Exercise Compiler Construction (9)

Hans de Nivelle

Due: 04.12.2013

In the previous List 8, we made a parser for Lisp expressions. In this task, we will write an evaluator for Lisp expressions. Wikipedia contains an explanation of Lisp. (`http://en.wikipedia.org/wiki/LISP`). The evaluator must be interactive: It reads a Lisp expression, evaluates it, and prints the result. After that, it reads a new expression.

Since in the previous task you wrote two parsers, you can choose the one that you like most.

The evaluation function needs two **varstores**, one for global variables and one for local variables. They must be passed as reference. If you used Maphoon in your prefered parser, then the easiest solution is to modify the grammar in such a way that it accepts only a single expression. Since it returns the result, you can evaluate the expression, and after that call the parser again. Otherwise you would have to pass the **varstores** as global parameters which is more difficult.

If you wrote a recursive descend parser, you have to add local and global as **varstore** references by yourself.

list eval(list E, varstore& global, varstore& local):

- If E is an identifier, then first look up E in local, and return the found value. If there is no value, then look up E in global, and return the value in global. If there is still no value, then return 'undefined', or throw an exception. If you decide to throw exceptions, you have to catch it in **main**.
- If E is a number, then return E.
- If E has form (if c t_1 t_2), then evaluate c. If the result equals 't', then return the evaluation of t_1, otherwise return the evaluation of t_2.
- If E has form (set A t), then evaluate t. Let t' be the result. Assign $A := t'$ in global. If A does not exist yet, it needs to be created.
- If E has form (setq A t), then assign $A := t$ without evaluating t.
- If E has form (quit), then quit from the interpreter.

- If E has form (define f (V_1 $\cdots$ V_n) E), then assign $f :=$ (lambda ($V_1 \cdots V_n$) E) in the global varstore.
- If E has form (f $t_1 \cdots t_n$), with $n \geq 0$ and f is a primitive function (listed below), then first evaluate all t_i. Let t'_i be the results of the evaluation. Apply the primitive function on the results. Note that Lisp is untyped (or dynamically typed), and that you sometimes have to check if the arguments are of proper type.
- If E has form (f $t_1 \cdots t_n$), and f occurs in the global varstore as an expression of form (lambda($V_1 \cdots V_n$) E), then
  1. evaluate all of the t_i. Call the results t'_i.
  2. Create a backtrack point in the local varstore. (It is shown in main.cpp how to do this.)
  3. Add the initializations $V_i := t_i$ to the local varstore.
  4. Evaluate E with the extended local varstore.
  5. Let the backtrack point go out of scope, and return the result of the evaluation.

Add some nice primitive functions,
plus,times,minus,divides, lessthan, greaterthan, equals, notequals, car,cdr,cons.

Numerical functions have to check that their arguments are indeed numerical, and create an error message when the arguments are of wrong type.

You can also add **or** and **and**, but they have to evaluate lazily, so you would have to add them as separate cases to the evaluator.

Show that the following functions work:

```
(define fact ( N )
   ( if ( equals N 0 ) 1
        ( times N ( fact ( minus N 1 ))) ));

( fact 6 );
--> 720

( set A ( fact 5 ));
--> 120

A;

--> 120

( define append ( L M )
     ( if ( equals L nil ) M
          ( const ( car L ) ( append ( car L ) M )) ));
```

Note: The first version of **list.cpp** contained a bug in
`std::ostream& operator << ( std::ostream& stream, list l )` The last lines of the print function should be as follows:

```
for( auto p = contents. begin( ); p != contents. end( ); ++ p )
 {
    stream << " " << *p;
 }
 if( l. isstring( ) && l. getstring( ) == "nil" )
    stream << " )";
 else
    stream << ". " << l << " )";
 return stream;
```

Note2: The lisp datastructure uses simple garbage collection based on reference counting. This is possible because we don't allow replacement. The original Lisp allowed replacement of car and cdr in a cons cell. This was a rather bad idea, because it simultaneously modifies all lists that contain a reference to the modified cons cell. In addition, it enables the creation of disconnected cycles which would keep each other alive, and which are harder to garbage collect. Since we don't have that, **class list** can use reference counting.