Exercise Compiler Construction (List 7)

Hans de Nivelle

02.12.2015

1. Consider the grammar $\mathcal{G} = (\{\hat{S}, S, T, U, \#, +, *, \mathrm{id}, (,)\}, \hat{S}, R)$ with $R =$

   $\hat{S} \rightarrow S\#$
   $S \rightarrow S + T$
   $S \rightarrow T$
   $T \rightarrow T * U$
   $T \rightarrow U$
   $U \rightarrow \mathrm{id}$
   $U \rightarrow (S)$

   (a) Download Maphoon from the course homepage, have a look at the manual.

   (b) Prepare a file `simple.m` containing the grammar given above, and run it through Maphoon. In subdirectory `examples` are a few examples.

   (c) Read the output of Maphoon. How many states does the prefix automaton have?

2. (a) In file `example_calculator.tar.gz` is a grammar for a calculator.

   (b) If you try to run the calculator (`./calculator`), it crashes. The reason is the fact that no attribute constraints have been specified. When an attribute is unspecified, the parser assumes that the attribute is absent. Because the tokenizer sometimes returns tokens with attributes, the parser crashes.

   Modify the attribute constraints of the tokens E,F,G,H, LISTARGS, IDENTIFIER, NUMBER in such a way that the parser does not crash anymore. For E,F,G,H, you should allow 0 or 1 value. The reason for allowing 0 values is to represent an undefined value. Undefined values are caused by division by zero, lookup of undefined variables, or computing ! of a non-integer or negative number.

   For LISTARGS, you should allow an arbitrary number of values. For ID and SCANERROR, you should allow exactly 1 string. For NUM, you should allow exactly one value.

   Currently, there is no nice way for quitting the calculator. Do not worry about that.

   (c) Write the attribute functions for the rules involving E,F,G,H. A few have been done already, so you can use those as example. If you want so see what is going on in the parser, you can set `#define MAPH_DEBUG 1` in file `parser.cpp`.

   (d) Why is there no error if you type `-1 ! ;` ?

3. (a) Type some string that causes a syntax error in the calculator, for example `a b c ;` . What do you see?

   (b) Verify that the parser gives up after 10 unparsable tokens. Change this number into 15, and check that it is indeed 15.

   (c) Write the attribute function for LISTARGS. Add some more functions to the rule `H : IDENTIFIER LPAR LISTARGS RPAR` .

4. Instead of evaluating expressions immediately, one can construct a parse tree. For this, the class **tree** can be used.

   Modify file **parser.m** in such a way that the parser constructs a parse tree instead of evaluating the expressions. (Make sure to keep your solution to List 2!) In order to do this, the attributes of E,F,G,H, LISTARGS must be redefined into **tree**, and the attribute functions must to be adapted.

   The parse tree can be printed in the rule `Command : E SEMICOLON`.

5. Write an evaluation function for parse trees. (For example by adding a method `double eval( varstore& ) const` to **class tree**.

   (It can be a simple, recursive procedure, it should be only $\pm$ 30 lines of code.)

   If you want, you can throw an `std::runtime_error` when the result is undefined, and catch it in the attribute function of rule `Command : E SEMICOLON`.