression, a program, ...) using tools such as FLEX/BISON which is often called a *compiler of compilers*. FLEX generates a file named `lex.yy.c` and BISON generates the filer `parser.tab.c`.

2.4.1 Abstract syntax tree

Consider the function $f(\mathbf{x}) = 2x_1.\cos(x_1).\sin(x_2{+}x_1){+}1$. The expression defining f is the word "2*cos(x1)*sin(x2+x
This word belongs to a language which can be defined by a context-free grammar G with the following alphabet $\Sigma = \{x,1,2,+,(,),=,s,i,n,c,o,s\}$. To describe the corresponding grammar, it is more efficient to

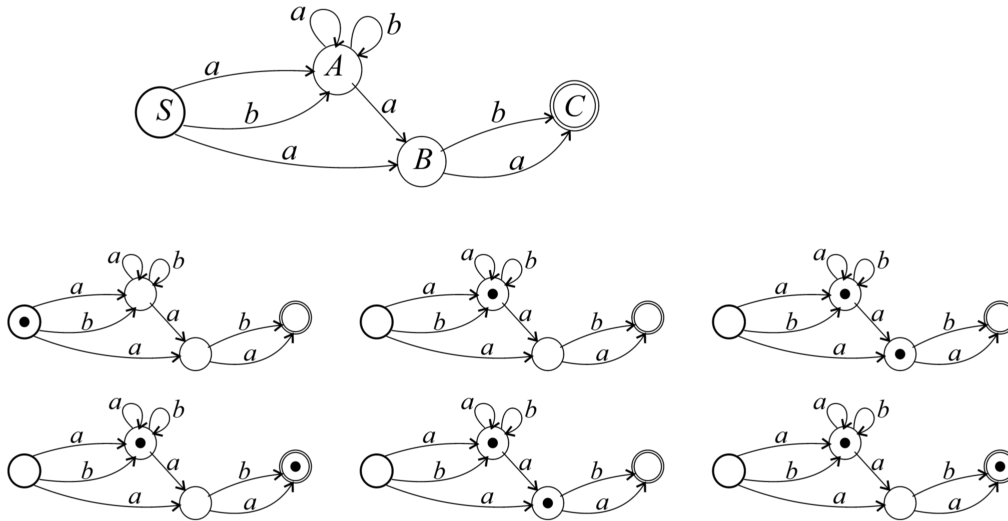

Figure 2.2 – A non deterministic automata recognising the string *babab*

use the compactness of regular expressions whenever possible. As a result, the description of the grammar is decomposed into two parts : the regular part of the language which can be described by regular expressions and the non regular part. For instance, a language to which the expression of our function f belongs can be described by

$$\left\{ \begin{array}{lllll} (1) & REAL & \rightarrow & INTEGER\ (.\{INTEGER\})? \\ (2) & INTEGER & \rightarrow & DIGIT^{+} \\ (3) & DIGIT & \rightarrow & \{0,1,2,3,4,5,6,7,8,9\} \\ (4) & LETTER & \rightarrow & \{a,b,\ldots,Z\} \\ (5) & VARIABLE & \rightarrow & LETTER\ (\ LETTER \mid DIGIT\ )^{*} \\ (6) & F & \rightarrow & \sin \mid \cos \end{array} \right. \tag{2.1}$$

for the regular part and

$$\left\{ \begin{array}{llll} (1) & S & \rightarrow & Expr \\ (2) & Expr & \rightarrow & REAL \mid VARIABLE \mid Expr + Expr \\ & & & \mid Expr * Expr \ \mid F(Expr) \mid (Expr) \end{array} \right. \tag{2.2}$$

for the non regular part. The abstract syntax tree (only the part corresponding to the non regular part) associated to the expression 2*cos(x1)*sin(x2+x1)+1 is given by Figure 2.3.

In the machine, this binary tree can be represented by the table below. The first column corresponds the number of the node (see Figure 2.3) the name column corresponds a real number, a variable's name or a function's name. The left and right columns are the number of the left and right sons (-1 corresponds to

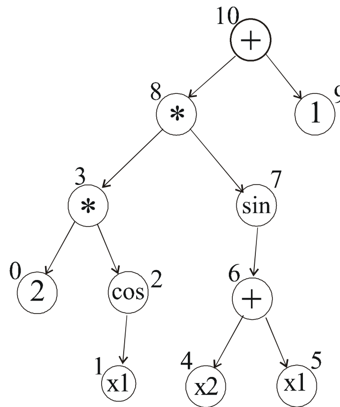

Figure 2.3 –

no son).

n°	name	left	right
0	2	−1	−1
1	x1	−1	−1
2	cos	1	−1
3	∗	0	2
4	x2	−1	−1
5	x1	−1	−1
6	+	4	5
7	sin	6	−1
8	∗	3	7
9	1	−1	−1
10	+	8	9

Equivalently, this table can also be represented by a vector of objects, the class of which contains the number of the corresponding node, the name, the numbers of the two sons. This class is defined in a file that we can call `syntax_tree.h` that will be used by FLEX and BISON.

