

Informatique Fondamentale IMA S8

Cours 5 : Lexical and Syntactic Analysis

Laure Gonnord

<http://laure.gonnord.org/pro/teaching/>
Laure.Gonnord@polytech-lille.fr

Université Lille 1 - Polytech Lille

April 2011



V - Lexical and syntactic analysis : the compiler front-end in practise

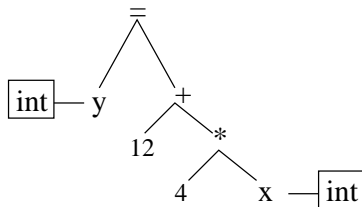
Refreshing memories

Compiler Front-End ► **Abstract Syntax Tree** (AST)

```
int y = 12 + 4*x;
```

⇒ [TKINT, TKVAR("y"), TKEQ, TKINT(12), TKPLUS, TKINT(4), TKFOIS, TKVAR("x"), TKPVIRG]

⇒



- 1 Lexical Analysis aka Lexing
 - Lexing with C
 - Lexing in Java : JFLEX
 - Producing tokens for Parsing !
- 2 Syntactic Analysis aka Parsing
- 3 Syntactic Analysis and rules
- 4 Towards a methodology for designing front-ends in Java
- 5 Other technologies for the front-end

What for ?

```
int y = 12 + 4*x ;
```

⇒ [TKINT, TKVAR("y"), TKEQ, TKINT(12), TKPLUS, TKINT(4), TKFOIS, TKVAR("x"), TKPVIRG]

- ▶ The Lexing produces from a flow of characters a list of tokens.

Algorithm

What's behind ?

From a Regular language, produce an automata (see **course 1**)

A tool for C : LEX - 1

lex : A (standard) tool that produces an automaton that recognises a given language and produces tokens :

- **input** : a set of regular expressions with actions (`toto.lex`).
- **output** : a `.c` that contains the associated automata.

A tool for C : LEX - 2

Demos :

- recognising (simple) arithmetic expressions.
- producing tokens.

.lex format and compilation

.lex construction

```
%{  
  // Initial C code  
%}  
// Macro defs  
%%  
// Rules  
%%  
// Auxiliary c procedures and (eventually) main
```

Compilation with :

```
lex toto.lex //produces lex.yy.c  
gcc -o toto lex.yy.c -ll // links with lex lib
```

.lex example

.lex dummy example

```
%{  
  // nothing there  
%}  
// simple macros  
CHIFFRE      [0-9]  
%%  
{CHIFFRE}+  ;  
[ \t \n]     ;  
<<EOF>>     { printf ("recognized_file !_!\n");  
              exit (0);  
            }  
.  
              { printf ("unrecognized_!\n_"); exit(1)  ;}  
%%  
// nothing
```

- ▶ recognise files with numbers, spaces, tab and newlines

.lex syntax

- *"string"* : a string
- *'c'* : a character
- [A-Z] : a character between A and Z
- <<EOF>> : end of file
- {DIGIT}⁺ : a number (one or more digits)
- [A-Za-z]* : a word (could be empty)
- [-+]?{DIGIT}⁺ a signed number (the sign is optional)
- and more

▶ See the **manual**

<http://dinosaur.compilertools.net/lex/index.html>

.lex variables and functions

Variables :

- `yyin` input file (default is `stdin`)
- `yyout` output file (default is `stdout`)
- `yytext` : last recognized string
- `yylen` longueur de `yytext`

Functions :

- `yylex()` call to `lex`, active until the first return
 - `yywrap()` useful to deal with several files.
- ▶ example.

Lex counts !

