Compiler Construction (List 5)

Hans de Nivelle

18.11.2015

1. (a) In the programming language Lisp, everything is a list. The empty list has form ( ) or **nil**. Non-empty lists have form (L_1), $(L_1\ L_2)$, $(L_1\ L_2\ L_3)$, etc.
   Give a complete grammar for Lisp. The elements of a list can be atoms, numbers, or lists by themselves. (You may ignore the existence of dotted pairs and arrays.)

   (b) Give a grammar for the language of Prolog-style lists. Lists have form
   $$[\,],\ [L],\ [L_1, L_2],\ [L_1, L_2, L_3],\ \text{etc.}$$
   The elements of the lists can again be lists by themselves.

   (c) Consider the language consisting of functional expressions of form c, and $f(t_1, \ldots, t_n)$, with $n > 0$, and $t_1, \ldots, t_n$ functional expressions by themselves. Give a grammar for this language.

2. (a) Give attribute functions for the grammars in Task 1. Take into account that $(L_1 \cdots L_n)$ denotes $\mathbf{cons}(L_1, \mathbf{cons}(L_2, \ldots \mathbf{nil}))$. Give a derivation for the list
   $$(\mathbf{car}\ (\mathbf{quote}\ (1\ 2\ 3))).$$

3. (a) Give DFA-based grammars for the grammars of Task 1. Give the FIRST and FOLLOW sets for every non-terminal occurring in each of the grammars, and the set of nullable symbols.

   (b) Using the top down parser obtained from the DFA-based grammar for Lisp, parse some expressions, e.g.

```
( a b c )
(( a ) ( b ) c )
( set a ( quote b ))
```

4. Download **top_down.tar.gz** from the course homepage, and make sure that it compiles. There are a **tokenizer** class, a **token** class and a **parsestack** class. Tokens have attributes, but you don't need to worry about them because we will not be concerned with attributes in this exercise.

In file **parser.cpp** is a main file that tests the tokenizer.

Use it to implement a DFA-based grammar for Lisp. Test it on some expressions, e.g.

```
( a b c )
(( a ) ( b ) c )
( set a ( quote b ))
( define ( abs x ) ( if ( < x 0 ) ( - x ) x ))
```

Note: This is an exercise about top down parsing, not about hacking. I am aware that LISP is simple enough to write a parser without using any systematic methods, please stay with the formalism.