File "syntax_tree.h"

```
class Node
{
     public :
     QString name ;
     int number,left,right ;
     Node(QString n="",int l=-1,int r=-1, int i=-1) ;
} ;
extern vector <Node> syntax_tree ;
#define YYSTYPE Node
extern YYSTYPE yylval ;
```

BISON generates elements of type `YYSTYPE`. The statement `#define YYSTYPE Node` replaces the `YYSTYPE` (used by BISON) by `Node`. The global variable `syntax_tree` is contains the abstract syntax tree generated by the paser (or compiler) generated by FLEX/BISON.

2.4.2 Flex

A FLEX file encloses the part of the grammar which corresponds to regular expressions. This file is divided into three sections, separated by lines that contain only two percent signs.

- The *definition section* is the place to define macros and to header files written in C. It is also possible to write any C code here, which will be copied verbatim into the generated C source file.
- The *rules section* contains the description of the regular expressions. When the lexer recognises some expressions in the input matching , it executes the associated C code given in the C code section.
- The *C code section* contains C statements and functions that are copied verbatim to the generated source file. These statements presumably contain code called by the rules in the rules section.

The following FLEX file correspond to the regular expressions (2.1). It recognizes strings of integers, reals, variables and functions.

File "scanner.l"

```
%{
     #include "syntax_tree.h" /* Definition section */
%}
     DIGIT          [0-9]
     INTEGER         {DIGIT}+
     REAL             {INTEGER}("."{INTEGER})?
     LETTER         [A-Za-z]
     VARIABLE         {LETTER}({LETTER}|{DIGIT})*
%%
     {REAL} {yylval.name=QString(yytext); return(tk_real);}
     [+*()] { yylval.name=QString(yytext); return *yytext;}
     "sin" {yylval.name="sin"; return(tk_function); }
     "cos" {yylval.name="cos"; return(tk_function);}
     {VARIABLE} {yylval.name=QString(yytext); return(tk_variable);}
%%
```

If this input is given to FLEX, it will be converted into a C file named `lex.yy.c`. The following table provides some explanations related to a FLEX code.

`*`	zero or more copies of the preceding expression
`+`	one or more copies of the preceding expression
`?`	zero or one copy of the preceding expression
`a|b`	`a` or `b`
`\t`	white space
`[abc]`	one of `a,b,c`
`[0-9]`	one digit
`[a-z]`	lowercase letter

The variable `yytext` is a pointer to the matched string. The variable `yylval` is the value associated with the token

2.4.3 Bison

A BISON file encloses the part of the grammar which corresponds the non regular part of the context free grammar. Input to BISON is composed of a *definition section* which consists of token declarations and C code bracketed by "%{" and "%}" and a *rule section* which encloses the *grammar*. The following BISON file is associated to the non regular grammar (2.2).

File. "parser.y"

```
%{
#include "lex.yy.c"
int yyerror(char *s) {printf("Error : %s\n", s); return(0);}
%}
%token tk_function
%token tk_variable
%token tk_real
%left '+' '*'
%%
input : expr;
expr : tk_variable { Node n($1.name,-1,-1,syntax_tree.size());
          $$.number=syntax_tree.size();syntax_tree.push_back(n);}
     |tk_real {Node n($1.name,-1,-1,syntax_tree.size());
          $$.number=syntax_tree.size();syntax_tree.push_back(n);}
     |expr '+' expr {Node n(QString("+"),$1.number,$3.number,syntax_tree.size());
          $$.number=syntax_tree.size();syntax_tree.push_back(n);}
     expr '*' expr {Node n(QString("*"),$1.number,$3.number,syntax_tree.size());
          $$.number=syntax_tree.size();syntax_tree.push_back(n);}
     |tk_function '(' expr ')'{Node n($1.name,$3.number,-1,syntax_tree.size());
          $$.number=syntax_tree.size();syntax_tree.push_back(n);}
     |'(' expr ')';
%%
```

2.4.4 Parsing and using an expression

The following program shows how to use the parser generated by FLEX and BISON. First, we have to create the abstract syntax tree (function `create_syntax_tree`). Then, functions can use the syntax tree, for instance to generate machine code, evaluate the function, generating the expression of its derivative, ... In the program below, the function `Eval` evaluates the expression for input 2D vectors of the form $\mathbf{x} = (x_1, x_2)$.

File : syntax_tree.cpp

```
#include "syntax_tree.h"
#include "parser.tab.c"
Node : :Node(QString n,int l,int r, int i)
     {name=n ; left=l ; right=r ; number=i ;}
int create_syntax_tree(char* f)
{
     syntax_tree.clear() ;
     yy_scan_string(f) ; // result in syntax_tree
}
Interval Eval(int i,vector2D x)
{
     Node n= syntax_tree[i] ;
     if (n.name=="+") return Eval(n.left,x)+Eval(n.right,x) ;
     if (n.name=="*") return Eval(n.left,x)*Eval(n.right,x) ;
     if (n.name=="cos") return cos(Eval(n.left,x)) ;
     if (n.name=="sin") return sin(Eval(n.left,x)) ;
     if (n.name=="x1") return x[1] ;
     if (n.name=="x2") return x[2] ;
     return (n.name.toFloat()) ;
}
Interval Eval(vector2D x) { return Eval(syntax_tree.size()-1,x) ; }
```