Lex is a little more expressive than regular automata :

.lex dummy example

```
%{
    int num_lines = 0;
    int num_chars = 0;
}%
//no macros

%%
\n          { num_lines++; num_chars++;}
.          { num_chars++ ;}
%%

int main() {
    yylex();
    printf("#_of_lines_=%d, #_of_chars_=%d\n",
           num_lines, num_chars );
    return 0;
}
```

Lexing tool for java : JFLEX

The official webpage : `jflex.de` (GPL)

Lexing in java -1

A minimal example from the distribution (standalone.flex) :

```
%%

%public          <<<---- generates a public class
%class Subst    <<<---- which name is Subst.java
%standalone     <<<---- standalone use (no cup)

%unicode        <<<---- encoding

%{
  String name;   <<<-- declaration of local var
%}

%%

"name_" [a-zA-Z]+ { name = yytext().substring(5); }
[Hh] "ello"      { System.out.print(yytext()+"_"+name+"!"); }
```

► Demo !

Lexing in java - 2

Commands :

```
flex standalone.flex //produces Subst.java file
javac Subst.java //produces Subst.class
java Subst totoexample // the lexer in action !
```

So Far ...

Lex/JFlex have been used to produce **acceptors** for (\simeq regular) languages.

\implies we have to produce tokens (terminal symbols)

Terminal symbols

For instance :

- numbers
- identifiers
- operations
- keywords
- braces, brackets

With C/Lex

Syntax :

```
{CHIFFRE}+           { return TK_INT ; }
```

The tokens must be declared (in yacc file)

With Java/FLEX

Syntax (using Symbol from `java_cup.runtime.Symbol`):

```
"+"          { return symbol(sym.PLUS); }
```

The token `sym.PLUS` must be declared (in cup file)

Terminal symbols - tokens and values

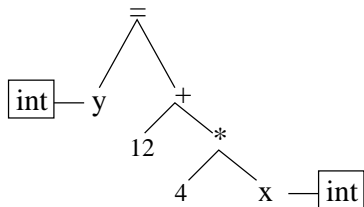
The token may have values :

- TK_INT (+value int)
 - TK_ID (+ value string)
 - ...
- ▶ See later for examples in C and Java.

- 1 Lexical Analysis aka Lexing
- 2 Syntactic Analysis aka Parsing
 - Parsing ?
 - Parsing in C : Yacc (or Bison)
 - Parsing in Java : Cup
- 3 Syntactic Analysis and rules
- 4 Towards a methodology for designing front-ends in Java
- 5 Other technologies for the front-end

What's Parsing ?

[TKINT, TKVAR("y"), TKEQ, TKINT(12), TKPLUS, TKINT(4),
TKFOIS, TKVAR("x"), TKPVIRG]



or “yes, it belongs to the grammar !”

From the grammar to the parser

The grammar must be a **context-free grammar**

$S \rightarrow aSb$

$S \rightarrow \epsilon$

In this grammar :

- S is the start symbol
- a and b are **non terminal** tokens (**produced by the lexing phase**)

Recognising $a^n b^n$ - with lex/yacc

example4.lex

```
%{  
  // inclusion of generated  
  // .h from yacc  
  #include "y.tab.h"  
  #include <stdio.h>  
  %}  
  // macros  
  %%  
  "a" {return TK_A;}  
  "b" {return TK_B;}  
  %%  
  // nothing
```

example4.y

```
%{ // initial code  
#include <stdio.h>  
%}  
  
%token TK_A TK_B  
%start S  
%%  
  // grammar rules  
  S : TK_A S TK_B  
    |  
    ;  
  %%  
  int main(void) {  
    yyparse();  
    printf("end_of_parsing\n");  
  }
```

► **Syntax error** on "aaaaab".

