# SU/FSI/master/info/MU4IN503
# Analyse lexicale et syntaxique
# lex et yacc

P. Manoury

janvier 2020

## 1 En C

### 1.1 Syntaxe abstraite

Fichier : `ast.h`

```
typedef struct _expr *Expr;
typedef struct _exprs *Exprs;
typedef enum _tag Tag;
typedef enum _oprim Oprim;

enum _tag {
  ASTNum, ASTId, ASTPrim
};

enum _oprim { AST_ADD, AST_SUB, AST_DIV, AST_MUL };

struct _expr {
  Tag tag;
  union {
    int num;
    char* id;
    struct {
      Oprim op;
      Exprs opands;
    } prim;
  } content;
};

struct _exprs {
  Expr head;
  Exprs tail;
};

Expr newASTNum(int n);
Expr newASTId(char* x);
Expr newASTPrim(Oprim op, Exprs es);
```

```
Exprs addExpr(Expr e, Exprs es);

#define mallocExpr malloc(sizeof(struct _expr))
#define mallocExprs malloc(sizeof(struct _exprs))
#define tagOf(r) r->tag
#define getNum(r) r->content.num
#define getId(r) r->content.id
#define getOp(r) r->content.prim.op
#define getOpands(r) r->content.prim.opands
```

Fichier : `ast.c`

```
#include <stdlib.h>
#include <stdio.h>
#include "ast.h"

Expr newASTNum(int v) {
  Expr r = mallocExpr;
  r->tag = ASTNum;
  r->content.num = v;
  return r;
}

Expr newASTId(char* v) {
  Expr r = mallocExpr;
  r->tag = ASTId;
  r->content.id = v;
  return r;
}

Expr newASTPrim(Oprim op, Exprs es) {
  Expr r = mallocExpr;
  r->tag = ASTPrim;
  r->content.prim.op = op;
  r->content.prim.opands = es;
  return r;
}

Exprs addExpr(Expr e, Exprs es) {
  Exprs r = mallocExprs;
  r->head = e;
  r->tail = es;
  return r;
}
```

## 1.2 Analayse lexicale

Fichier : `lexer.lex`

```
%{
```

```
#include <stdlib.h>
#include "ast.h"
#include "y.tab.h"
%}

nls "\n"|"\r"|"\r\n"
nums "-"?[0-9]+
idents [a-zA-Z][a-zA-Z0-9]*
%%

[ \t]  { /* On ignore */ }
{nls}  {  }



"add"   return(PLUS);
"sub"   return(MINUS);

"mul"   return(TIMES);
"div"   return(DIV);

"("   return(LPAR);
")"   return(RPAR);

{nums}    {
  yylval.num=atoi(yytext);
      return(NUM);
    }

{idents}    {
      yylval.str=strdup(yytext);
      return(IDENT);
    }
```

### 1.3 Analyse grammaticale

Fichier : `parser.y`

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "ast.h"
#include "prologTerm.h"

int yylex (void);
int yyerror (char *);

 Expr theExpr;

%}
```

```
%token<num>  NUM
%token<str>  IDENT
%token  PLUS MINUS TIMES  DIV
%token  LPAR RPAR
%token  NL

%union {
  int num;
  char* str;
  Expr expr;
  Exprs exprs;
}

%type<expr> expr
%type<exprs> exprs
%type<expr> line

%start line
%%

line: expr    { theExpr = $1; }
  ;

expr:
  NUM                       { $$ = newASTNum($1); }
| IDENT                     { $$ = newASTId($1); }
| LPAR PLUS exprs RPAR   { $$ = newASTPrim(AST_ADD,$3); }
| LPAR MINUS exprs RPAR   { $$ = newASTPrim(AST_SUB,$3); }
| LPAR TIMES exprs RPAR   { $$ = newASTPrim(AST_MUL,$3); }
| LPAR DIV exprs RPAR     { $$ = newASTPrim(AST_DIV,$3); }
;
exprs:
  expr { $$ = addExpr($1,NULL); }
| expr exprs { $$ = addExpr($1,$2); }
%%

int yyerror(char *s) {
  printf("error: %s\n",s);
  return 1;
}

int main(int argc, char **argv) {
  yyparse();
  printExpr(theExpr);
  printf("\n");
  return 0;
}
```

### 1.4 Termes Prolog

Fichier : `prologTerm.h`

