Computing while Parsing

with **Parsers which Compute**

Michaël PÉRIN, Polytech Grenoble

December 2021

Context-free grammar

Our running example : the simplest context-free language

$$L = \{a^n b^n | n \in Nat\}$$

A context-free grammar which generates/recognizes L

```
Seed S
S → a . S . b
  | ϵ
```

From grammar to parser (1)

Explicit reading until EOF (End Of File)

- introduce a new non-terminal P
- which becomes the seed

Grammar with explicit EOF

```
Seed P
P → S . \EOF
S → a . S . b
  | ϵ
```

From grammar to parser (2)

1. Non-terminals become boolean functions
   - true means **accepted** / recognized
   - false means **rejected**

Parser

```
start P ;

bool P() ::= S() ; "\EOF"

bool S() ::= "a" ; S() ; "b"
           | ϵ
```

From grammar to parser (3)

1. Non-terminals become boolean functions
2. Computing the boolean result

Complete the ... with computations

```
start P ;

bool P() ::= acc = S() ; "\EOF" { return ...; }

bool S() ::=
      | "a" ; acc = S() ; "b"  { return ...; }
      |  ϵ  { return ...; }
      | "b" { return ...; } // useless, just for help
```

From grammar to parser (4)

SOLUTION

```
start P ;

bool P() ::=  acc = S() ; "\EOF"  { return acc; }

bool S() ::=
      | "a" ; acc = S() ; "b" { return acc; }
      |  ϵ  { return true; }
      | "b" { return false; } // useless
```

From grammar to parser (4)

SOLUTION

```
start P ;

bool P() ::=  acc = S() ; "\EOF"  { return acc; }

bool S() ::=
      | "a" ; acc = S() ; "b" { return acc; }
      |  ϵ  { return true; }
      | "b" { return false; } // useless
```

(Almost) Standard Notation for Parsers

```
bool S() : { bool acc; } // declarations of var.
{     | "a" ; acc = S() ; "b" { return acc; }
      |  ϵ  { return true; }
      | "b" { return false; } // useless
}
```

The code produced by the Parser Generator


```
int main(){ return P(); }

bool P(){
  bool acc = S() ;
  if (read()=="\EOF"){ remove("\EOF"); return acc; }
  return false;
}

bool S(){
  switch ( read() ){
    case "a": remove("a") ;
      bool acc = S() ;
      if ( read()=="b" ){ remove("b"); return acc; }
      break;
    case "" : retrun true;
    default: return false;
  }
}
```

(opt) Dealing with syntax errors

Actually, the generated parser never returns false :

- it either returns true
- or fails with an exception.

How to write a user-friendly parser

- that never fails with an exception. Instead it parses all the file and returns a boolean
- and repairs/shows syntax errors

(opt) A User-Friendly Parser that never fails (1/2)

Each parsed symbol is reprinted

- unexpected symbols appear in red
- missing symbols appear in green

Here are the expected behaviours

input $\rightsquigarrow$ *accepted, repair*

- ab $\rightsquigarrow$ true, ab
- b $\rightsquigarrow$ false, b
- bbab $\rightsquigarrow$ false, bbab
- aab $\rightsquigarrow$ false, aabb
- aabab $\rightsquigarrow$ false, aabab
- aaabba $\rightsquigarrow$ false, aaabbab

(opt) How to never fail ? (1/2)

Each rule must accept all alphabet symbols

1. Complete rules so that each possible symbol is consumed *(with an appropriate action)*
2. Skip unexpected prefix and retry, or finally give up.