A Makefile for lex/yacc

```
NAME=exemple4

all: y.tab.c lex.yy.c
    gcc -o $(NAME) y.tab.c lex.yy.c -ll -ly

lex.yy.c : $(NAME).lex
    lex $(NAME).lex

y.tab.c : $(NAME).y
    yacc -d $(NAME).y

clean:
    rm lex.yy.c y.tab.c y.tab.h *~
```

Recognising $a^n b^n$ - with Flex and Cup -1

anbn.flex

```
import java_cup.runtime.*; // import Symbol class etc
%%
%class Anbn
%unicode
%line
%column
%cup
%{ /* a function to create tokens along with line , col. numbers */
private Symbol symbol(int type) {
    return new Symbol(type, yyline, yycolumn);
}
%}
/* macros */
%%
/* rules */
'a' {return symbol(sym.TKA) ; }
'b' {return symbol(sym.TKB) ; }
```

► flex generates **Anbn.java**

Recognising $a^n b^n$ - with Flex and Cup -2

anbn.cup

```
import java_cup.runtime.*;
parser code {:
    public void report_fatal_error( String message, Object info )
    throws Exception {report_error (message, info );
    throw new Exception("Syntax_Error");
    }
:};

terminal TKA,TKB;
non terminal S;

S ::= TKA S TKB | ;
```

► **cup** generates two classes : **parser.java** and **sym.java**

Recognising $a^n b^n$ - with Flex and Cup -3

The main class :

Analyseur.java

```
import java.io.*;

public class Analyseur {
    static public void main ( String argv[] ){

try {
    parser p = new parser(new Anbn(new FileReader(argv[0]))) ;
    Object result = p.parse();
    System.out.println("\n_file_ OK" ) ;
} catch ( Exception e ) {
    System.out.println("\n_Syntax_Error" ) ;
}
}
}
```

► Compiled with all **.java**. ► **warning**, to run, java-cup-11a.jar must be in the classpath (or used with the -cp option)

A Makefile for flex/cup

```
JFLEX=/home/laure/analyseurs/jflex-1.4.3/bin/jflex
CUPJAR=/home/laure/analyseurs/jflex-1.4.3/java-cup-11a.jar

# directory/package containing your sources
CUPFILE=anbn.cup # the cup file containing your grammar
LEXFILE=anbn.flex # the flex file containing your lexical analyzer

CPATH=.:$(CUPJAR)
SOURCES=*.java

all: cup flex
javac -cp $(CPATH) $(SOURCES)

cup: $(CUPFILE)
java -jar $(CUPJAR) $(CUPFILE)

flex: $(LEXFILE)
$(JFLEX) $(LEXFILE)

clean:
rm *.class parser.java sym.java *~ Anbn.java
```

- 1 Lexical Analysis aka Lexing
- 2 Syntactic Analysis aka Parsing
- 3 Syntactic Analysis and rules**
- 4 Towards a methodology for designing front-ends in Java
- 5 Other technologies for the front-end

So Far ...

Lex/Yacc and JFlex/Cup have been used to produce **acceptors** for context-free languages

⇒ the abstract syntax tree remains to be constructed (then used !)

Semantic actions

Semantic actions : code that are performed each time a grammar rule is matched.

Example in C/Yacc

```
S : TK_A S TK_B { printf("rule_1\n"); }
```

Example in Java/Cup

```
S ::= TKA S TKB { : System.out.println("rule1"); : }
```

► We can do more than pretty print !

Semantic actions and implicit AST in C - 1

Example : evaluation of an arithmetic expression in C (12+5*6).

example5.lex

```

#include "y.tab.h"
#include <stdio.h>
%}
DIGIT      [0-9]
%%
{DIGIT}+  { yylval=atoi(yytext);
           return TK_INT; }
"+"       { return TK_PLUS; }
"*"       { return TK_TIMES; }
";"       { return TK_SEMICOL; }
[ \t \n] ;
%%

```

Semantic actions and implicit AST in C - 2

Example : example5.y

```

%{ // code initial
#include <stdio.h>
%}
%token TK_INT TK_PLUS TK_TIMES TK_SEMICOL
%left TK_PLUS
%left TK_TIMES
%start S
%%
//rules
S: E TK_SEMICOL { printf("result_:_%d\n", $1); }
;
E: TK_INT { $$=$1; }
  | E TK_PLUS E { $$=$1+$3; }
  | E TK_TIMES E { $$=$1*$3; }
;
%%
int main(void) {
  yyparse();
  printf("end_of_parsing\n");
}

