


# Introduction au Parsing

Victor “Hippolyte” Boudon

02.07.2015






Qui sont les Koalas et que vous veulent-ils ?

Concepts de base du parsing

Implémenter un parseur

Application au 42sh



Qui sont les Koalas et que veulent-ils ?

Concepts de base du parsing

Implémenter un parseur

Application au 42sh

## Les Koalas


- ▶ Koala est un acronyme qui signifie Kind Of Advanced Language Assistant
- ▶ Les Koalas sont un groupe d'assistants dirigés par Jason Brillante
- ▶ Les Koalas encadrent les modules de C++, kOOC, Java, UML, C#, et OCAML



## Pourquoi cette conférence ?

- ▶ Pour vous apporter des connaissances et une méthodologie intéressante à la réalisation du 42sh
- ▶ Démystifier le parsing





Qui sont les Koalas et que vous veulent-ils ?

Concepts de base du parsing

Implémenter un parseur

Application au 42sh

# Parsing ?

- ▶ Le parsing, ou analyse syntaxique, est le processus d'analyse d'une string, qu'elle soit issue d'un langage naturel ou informatique, conformément aux règles d'une grammaire formelle.
- ▶ Le parsing débouche généralement sur la création d'un Abstract Syntax Tree (AST).



# Langage

Un langage est un ensemble de mots (séquences de symboles) choisis sur un alphabet.

Le but de notre parsing est multiple :

- ▶ Valider que le mot d'entrée fait bien partie du langage
- ▶ Structurer les informations de ce mot pour en faciliter l'utilisation.





# Grammaire Formelle

Une grammaire formelle est un des moyens de représenter un langage. Elle est composée de plusieurs parties:

- ▶ Un ensemble fini de symboles, appelés symboles terminaux (qui sont les "lettres" de l'alphabet).
- ▶ Un ensemble fini de symboles, appelés non-terminaux.
- ▶ Un élément de l'ensemble des non-terminaux, appelé axiome, noté conventionnellement  $S$  qui est la règle d'entrée de la grammaire.
- ▶ Un ensemble de règles de production, qui sont des paires formées d'un non-terminal et d'une suite de terminaux et de non-terminaux

Voici un exemple de grammaire reconnaissant des palindromes sur l'alphabet  $\{a, b\}$ .

$$S \rightarrow aSa$$

$$S \rightarrow bSb$$

$$S \rightarrow \lambda$$



## Extended Backus Naur Form

L'EBNF est l'un des formalismes les plus répandus pour représenter une grammaire.

Par exemple, voici une grammaire reconnaissant une opération arithmétique:

```
1 entry = num , ope , num;  
2 num = ('1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' | '0')+;  
3 ope = '+' | '-' | '/' | '\%' | '*';
```

Quelques choses à noter ici:

- ▶ Une règle est définie par un nom (ici "entry"), suivie d'une énumération des éléments composants la règle.
- ▶ Le symbole "|" représente une alternative. Ici, la règle op valide soit un "+" soit un "-", etc...
- ▶ Deux règles peuvent être concaténées : la règle entry valide num suivie de ope, suivie de num.
- ▶ Le ";" signifie la fin d'une règle

## Les opérateurs de l'EBNF

- ▶ "\*" : Répétition (0 ou n fois)
- ▶ "+" : Au moins une fois
- ▶ "-" : Absence
- ▶ "," : Concaténation
- ▶ "|" : Choix

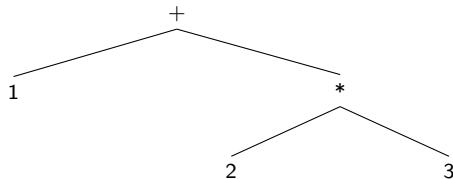
# Abstract Syntax Tree (AST)

L'un des buts du parsing est de structurer les données d'une manière permettant de s'en servir facilement. Pour cela on utilise un AST.

Par exemple, l'expression

$$1 + 2 * 3$$

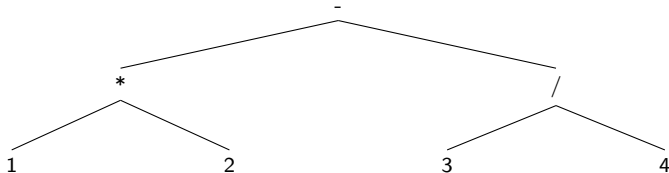
devra résulter en l'arbre suivant :




## AST, suite

La construction de notre AST nous permet également de gérer les priorités : les opérations de plus fortes priorités devront être placées le plus bas dans l'arbre.

L'expression  $1 * 2 - 3 / 4$  produira :





Qui sont les Koalas et que veulent-ils ?

Concepts de base du parsing

Implémenter un parseur

Application au 42sh

## Les différents algorithmes de parsing

Il existe deux grandes méthodes de parsing:

- ▶ Bottom-up (LR, LALR): Ces parseurs sont basés sur des tables d'états, de transitions ainsi qu'une pile. Ces parseurs sont à la fois performants et très complexes.  
Habituellement on utilise des outils comme yacc pour les générer.
- ▶ Top-Down (LL, PEG): ces parseurs retracent la dérivation de l'axiome vers les éléments terminaux.