Transform the orignal rule $S \rightarrow a.S.b \mid \epsilon$ into

```
S → A . S . B
  | ϵ

A → 'a'
  | 'b'.A   // skip 'b' and retry
  // ϵ will clash with S → ϵ

B → 'b'
  | 'a'.B   // skip 'a' and retry
  | ϵ       // give up
```

(opt) How to never fail ? (2/2)

Check that ...

at each • every symbol of $\Sigma \cup \{\epsilon\}$ is accepted

```
S → • A • S • B
  | • ϵ

A → 'a'
  | 'b'.A   // skip 'b' and retry

B → 'b'
  | 'a'.B   // skip 'a' and retry
  | ϵ       // give up
```

check is performed by computing 1-prefixes

- $prefix(A) = \{a, b\}$
- $prefix(S) = \{\epsilon\} \cup prefix(A) = \{\epsilon, a, b\}$ ✓
- $prefix(B) = \{\epsilon, a, b\}$ ✓

(opt) Adding appropriate actions (1/2)

How to write a user-friendly parser that ...

- never fails with an exception ✓
- parses all the file and returns a boolean

Exercice (2min)

```
bool S → a=A . s=S . b=B {return ...;}
   | ϵ {return true;}

bool A → 'a' {return true;}
   | 'b'.A {return ...;} // skip b and retry


bool B → 'b' {return true;}
   | 'a'.B {return ...;} // skip a and retry
   | ϵ     {return ...;} // give up
```

(opt) Adding appropriate actions (1/2)

How to write a user-friendly parser that ...

- never fails with an exception ✓
- parses all the file and returns a boolean

Solution

```
bool S → a=A . s=S . b=B {return a & s & b;}
   | ϵ {return true;}

bool A → 'a' {return true;}
   | 'b'.A {return false;} // skip b and retry


bool B → 'b' {return true;}
   | 'a'.B {return false;} // skip a and retry
   | ϵ     {return false;} // give up
```

(opt) Repairing syntax errors (1/2)

How to write a user-friendly parser which repairs/shows syntax errors

Each parsed symbol is reprinted in black except

- for unexpected symbols
- for missing symbols

Exercice (2min)

```
bool S → a=A . s=S . b=B {return a & s & b;}
   | ϵ {return true;}

bool A → 'a' {print(...); return true;}
   | 'b'.A   {print(...); return false;}


bool B → 'b' {print(...); return true;}
   | 'a'.B   {print(...); return false;}
   | ϵ       {print(...); return false;}
```

(opt) Repairing syntax errors (2/2)

How to write a user-friendly parser which repairs/shows syntax errors

Each parsed symbol is reprinted in black except

- for unexpected symbols
- for missing symbols

Solution

```
bool S → a=A . s=S . b=B {return a & s & b;}
   | ϵ {return true;}

bool A → 'a' {print(a); return true;}
   | 'b'.A   {print(b); return false;}


bool B → 'b' {print(b); return true;}
   | 'a'.B   {print(a); return false;}
   | ϵ       {print(b); return false;}
```

(opt) A User-Friendly Parser that never fails (2/2)

General principle

1. Change each terminal t into a smart rule $T()$ that reprint the symbol in black, red of green.
2. Compute 1-prefix and complete rules so that each possible symbol is consumed
3. Remove non-determinism introduced by "give up" ϵ

Principle of rule completion

Skip unexpected prefix and retry, then finally give up.

```
NT → original
     // additional cases
     | Not_NT1.NT   {return false;}   // skip and retry
     | ϵ   {print(prefix(NT)); return false;} // give up

Not_NT1 → s {print(s)}   ∀s ∈ (Σ − prefix1(NT))
```

Note: If the original rule already had an ϵ case, keep it unchanged.

