OcamlYacc - Grammars

November 4, 2016

1 Intro

Ocamlyacc is a parser which builds the pushdown automaton for ascendant analysis. In this session, the goal is to use Ocamlyacc for several simple grammars.

In the archive, you will find a model of the full analyser for the following grammar G of axiom S' :

$$S' \rightarrow S\$^k \quad (0)$$
$$S \rightarrow aUb \quad (1)$$
$$U \rightarrow c \quad (2)$$
$$U \rightarrow Uc \quad (3)$$

In the archive, you have

- a Makefile (to use in order to compile/build all files)
- `l`exer.mll Comparing to the lexer of the previous session, this one builds the ocaml data structure of the analysed word (token, lexical unit...). Thus, the symbole/character 'a' is translated in the ocaml value A:

```
{
        open Parser
}

rule main = parse
                'a'             { A }
        |       'b'             { B }
        |       'c'             { C }
        |       '\n'            { EOF }
        |       eof             { EOF }
        |       _               { failwith "bad character !" }
```

- parser.mly The first lines "token" corresponds to the terminals symbols. The type is used to decorate the S' non-terminal with a type in order to obtain a value at the end of the analysis. The start is to specify the axiome.

  Each rule is associated with an action, for instance, the action of the axiom has the type defined in the first lines. In our example, actions are just printing the taken action.

  Note that terminals are in lower case, non-terminal in upper case.

```
%token A
%token B
%token C
%token EOF
%type <unit> sp
%start sp

%%

sp: s EOF       { () }
;

s:      A u B           { print_string "S -> a U b\n" }
;

u:      C               { print_string "U -> c\n" }
```

```
|       u C             { print_string "U -> U c\n" }
;

%%
```

- prog.ml which calls the lexer.

To test this analyser, execute the Makefile (make) and then execute prog: ./prog
Test it on several words : *acb*, *accccb*, *ab*, *b*...

2 Exploring files

Have a look on the file parsing.output and you will see the pushdown automaton for the grammar.

3 Adding action

Now, we want to add an action to the grammar: counting the number of "c" (it could be done at the lexer level, but the choice is made here to introduce the action at the parser level).

```
1  %token A
2  %token B
3  %token C
4  %token EOF
5  %type <unit> sp
6  %start sp
7
8  %%
9
10 sp: s EOF        { print_string ("Total of c: "^(string_of_int $1)^ "\n") }
11 ;
12
13 s:       A u B            { print_string "S -> a U b\n"; $2 }
14 ;
15
16 u:       C                { print_string "U -> c\n"; 1 }
17 |       u C                { print_string "U -> U c\n"; $1+1 }
18 ;
19
20 %%
```

We start to add, line 16, the value of reading a single "c": 1. At line 17, the value of the uC is the value recursively obtained by u ($1 because it is the first term of the right-hand side of the rule) plus the value of 1. In line 13, the action transmits the value of u ($2). And line 10 will print the final value.

4 Other grammars

For the following grammars, starting from a copy of the "model" directory, write the analyser. Start with simple actions like printing the name of the used rule.

4.1 Grammaire 1

S → U V a
U → a
V → W | b
W → λ

4.2 Grammaire 2

S → a U | U b | a b
U → c U | λ

4.3 Grammaire 3

S → U a | b U c | V c | b V a
U → d
V → d

For this grammar, there are some conflicts. Find why and a solution to suppress the conflicts.

4.4 Grammaire 4 (optionnel)

S → U
V → λ
W → λ
U → V W U | a

For this grammar, there are some conflicts. Find why and a solution to suppress the conflicts.