```
void printExpr(Expr);
void printExprs(Exprs);
```

Fichier : `prologTerm.c`

```
#include <stdio.h>
#include "ast.h"
#include "prologTerm.h"

void printOp(Oprim op) {
  switch(op) {
  case AST_ADD : printf("add"); break;
  case AST_SUB : printf("sub"); break;
  case AST_MUL : printf("mul"); break;
  case AST_DIV : printf("div"); break;
  }
}

void printNum(int n) {
  printf("num(%d)",n);
}

void printId(char* x) {
  printf("var(%s)",x);
}

void printExpr(Expr e) {
  switch(tagOf(e)) {
  case ASTNum : printNum(getNum(e)); break;
  case ASTId : printId(getId(e)); break;
  case ASTPrim : {
    printOp(getOp(e));
    printf("([");
    printExprs(getOpands(e));
    printf("])");
    break;
  }
  }
}

void printExprs(Exprs es) {
  if (es) {
    while (es->tail) {
      printExpr(es->head);
      printf(",");
      es = es->tail;
    };
    printExpr(es->head);
  }
}

/*
void printExprs(Exprs es) {
```

```
  if (es) {
    if (es->tail) {
      printExpr(es->head);
      printf(",");
      printExprs(es->tail);
    }
    else {
      printExpr(es->head);
    }
  }
}
*/
```

### 1.5 Makefile

```
LEX_C = flex
YACC_C = yacc
GCC = gcc

prologTerm: parser ast.h ast.c prologTerm.c
$(GCC) -c ast.c
$(GCC) -c prologTerm.c
$(GCC) -o prologTerm ast.o prologTerm.o lex.yy.o y.tab.o -lm -ll

parser: lexer.lex parser.y
$(YACC_C) -d parser.y
$(LEX_C) lexer.lex
$(GCC) -c lex.yy.c
$(GCC) -c y.tab.c

clean:
rm lex.yy.*
rm y.tab.*
rm *.o
rm prologTerm
```

## 2 En JAVA

### 2.1 Synatxe abstraite et termes prolog

Fichier : `Op.java`

```
public enum Op {
    ADD("add"), SUB("sub"), MUL("mul"), DIV("div");

    private String str;

    Op(String str) { this.str = str; }

    public String toString() {
return this.str;
    }
```

```
}
```

Fichier : `Ast.java` (interface)

```
interface Ast {

public String toPrologString();

}
```

Fichier : `AstNum.java`

```
public class AstNum implements Ast {

Integer val;

AstNum(Integer n) {
val = n;
}

@Override
public String toPrologString() {
return ("num("+val+")");
}

}
```

Fichier : `AstId.java`

```
public class AstId implements Ast {

String name;

AstId(String x) {
name = x;
}

@Override
public String toPrologString() {
return "var("+name+")";
}

}
```

Fichier : `AstPrim.java`

```
import java.util.ArrayList;

public class AstPrim implements Ast {

    Op op;
    ArrayList<Ast> opands;

    AstPrim(Op op, ArrayList<Ast> es) {
this.op = op;
```

```
this.opands = es;
    }

    public String toPrologString() {
String r = "";
r = op.toString()+"([";
for(int i=0; i < opands.size()-1; i++)
    r += opands.get(i).toPrologString()+",";
r += opands.get(opands.size()-1).toPrologString();
r += "])";
return r;
    }