Parsers which compute

Hand-made parsers illustrated on 3 examples

1. Parsing and builing natural numbers
   - a simple but wrong solution
   - the correct but trickier parser
2. Parsing and evaluation of arithmetic expressions
3. Parsing and translation from C to Pascal

LALR parsers

- When it works, a LALR parser generator is the easy way
- a LALR engine is tricky
- A brief presentation at the end

1^{st} example : The Grammar of Natural Numbers

Definition using regular expression

```
Digit → '0' | '1' | ... | '9'

Nat   → Digit . (Digit)*
```

Exercise :

Equivalent Grammar without $(\ldots)^*$

1^{st} example : The Grammar of Natural Numbers

Definition using regular expression

```
Digit → '0' | '1' | ... | '9'

Nat   → Digit . (Digit)*
```

Exercise :

Equivalent Grammar without (...)*

Equivalent Grammar without (...)*

```
Digit       → '0' | '1' | ... | '9'

Nat         → Digit . Some_Digits

Some_Digits → Digit . Some_Digits
            | ϵ
```

Natural numbers : parsing & reconstruction (1/6)

- "123" is recognized as '1'.'2'.'3'
- it is easy to produce "123" by glueing the characters
- but we want the integer 123 not the string "123"

Parser as functions returning an integer

```
int Digit() ::= '0' { return ...; }
              | ...
              | '9' { return ...; }

int Nat() ::=
    | ... = Digit() ; ... = Some_Digits()
      { return ...; }

int Some_Digits() ::=
    | ... = Digit() ; ... = Some_Digits()
      { return ...; }
    | ϵ
      { return ...; }
```

Natural numbers : parsing & reconstruction (2/6)

Reconstruction means Computing the integer from

- the result of `d = Digit()`
- the result of `i = Some_Digits()`

Parser as functions returning an integer

```
int Digit() ::= '0' { return 0; }
              | ...
              | '9' { return 9; }

int Nat() ::=
    | d = Digit() ; i = Some_Digits()
      { return ...; }

int Some_Digits() ::=
    | d = Digit() ; i = Some_Digits()
      { return ...; }
    | ϵ
      { return ...; }
```

Natural numbers : parsing & reconstruction (3/6)

Wrong Proposal

```
int Nat() ::=
    | d = Digit() ; i = Some_Digits()
      { return 10 * d + i; }

int Some_Digits() ::=
    | d = Digit() ; i = Some_Digits()
      { return 10 * d + i; }
    | ϵ
      { return 0; }
```

Let's try

```
Some_Digits("2") ⇝ d=Digit("2"); i=Some_Digits("")
   ⇝ 10 * d + i = ...

Nat("12") ⇝ d=Digit("1"); i=Some_Digits("2")
   ⇝ 10 * d + i = ...
```

Natural numbers : parsing & reconstruction (3/6)

Wrong Proposal

```
int Nat() ::=
    | d = Digit() ; i = Some_Digits()
      { return 10 * d + i; }

int Some_Digits() ::=
    | d = Digit() ; i = Some_Digits()
      { return 10 * d + i; }
    | ϵ
      { return 0; }
```

Fail !

```
Some_Digits("2") ⇝ d=Digit("2"); i=Some_Digits("")
    ⇝ 10 * d + i = 20

Nat("12") ⇝ d=Digit("1"); i=Some_Digits("2")
    ⇝ 10 * d + i = 30
```

Natural numbers : parsing & reconstruction (4/6)

The function `Some_Digits` takes, as argument, the integer constructed from the already parsed digits.

Complete the ... with computations

```
int Nat() ::=
    | d = Digit() ; n = Some_Digits(...)
      { return ...; }

int Some_Digits(int i) ::=
    | d = Digit() ; n = Some_Digits(...)
      { return ...; }
    | ϵ
      { return ...; }
```

Natural numbers : parsing & reconstruction (5/6)

The ϵ case

The argument of `Some_Digits` is the value to return if nothing follows.

