Compilers 2014-2015 Practicum 1
February 18, 2015

An Interpreter in a Functional Programming Style

Contact info:
Address *ELIS (floor -3), Technicum (building I), Sint-Pietersnieuwstraat 41*
Email *compiler@elis.ugent.be*

IMPORTANT: You should always hand in your solutions using the Dropbox functionality of Minerva, both an intermediate version at the end of the lab as well as a final one before the deadline. Use the `make_tarball.sh` script to generate an archive, and make sure you send it to the teaching assistants.

DEADLINE: February 24, 23:59.

1 Goal

The goal of this exercise is to implement an interpreter for very simple programs without any loops or jumps (so called straight-line programs). To avoid parsing the textual representation, we will provide the programs in a tree structure, the abstract syntax tree. Later on, you will notice that this representation is a very important concept in a compiler. In this and all the following exercises, we will use the C programming language. On top of that, we will use a functional programming style in this exercise. This means that you will have to write code that has no side effects. More specifically, you only assign a value to a variable during initialisation and this value should not change afterwards (although separate declarations are allowed). The same accounts for the members of structs. To simplify the initialisation of a struct, you can create a constructor function. We give examples of this in the initial files.

2 The language

The language is described by the following grammar:

lhs	rhs	meaning
stm	*stm;stm*	CompoundStm
stm	id := *exp*	AssignStm
stm	print (*expList*)	PrintStm
exp	id	IdExp
exp	num	NumExp
exp	*exp binop exp*	OpExp
exp	(*stm,exp*)	EseqExp
expList	*exp,expList*	PairExpList
expList	*exp*	LastExpList
binop	+	Plus
binop	-	Minus
binop	*	Times
binop	/	Div

A program in this language is in fact just a statement, and so the root of the abstract syntax tree will be a statement vertex. The semantics of this language are mainly self-explaining, but some of the constructions require additional information:

- a PrintStm will evaluate the expressions in the expList from left to right and will print the results one after the other with a space in between. At the end of the output a newline is printed. When evaluating subexpressions, one has to deal with the possible side effects of the previously evaluated expressions and statements.
- in an EseqExp the statement *stm* is executed first, and only then *exp* is evaluated, but the side effects of the execution of *stm* are already visible!
- in an OpExp the left argument is evaluated first, after which the right argument is evaluated, but the side effects of the execution of the left argument are already visible!

3 Assignment

Download the initial files from Minerva, unpack the tarball and inspect the code. In `slp.h` you'll find the definitions of the data structures of the abstract syntax tree. In `main.c` you'll find the functions `interpStm` and `interpExp`, that you will have to implement. These functions take two arguments: a statement or an expression and an `A_table`. This last argument represents the environment in which the statement or the expression has to be evaluated. This environment is nothing more than a linked list with defined variables and their current value.

In order not to break the functional programming style, the old values of the variables are not deleted from the environment: each time an assignation is done, the new variable-value pair is inserted at the head of the list. The current value of a variable can then be found by walking the list from the beginning until the first occurrence of the variable is found. To help you with this, we have provided two functions, `lookup` and `update`, in `main.c`.

The `interp*` functions return a `struct IntAndTable`. This struct contains the value of the evaluated expression (in case of a statement you can fill out a random value), and the new environment after the evaluation of the expression or the statement. Once you have inserted your code, you can compile everything using the makefile and the `make` command. This will create an executable that is called `main` which will execute the straight line programs from `prog1.c` and `prog2.c`.

If everything is working properly, the output of these programs should be:

```
Output program 1:
-----------------
8 7
80

Output program 2:
-----------------
8 8 8 8 8 8
88 88 88 88
888 888 888
```

```
88888 88888
888 888 888
88 88 88 88
8 8 8 8 8 8
```

Try other programs by changing `prog1.c` until you have convinced yourself that your interpreter works well.

Note: grading will be done based on your `main.c` file, other files mustn't be modified and should not be included in the archives you hand in.