```

Semantic actions and implicit AST in Java

a part of `expr.flex`

```
{ integer } { return symbol(sym.TKINT,new Integer(yytext()));;}
"+"         {return symbol(sym.TKPLUS) ; }
```

a part of `expr.cup`

```
E ::= TKINT:n
    { : RESULT = new Integer(n.intValue()); :}
|   E:e1 TKPLUS E:e2
    { : RESULT = new Integer(e1.intValue() + e2.intValue()) ; :}
```

Do not forget to declare that `* > +`

Explicit AST, why ?

Why not program our compilers entirely using semantic actions ?

- Because manipulating a tree is easier.
- Because the semantics actions are not really easy to read
- Because of **the separation of concerns**

http:

[//en.wikipedia.org/wiki/Separation_of_concerns](http://en.wikipedia.org/wiki/Separation_of_concerns)

▶ Parse, **then** evaluate/print/construct another internal representation, ...

Semantic actions and explicit AST in Java - 1

The class ASTExpr.java : The Tree !

```

public class ASTExpr {
    final static int INT=0, ADD=1, MUL=2 ;
    int tag ;
    int asInt ; // value used if tag = INT
    ASTExpr e1, e2 ; // used if ADD or MUL
    //Constructors
    ASTExpr(int i) { tag = INT ; asInt = i ; }
    ASTExpr(ASTExpr e1, int op, ASTExpr e2) {
        tag = op; this.e1 = e1; this.e2 = e2; }
    //evaluation of an expression
    int eval() {
        switch (this.tag) {
            case ASTExpr.INT: return this.asInt;
            case ASTExpr.ADD: return this.e1.eval()+this.e2.eval();
            case ASTExpr.MUL: return this.e1.eval()*this.e2.eval();
        }
        throw new Error("incorrect_tag");
    }
}

```

Semantic actions and explicit AST in Java - 2

a part of expr.cup

```
non terminal ASTExpr E;
```

```
S ::= E:e TKSEMICOL { :System.out.println(e.eval()); :}  
;
```

```
E ::= TKINT:n  
    { : RESULT = new ASTExpr(n.intValue()); :}  
  | E:e1 TKPLUS E:e2  
    { : RESULT = new ASTExpr(e1,ASTExpr.ADD,e2); :}  
;
```

- 1 Lexical Analysis aka Lexing
- 2 Syntactic Analysis aka Parsing
- 3 Syntactic Analysis and rules
- 4 Towards a methodology for designing front-ends in Java
- 5 Other technologies for the front-end

The running example

```
vars x,y,z;  
y:=13;  
z:=80;  
x:=y+z;  
z:=x*12;  
print(x);
```

- ▶ Parse and evaluate expressions in Java !

Questions

- What is the grammar ? (and keywords, and end symbols ...)
- Write the lex and cup files.
- Construct the intermediate representation in Java **But**
How ?

A class hierarchy as intermediate representation

A quick look at **the grammar** :

```
program ::= instruction_l  
instruction_l ::= instruction instruction_l  
instruction ::= declaration | assignment | print
```

Then

- Each non-terminal is a class.
- Instruction will be an abstract class
- class Declaration extends Instruction
- the class Program will have an interpret() function.

Last (?) problem

We have to store the variables and their (current) values, **the context**

- ▶ Use a hashmap !

```
HashMap<String,Integer> currentContext
```

- ▶ Evaluating an expression requires this context !

- 1 Lexical Analysis aka Lexing
- 2 Syntactic Analysis aka Parsing
- 3 Syntactic Analysis and rules
- 4 Towards a methodology for designing front-ends in Java
- 5 Other technologies for the front-end

Front-end more recent technologies

- XML parsers (java, ...) : more for data languages
- ANTLR (multi languages)
- ROSE (C/C++ frontend) : source to source translator, provides high level functions in C++
- LLVM, (C/C++) more for code optimisation, still in research domain.