Solution (1/2)

```
int Nat() ::=
    | d = Digit() ; n = Some_Digits(...)
      { return n; }

int Some_Digits(int i) ::=
    | d = Digit() ; n = Some_Digits(...)
      { return n; }
    | ϵ
      { return i; }
```

Natural numbers : parsing & reconstruction (6/6)

Solution (2/2)

```
int Nat() ::=
    | d = Digit() ; n = Some_Digits(d)
      { return n; }

int Some_Digits(int i) ::=
    | d = Digit() ; n = Some_Digits(10*i+d)
      { return n; }
    | ϵ
      { return i; }
```

Check

```
Nat("234")
 ⇝ Digit("2") ; Some_Digits(2,"34")
  ⇝ Digit("3") ; Some_Digits(10*2+3,"4")
   ⇝ Digit("4") ; Some_Digits(23*10+4,"") ⇝ 234
```

JavaCC implementation


```
SKIP:  { " " }
TOKEN: { "0" | "1" | "2" | "3" | "4"
       | "5" | "6" | "7" | "8" | "9" }

int Parser() : { int n; } // declaration of var.
{ n=Nat() ; <EOF>   {return n;}
}

int Nat() : { int d,n; }
{ d=One_Digit() ; n=Some_Digits(d)    {return n;}
}

int Some_Digits(int i) : { int d,n; }
{ d=One_Digit() ; n=Some_Digits(10*i+d)  {return n;}
| /*epsilon*/    {return i;}
}

int One_Digit() : {}
{ "0"    {return 0;}
| ...
| "9"    {return 9;}
}
```

Parsing numbers in practise

The parser ($Nat : string \rightarrow int$) of natural numbers is great example

. . . for the purpose of a course

- it illustrates subtle problems
- on a simple grammar

The good way to parse numbers

- parse it as string = array of chars

```
"12345" = ['1'|'2'|'3'|'4'|'5']
```

- Then, compute the corresponding number with a for-loop using Horner scheme

```
Horner ['1'|'2'|'3'|'4'|'5'] =
  (((1 × 10 + 2) × 10 + 3) × 10 + 4) × 10 + 5
```

Summary

1. In the litterature, parsers which compute are called attributed grammars
   - result of parsing functions are called synthesized values
   - argument of parsing functions are called inherited values
2. Arguments of a parsing function cannot be avoided
   - when the order of computations is not the order of reading (e.g. for Natural numbers)
   - when you don't know what value to return in the ϵ case
3. Parser which computes is a powerful tools to solve many Data Processing Problems with few lines of code

2^{nd} example : A command line arithmetic calculator

The goal

- implementing a command line calculator
- for simple arithmetic expressions

```
# 3+ 5*4 + 3 * 6 +1    return
# 42
```

It start with a grammar of arithmetic expressions

```
Expr → Expr . '+' . Expr
     | Expr . '*' . Expr

Expr → Nat | '(' . Expr . ')'

Nat → ...
```

A command line arithmetic calculator (1/3)

A nice grammar of arithmetic expressions

```
Expr → Expr . '+' . Expr
     | Expr . '*' . Expr

Expr → Nat | '(' . Expr . ')'

Nat → ...
```

The parser must forbid infinite loop

- It must consume some characters between two recurivse calls.
- Rule of the form

  Expr → *Expr* . '+' . Expr

  is prohibited

A command line arithmetic calculator (1/3)

The parser must forbid infinite loop

- It must consume some characters between two recurivse calls.
- Rule of the form

  Expr → *Expr* . '+' . Expr

  is prohibited

A repaired grammar (step 1)

```
        Expr → NatOrPExpr . '+' . Expr
             | NatOrPExpr . '*' . Expr
             | NatOrPExpr

NatOrPExpr → Nat | '(' . Expr . ')'
```

A command line arithmetic calculator (1/3)

A repaired grammar (step 1)

```
          Expr → NatOrPExpr . '+' . Expr
               | NatOrPExpr . '*' . Expr
               | NatOrPExpr

  NatOrPExpr → Nat | '(' . Expr . ')'
```

The parser should be deterministic.

- Rule of the form

```
Expr → NatOrPExpr . '+' . Expr
     | NatOrPExpr . '*' . Expr
     | NatOrPExpr
```

is prohibited

A command line arithmetic calculator (1/3)

The parser should be deterministic.