}
```

## 2.2 Analyse lexicale

Fichier : `lexer.lex`

```
%%

%byaccj

%{
  private Parser yyparser;

  public Yylex(java.io.Reader r, Parser yyparser) {
    this(r);
    this.yyparser = yyparser;
  }
%}

nums = -?[0-9]+
ident = [a-z][a-zA-Z0-9]*
nls  = \n | \r | \r\n

%%

/* operators */
"add"  { return Parser.PLUS; }
"sub"  { return Parser.MINUS; }
"mul"  { return Parser.TIMES; }
"div"  { return Parser.DIV; }

/* parenthesis */
"("  { return Parser.LPAR; }
")"  { return Parser.RPAR; }

/* newline */
{nls}   { return 0; } //{ return Parser.NL; }

/* float */
```

```
{nums}  { yyparser.yylval = new ParserVal(Integer.parseInt(yytext()));
          return Parser.NUM; }

{ident} { yyparser.yylval = new ParserVal(yytext());
  return Parser.IDENT;
}

/* whitespace */
[ \t]+ { }

\b     { System.err.println("Sorry, backspace doesn't work"); }

/* error fallback */
[^]    { System.err.println("Error: unexpected character '"+yytext()+"'"); return -1; }
```

### 2.3 Analyse grammaticale (BYACC/J)

Fichier : `parser.y`

```
%{
  import java.io.*;
  import java.util.ArrayList;
%}

%token NL                    /* newline  */
%token <ival> NUM            /* a number */
%token <sval> IDENT          /* an identifier */
%token PLUS MINUS TIMES DIV  /* operators */
%token LPAR RPAR             /* parethesis */

%type <obj> line
%type <obj> expr
%type <obj> exprs

%start line
%%


line:  expr   { prog=(Ast)$1; $$=$1; }
;

expr:
  NUM                      { $$ = new AstNum($1); }
| IDENT                    { $$ = new AstId($1); }
| LPAR PLUS exprs RPAR   { $$ = new AstPrim(Op.ADD,(ArrayList<Ast>)$3); }
| LPAR MINUS exprs RPAR  { $$ = new AstPrim(Op.SUB,(ArrayList<Ast>)$3); }
| LPAR TIMES exprs RPAR  { $$ = new AstPrim(Op.MUL,(ArrayList<Ast>)$3); }
| LPAR DIV exprs RPAR    { $$ = new AstPrim(Op.DIV,(ArrayList<Ast>)$3); }
;
exprs:
expr                   { ArrayList<Ast> r = new ArrayList<Ast>();
                         r.add((Ast)$1);
```

```
 $$ = r; }
| expr exprs           { ((ArrayList<Ast>)$2).add((Ast)$1); $$ = $2; }
;
%%

  public Ast prog;

  private Yylex lexer;


  private int yylex () {
    int yyl_return = -1;
    try {
      yylval = new ParserVal(0);
      yyl_return = lexer.yylex();
    }
    catch (IOException e) {
      System.err.println("IO error :"+e);
    }
    return yyl_return;
  }


  public void yyerror (String error) {
    System.err.println ("Error: " + error);
  }


  public Parser(Reader r) {
    lexer = new Yylex(r, this);
  }
```

### 2.4 Termes prolog

Fichier : `PrologTerm.java`

```
import java.io.*;

class PrologTerm {

    public static void main(String args[]) throws IOException {

Parser yyparser;
Ast prog;

yyparser = new Parser(new InputStreamReader(new FileInputStream(args[0])));
yyparser.yyparse();
prog = (Ast) yyparser.yyval.obj;

if (prog != null)
System.out.println(prog.toPrologString());
else
```

```
System.out.println("Null");
    }

}
```

### 2.5 Makefile

```
LEX_J  = jflex
YACC_J = ~/tmp/yacc.macosx -J
JAVAC = javac

prologTerm: parser Op.java PrologTerm.java
$(JAVAC) PrologTerm.java

parser: parser.y lexer.lex
$(LEX_J) lexer.lex
$(YACC_J) parser.y

clean:
rm Parser*.java
rm Yylex.java
rm *.class
```

## 3 En OCAML

### 3.1 Syntaxe abstraite

Fichier : `ast.ml`

