Informatique Fondamentale IMA S8

Cours 5 : Lexical and Syntactic Analysis and Compiler front-end construction

Laure Gonnord

`http://laure.gonnord.org/pro/teaching/`

`Laure.Gonnord@polytech-lille.fr`

Université Lille 1 - Polytech Lille

2012

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

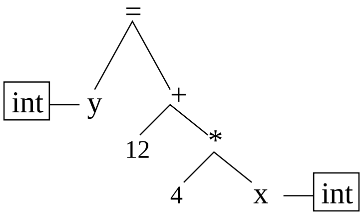
Refreshing memories

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

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

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

$\Longrightarrow$



What for ?

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

$\Longrightarrow$ [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**)

Tools : lexical analysers constructors

Lex(C) JFlex(Java), OCamllex, ... are tools that produces an automaton that recognises a given language and produces tokens :
(here we focus on JFlex)

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

Lexing tool for java : JFLEX

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

A first example : `numbers.flex`.

.lex format and compilation

.flex construction

```
%%
%public                 <<<--- generates a public class
%class xx               <<<--- which name is xx.java
%standalone             <<<--- standalone use (no cup)
%unicode                <<<--- encoding

 %{
                        <<<-- declaration of local vars
 %}
%%
                        <<<-- flex rules
```

Compilation with :

```
jflex toto.flex          // produces xx.java
javac xx.java            // compiles into xx.class
java xx filetotest       // tests if file2text is recog
```

Lexing in java - ex2

An example from the flex 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+"!"); }
```

.flex variables and functions

Access to **lexing info** :

- `yytext()` : last recognized string
- `yylength()` : `yytext`'s length

(and other, see flex documentation on `/usr/share/doc/jflex/`)

Functions :

- `yylex()` lexing standard function (standalone)
- lexing function with cup : `next_token()`

Lexing in java - ex 3 - more than regular languages

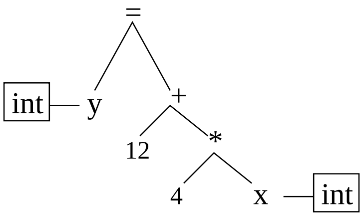
Counting in JFLEX - counts.flex

```
[..]
%{/*variable declaration*/
   int num_lines;
%}

%init{/*code to be embedded in the class constructor*/
   num_lines=0;
%init}

%eof{ /*When EOF occurs, this code is executed*/
  System.out.println(num_lines+" lignes");
%eof}

NEWLINE=\r|\n|\r\n
SPACE=\ |\t
%%
{NEWLINE}                 {num_lines++; }
{SPACE}                   {}
```

What's Parsing ?

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

$\Longrightarrow$



or "yes, it belongs to the grammar !"

From the grammar to the parser

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

```
S-> aSb
S-> eps
```

In this grammar :

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

▶ This grammar recognizes $a^n b^n$.

So Far ...

JFlex has been used to produce acceptors for ($\simeq$ regular) languages.
We want more (general) grammars acceptors !
▶ From a context-free grammar, construct an automaton (see **course 2**).

Thus we need a way :

- to declare terminal symbols (**tokens**)
- to produce them from the input file. (JFlex)
- to write grammars.

▶ In `.flex` and `.cup` files.

Terminal symbols/tokens

For instance :

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

Returning tokens With Java/JFLex

Tokens as symbol instances

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

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

The token may have **values** :

- an integer value for numbers
- a string value for identifiants

Tokens with values

```
[0-9]+          { return symbol(sym.TKINT,new Integer(yytext()));}
```

Tokens must be declared (in cup file), see later.

.cup format and compilation

.cup construction

```
import java_cup.runtime.*; <<<--- imports

init with {: ... :};      <<<--- optional user code
                      this one to be executing before parsing

terminal .... :           <<<--- token declaration
non terminal ... ;

precedence  ...;          <<<--- precedence and associativity (opt)

S::=     ... ;            <<<-- grammar rules and (opt) actions
```

The compilation of the cup file produces parser.java and sym.java

```
java -jar java-cup-11a.jar toto.cup
```

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
%{
/* create tokens along with line,col. numbers */
private Symbol symbol(int type) {
    return new Symbol(type, yyline, yycolumn);
 }
%}
%%
 /* 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 {: /* to handle syntax errors */
 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␣file␣not␣found␣?" ) ;
          }
    }
}
```

- 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
```

So Far ...

JFLex/Cup have been used to produce acceptors for context-free languages

$\Longrightarrow$ 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.

Printing as a semantic action in JavaCup

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

- We can do more than pretty print !

Semantic actions for expressions

a part of expr.flex

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

a part of expr.cup

```
non terminal Integer E;

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

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

- we can attach **attributes** to (non) terminals.

Semantic actions : priorities

We can tell JavaCup how to solve conflicts while parsing :

- **precedences** to solve `3 + 4 * 5`
- **associativity** to solve the pb of `3 + 4 + 5`
- **nonassoc** : `2==0` but not `3==6==10`

a part of expr.cup

```
precedence left TKPLUS;
precedence left TKTIMES; // * has more precedence than +
[...]
S ::= ...
```

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`

▶ 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); :}
;
```

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 !

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.