- Rule of the form

```
Expr → NatOrPExpr . '+' . Expr
     | NatOrPExpr . '*' . Expr
     | NatOrPExpr
```

is prohibited

Repaired grammar (step 2)

```
        Expr → NatOrPExpr . Opt_op_Expr

Opt_op_Expr → '+' . Expr
            | '*' . Expr
            | ϵ

 NatOrPExpr → Nat | '(' . Expr . ')'
```

A command line arithmetic calculator (1/3)

Repaired grammar (step 2)

```
          Expr → NatOrPExpr . Opt_op_Expr

 Opt_op_Expr → '+' . Expr
             | '*' . Expr
             | ϵ

  NatOrPExpr → Nat | '(' . Expr . ')'
```

We must give priority to $*$ over $+$

- an *Expr* is a **sum of products** *(abbr. SoP)*
- a *Product* is a **product of factors** *(abbr. PoF)*
- a *Factor* is a **naturals or a (sub-expression)**

A command line arithmetic calculator (1/3)

We must give priority to $*$ over $+$

- an *Expr* is a **sum of products** *(abbr. SoP)*
- a *Product* is a **product of factors** *(abbr. PoF)*
- a *Factor* is a **naturals or a (sub-expression)**

Repaired grammar (step 3)

```
          Expr → SoP
           SoP → PoF . Opt_plus_SoP
Opt_plus_SoP → '+' . SoP
             | ϵ
           PoF → Factor . Opt_mult_PoF
Opt_mult_PoF → '*' . PoF
             | ϵ
        Factor → Nat | '(' . Expr . ')'
```

A command line arithmetic calculator (1/3)

Repaired grammar (step 3)

```
          Expr → SoP
           SoP → PoF . Opt_plus_SoP
Opt_plus_SoP → '+' . SoP
             | ϵ
           PoF → Factor . Opt_mult_PoF
Opt_mult_PoF → '*' . PoF
             | ϵ
        Factor → Nat | '(' . Expr . ')'
```

Priority is enforced by the transformation path

- Level of priority = the depth along the path
  $Expr \rightarrow Term \rightarrow Factor \rightarrow Nat \mid (Expr)$
- The deeper, the more implicit parenthesis there are
  $Expr \rightarrow (\ Term\ )_1 \rightarrow (\ Factor\ )_2$

A command line arithmetic calculator (2/3)

The final grammar

```
            Seed → Expr . '\n'
            Expr → SoP
             SoP → PoF . Opt_plus_SoP
             PoF → Factor . Opt_mult_PoF

    Opt_plus_SoP → '+' . SoP
                 | ϵ
    Opt_mult_PoF → '*' . PoF
                 | ϵ

          Factor → Nat
                 | '(' . Expr . ')'
```

A parser made of functions returning int (1/2)

```
int Factor() ::=
  | n= Nat()                    { return n; }
  | '(' ; e= Expr() ; ')'       { return e; }
```

A command line arithmetic calculator (3/3)

A parser made of functions returning int (2/2)

```
int Expr() ::= s= Sop() ; '\n' { return s; }

int SoP() ::=
  | p= PoF() ; s= Opt_plus_SoP(...)
    { return ...; }

int PoF() ::=
  | f= Factor() ; p= Opt_mult_Expr(...)
    { return ...; }

int Opt_plus_SoP(...) ::=
  | '+' ; s= SoP() { return ...; }
  | ϵ              { return ...; }

int Opt_mult_Expr(...) ::=
  | '*' ; p= PoF() { return ...; }
  | ϵ              { return ...; }
```

A command line arithmetic calculator (3/3)