## Représenter l'arbre en C

Tout d'abord, nous devons définir la structure de notre arbre.

```
1  typedef struct s_tree {  
2      struct s_tree *left;  
3      struct s_tree *right;  
4      enum e_type type;  
5      void *value;  
6  } t_tree;  
7  
8  enum e_type {  
9      NUM, OPE  
10 };
```



## Utiliser notre arbre

Une fois l'arbre construit, l'exécution de l'arbre tient en une simple fonction recursive:

```
1  int
2  eval_tree(t_tree const *t, int *err)
3  {
4      if (!t) {
5          *err = EMPTY_TREE;
6          return 0;
7      }
8      if (!is_ope(t))
9          return *(int *)t->value;
10     return do_op(*(char*)t->op, eval_tree(t->left), eval_tree(t->right));
11 }
```



# La grammaire d'une expression mathématique

```
1 entry = num (ope , entry)?;  
2 ope = ('+' | '-' | '/' | '*' | '%');  
3 num = ('1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' | '0')+;
```

## Gérer les priorités

Ici, nous allons adapter la grammaire pour gérer les priorités:

```
1 entry = ope_low;  
2 ope_low = ope_high , ( ('+' | '-') , ope_high)*;  
3 ope_high = num , (('*' | '/' | '%') , num) *;  
4 num = ('1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' | '0')+;
```

## règles et fonction

- ▶ Dans l'implémentation de notre grammaire, chaque règle sera représentée par une fonction C.
- ▶ Cette fonction devra retourner un noeud de l'AST si l'entrée valide la règle, et NULL sinon.



## Stream, règle et consommation

- ▶ L'entrée du programme peut être représentée comme un char \* associé à un compteur. Ce compteur marque la position actuelle dans le stream.
- ▶ Une règle consomme le stream tout en le validant. Lorsqu'une règle valide un caractère, elle le consomme. Si une règle échoue, le stream doit rester intact.



## Contexte et alternatives

```
1 entry = first_rule | second_rule;  
2 first_rule = "aa"  
3 second_rule = "ab"
```

Lorsque l'on rencontre une alternative, il est important de sauvegarder le contexte courant : en effet, si la première alternative échoue, il doit être possible de revenir au contexte initial pour valider l'autre alternative.

## Construire l'arbre

Chaque règle sera construite comme une fonction, qui en plus de valider son entrée, insère le noeud associé dans l'arbre.

Voici la fonction qui se chargera de valider la règle num.

```
1 | num = ('1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' | '0')+;
```

```
1 | t_tree *  
2 | parse_num(char const *stream, int *position)  
3 | {  
4 |     int back = *position;  
5 |     if (!isdigit(stream[back]))  
6 |         return NULL;  
7 |     while (isdigit(stream[back]))  
8 |         back++;  
9 |     t_tree *value = create_tree(substr(stream, *position, back), NUM, NULL,  
10 |                                NULL);  
11 |     *position = back + 1;  
12 |     return value;  
12 | }
```




## Une règle un poil plus complexe...

```
1 ope_low = ope_high , ( ('+' | '-') , ope_high)*;
```

```
1 t_tree *
2 parse_ope_low(char const *stream, int *position)
3 {
4     int back = *position;
5     t_tree *value = parse_ope_high(stream, &back);
6     if (!value)
7         goto err;
8     char ope = stream[back];
9     while (ope == '+' || ope == '-') {
10         t_tree *right = parse_ope_high(stream, &(&back));
11         if (right == NULL)
12             goto err;
13         value = create_tree(chardup(ope), OPE, value, right);
14         ope = stream[back];
15     }
16     *position = back + 1;
17     return value;
18
19 err:
20     delete_tree(value);
21     return NULL;
22 }
```







Qui sont les Koalas et que veulent-ils ?

Concepts de base du parsing

Implémenter un parseur

Application au 42sh

## Le 42sh dans tout ça...

Le but du projet est de développer un interpréteur capable d'exécuter des commandes de cette forme :

```
1 | ls -l | cat -e
2 | ls -l | cat -e | wc -l
3 | > toto ls
4 | << toto cat -e
5 | ls > toto -la < tutu
6 | ls -l | cat -e > tutu && echo "tutu" | wc -c || echo "DONE"
```

A quoi doivent ressembler les arbres associés à ces commandes ?...



## La grammaire du 42sh...

Voici une grammaire simplifiée (et pas tout à fait exacte...) de la grammaire d'un 42sh.

```
1 entry = [expr]+;
2 expr = semicolon_expr;
3 semicolon_expr = and_expr [';' expr]+;
4 and_expr = or_expr ["&&" and_expr]?;
5 or_expr = pipe_expr ["||" or_expr]?;
6 pipe_expr = command ['|' pipe_expr]?;
7 command = word [word]* [{">" | ">>" | "<" | "<<"} word]*;
8 word = ['a'..'z' | 'A'..'Z' | '_' | '-' ]+;
```

# Questions ?