```
type op = Add | Mul | Sub | Div

let string_of_op op =
  match op with
      Add -> "add"
    | Mul -> "mul"
    | Sub -> "sub"
    | Div -> "div"

let op_of_string op =
  match op with
      "add" -> Add
    | "mul" -> Mul
    | "sub" -> Sub
    | "div" -> Div

type expr =
    ASTNum of int
  | ASTId of string
  | ASTPrim of op * expr list
```

### 3.2 Analyse lexicale

Fichier : `lexer.mll`

```
{
  open Parser        (* The type token is defined in parser.mli *)
  exception Eof

}
rule token = parse
    [' ' '\t' '\n']       { token lexbuf }     (* skip blanks *)
  | ['0'-'9']+('.'['0'-'9'])? as lxm { NUM(int_of_string lxm) }
  | "add"             { PLUS }
  | "sub"             { MINUS }
  | "mul"             { TIMES }
  | "div"             { DIV }
  | '('               { LPAR }
  | ')'               { RPAR }
  | ['a'-'z']['a'-'z''A'-'Z''0'-'9']* as lxm { IDENT(lxm) }
  | eof               { raise Eof }
```

### 3.3 Analyse grammaticale

Fichier : `parser.mly`

```
%{
open Ast
%}

%token <int> NUM
%token <string> IDENT
%token PLUS MINUS TIMES DIV
%token LPAR RPAR

%type <Ast.expr> expr
%type <Ast.expr list> exprs

%start expr             /* the entry point */

%%

  expr:
    NUM                       { ASTNum($1) }
  | IDENT                     { ASTId($1) }
  | LPAR PLUS exprs RPAR   { ASTPrim(Ast.Add, $3) }
  | LPAR MINUS exprs RPAR  { ASTPrim(Ast.Sub, $3) }
  | LPAR TIMES exprs RPAR  { ASTPrim(Ast.Mul, $3) }
  | LPAR DIV exprs RPAR    { ASTPrim(Ast.Div, $3) }
  ;
  exprs :
    expr       { [$1] }
  | expr exprs { $1::$2 }
  ;
```

### 3.4 Termes Prolog

Fichier : `prologTerm.ml`

```
open Ast

let rec print_expr e =
  match e with
      ASTNum n -> Printf.printf"num(%d)" n
    | ASTId x -> Printf.printf"var(%s)" x
    | ASTPrim(op, es) -> (
Printf.printf"%s" (string_of_op op);
Printf.printf"([";
print_exprs es;
Printf.printf"])"
      )
and print_exprs es =
  match es with
      [] -> ()
    | [e] -> print_expr e
    | e::es -> (
print_expr e;
print_char ',';
print_exprs es
      )
;;

let fname = Sys.argv.(1) in
let ic = open_in fname in
  try
    let lexbuf = Lexing.from_channel ic in
    let e = Parser.expr Lexer.token lexbuf in
      print_expr e;
      print_char '\n'
  with Lexer.Eof ->
    exit 0
```

### 3.5 Makefile

```
LEX_ML = ocamllex
YACC_ML = /usr/local/bin/ocamlyacc
OCAMLC = ocamlc

prologTerm: parser prologTerm.ml
$(OCAMLC) -o prologTerm ast.cmo lexer.cmo parser.cmo prologTerm.ml

parser: ast.ml lexer.mll parser.mly
$(OCAMLC) -c ast.ml
$(LEX_ML) -o lexer.ml lexer.mll
```

```
$(YACC_ML) -b parser parser.mly
$(OCAMLC) -c parser.mli
$(OCAMLC) -c lexer.ml
$(OCAMLC) -c parser.ml

clean:
rm -f *.cmo
rm -f *.cmi
rm -f prologTerm
rm -f lexer.ml
rm -f parser.mli
rm -f parser.ml
rm *~
```