Solution

```
int Expr() ::= s= Sop() ; '\n' { return s; }

int SoP() ::=
  | p= PoF() ; s= Opt_plus_SoP(p)
    { return s; }

int PoF() ::=
  | f= Factor() ; p= Opt_mult_Expr(f)
    { return p; }

int Opt_plus_SoP(int i) ::=
  | '+' ; s= SoP() { return i+s; }
  | ϵ              { return i; }

int Opt_mult_Expr(int i) ::=
  | '*' ; p= PoF() { return i*p; }
  | ϵ              { return i; }
```

3^{rd} example : Translation C $\rightarrow$ Pascal (1)

We consider the declaration of variables in C

```
Seed → Decl*

Decl → Type . Var.(',',.Var)* . Asg? . ';'

Asg → '=' . Value // Assignment

Value → Int | Float

Type → "float" | "int"

Var → ...
Int → ...
Float → ...
```

Removing regular expressions

No computations in regular expressions

It is mandatory to replace regular expressions by equivalent grammar rules before adding computations.

Kleene iterations : $N* \rightsquigarrow NStar$, $N+ \rightsquigarrow NPlus$

- $NStar \rightarrow N.NStar \mid \epsilon$
- $NPlus \rightarrow N.NStar$ in addition to the previous rule

optional $N? \rightsquigarrow OptN$

- $OptN \rightarrow N \mid \epsilon$

iteration with separator $N.(sep.N)* \rightsquigarrow N.MoreN$

- $MoreN \rightarrow sep.N.MoreN \mid \epsilon$

Translation C → Pascal (2)

The grammar without regular expression

```
Seed → DeclStar

DeclStar → Decl . DeclStar
         | ϵ

Decl → Type . Var . MoreVars . OptAsg . ';'

MoreVars → ',' . Var . MoreVars
         | ϵ

OptAsg → '=' . Value
       | ϵ

Value → Int | Float

Type → "float" | "int"
```

Translation C → Pascal (3)

From the grammar to a parser (easy)

```
void Seed() ::= DeclStar() ; '\EOF' {return;}

void DeclStar() ::=
 | Decl() ; DeclStar() {return;}
 | ϵ {return;}

void Decl() ::=
 | Type() ; Var() ; MoreVars() ; OptAsg() ; ';'
     {return;}

void MoreVars() ::=
 | ',' ; Var() ; MoreVars() {return;}
 |  ϵ {return;}

void OptAsg() ::=
 | '=' ; Value() {return;}
 | ϵ {return;}

...
```

Translation C → Pascal (4)

Now that we have a parser of variable declarations in the C language...

we can add computations to the parser to produce a string which represents the translation of declarations into the Pascal programming langage.

```
float x = 0.0; int y,z=0;
```

⇝

```
var x:real; x:=0.0;
var y,z:integer; y:=0; z:=0;
```

The expected results are strings that we glue with +

```
String Type() ::=
  | "float" { return "real"; }
  | "int"   { return "integer"; }
```

Translation C → Pascal (5)

Part 1 : Complete the ... with computations

```
String DeclStar() ::=
 | d= Decl() ; ds= DeclStar()
   { return ... }
 | ϵ
   { return ... }

String Decl() ::=
 | t= Type() ;
   v= Var() ;
   l= MoreVars() ;
   a= OptAsg(...) ;
      ';'
   { return ...
            ...
            ...
   }
```

Translation C → Pascal (5)

Part 1 : Solution

```
String DeclStar() ::=
 | d= Decl() ; ds= DeclStar()
   { return d + ds; }
 | ϵ
   { return ""; }

String Decl() ::=
 | t= Type() ;
   v= Var() ;
   l= MoreVars() ;
   a= OptAsg(...) ;
      ';'
   { return "var" + l.add(0,v).to_string_sep_by(",")
                  + ":" + t + ";"
                  + a;
   }
```

Translation C → Pascal (7)

Part 2 : Complete the . . . with computations

```
list<String> MoreVars() ::=
// local variable declarations
 { String ...;
   list<String> ...;
   list<String> ...;
 }

 | ',' ; v= Var() ; l= MoreVars()
   { return ... }

 | ϵ
   { return ... }
```

Translation C → Pascal (7)

Part 2 : Solution

```
list<String> MoreVars() ::=
// local variable declarations
 { String v;
   list<String> l;
   list<String> empty = new list<String>();
 }

 | ',' ; v= Var() ; l= MoreVars()
   { return l.add(0,v); }

 | ϵ
   { return empty; }
```

Translation C → Pascal (8)

Part 3 : Complete the ... with computations

```
String OptAsg(list<String> l) ::=
  | '=' ; val = Value()
    { return ...
             ...
    }
  | ϵ
    { return ... }

Val Value() ::=
  | i = Int()    { return new ... }
  | f = Float() { return new ... }

String Ident() ::=  ...
Val Int() ::= ...
Val Float() ::= ...
```

Translation C → Pascal (8)

Part 3 : Solution

```
String OptAsg(list<String> l) ::=
  | '=' ; val = Value()
    { return l.to_string_followed_by
                ( ":="+val.to_string()+";" );
    }
  | ϵ
    { return ""; }

Val Value() ::=
  | i = Int()    { return new Ival(i); }
  | f = Float() { return new Fval(f); }

String Ident() ::= ...
Val Int() ::= ...
Val Float() ::= ...
```

(opt) LALR parser generators, the easy way

We dived into the engineering details of parsers which compute.
Actually, the exists an easier way.

The family of LALR (Look Ahead Left-to-right) parser generators

. . . greatly simplify the development of parsers

- It started with Lex & Yacc (Yet Another Compiler-Compiler) based on Donald Kuth's parsing algorithm.
- Variants of the lexer generator (Lex) and the parser generator (Yacc) exist for almost every programming language (C, Java, Ada, Ocaml, Haskell).

A well-chosen example follows

- We go back to our parser of arithmetic expressions
- using Lex/Yacc this time.

(opt) The arithmetic calculator in Lex/Yacc

Lex is a generator of lexical analyzers (= lexers)

- a lexer eases the work of the parser
- by splitting the buffer of chars into a stream of tokens
- a token is an atom of recognition for the parser

Yacc is a generator of syntactic analyzers (= parsers)

- Yacc helps writing parsers which computes
- non-terminals are functions which parse and compute
- they return results but have no argument
- `$i` refers to the result of the i^{th} non-terminal

(opt) The arithmetic calculator in Lex/Yacc

The lexer of arithmetic expressions (in OcamlLex syntax)

```
rule tokenizer(buffer) = parse
| ' '   { tokenizer(buffer) }
        // recursive call to skip ' '
| '\n'  { EOL }
| '+'   { PLUS }
| '*'   { MULT }
| '('   { LP }
| ')'   { RP }
| ['0'-'9']+ as string { NAT(string) }
```

(opt) The arithmetic calculator in Lex/Yacc

The parser of arithmetic expressions (in OcamlYacc syntax)

```
token <string> NAT
token PLUS MULT LP RP EOL
left PLUS  /* priority 1 and left associativity */
left MULT  /* priority 2 and left associativity */
start seed
type <int> seed

seed:
  | expr EOL { $1 }
;
expr:
  | expr PLUS expr { $1 + $3 }
  | expr MULT expr { $1 * $3 }
  | NAT            { int_of_string($1) }
  | LP expr RP     { $2 }
;
```

(opt) Summary

The many advantages of Lex/Yacc

- it accepts ambiguous and left-recursive grammars
- it deals with priorities and associativity of operators
- it produces efficient parsers

If it is so easy with Yacc, why bothering us with others ways ?

- Yacc does not allow parameters in parsing functions
- Some grammars produce LR conflicts which are difficult to solve

PRAGMATISM : Choose the most appropriate tool for your needs

- It is worth starting with Yacc
- Change for a hand-made parser if you can't get Yacc to solve your problem