Exercise sheet 3
for Monday 14 February and Wednesday 16 February 2011

Last update 2011-02-16

Hand in your solutions by sending a single zip file to sestoft@itu.dk by Friday 25 February 2011. The file name must be **`SASP-03-yourname.zip`**; for instance `SASP-03-OleHansen.zip`.

Either just briefly describe the necessary changes, with code fragments showing the code changes you have made; **or** include the full edited files including easily searchable comments that clearly indicate what changes you have made and for which exercise.
All the subproblems below concern extensions to the simple expression language:

- whose lexer and parser specification is in file Expressions.ATG
- whose abstract syntax etc is in file Expressions.cs
- whose abstract stack machine is in file Machine.cs

In each subproblem you must:

- extend the Coco/R lexer and parser specification in Expressions.ATG;
- extend the interpretation methods (`Eval`),
- extend the static checking methods (`Check`),
- extend the compilation methods (`Compile`), all in file Expressions.cs;
- and, if necessary, modify other parts of Expressions.cs.

However, it should not be necessary to modify the abstract stack machine in Machine.cs.

Getting started

Download the following files:

- Coco/R for C#, that is, Coco.exe, Scanner.frame and Parser.frame, from the Coco/R homepage
- Expressions.ATG, the lexer and parser specification
- Expressions.cs, the expression interpreter, checker and compiler
- Machine.cs, the abstract stack machine
- e1.txt, a file containing the expression: 2 + 3 * 4

To check that everything works out of the box, do the following (assuming you are using Microsoft .NET; for Mono, replace `csc` by `gmcs` and use `mono` to run `.exe` files, including Coco.exe):

```
Coco.exe -namespace Expressions Expressions.ATG
csc Expressions.cs Scanner.cs Parser.cs
Expressions.exe ex1.txt run
Expressions.exe ex1.txt check

Expressions.exe ex1.txt compile
csc Machine.cs
Machine.exe a.out
```

Extending the expression language

Exercise 3.1 Add the modulus operator (`e1%e2`) to the expression language. The operator has the same meaning as in C#. The operator should have the same precedence as multiplication (`e1*e2`) and integer division (`e1/e2`). Note that the stack machine has a `MOD` instruction with a suitable effect.

Exercise 3.2 Add a logical "and" (`e1&e2`) operator, which evaluates `e1` and `e2` and whose result is true if both `e1` and `e2` evaluate to true, and false otherwise. This operator has lower precedence than the relational operators (`==`, `!=`, and so on). Note that when implementing Compile, you can use the stack machine's `MUL` instruction to implement logical `AND`, if you assume that false is represented by 0 and true is represented by 1.

Exercise 3.3 Consider the logical "or" (`e1|e2`) operator, which evaluates `e1` and `e2` and whose result is false if both `e1` and `e2` evaluate to false, and true otherwise. One question is: How do you compile logical "or" to code for the stack machine? That is, do the existing instructions for the stack machine suffice to implement logical "or" such that it preserves the invariant that false is represented by 0 and true is represented by 1?

Exercise 3.4 Add simple let-expressions to the expression language. A possible syntax might be:

```
let x = e1 in e2 end
```

This expression should be evaluated by evaluating `e1`, binding the value to `x`, and then evaluating `e2`.

The expression should be type checked by finding the type for `e1`, and then using that as the type of `x` when finding the type of `e2`; the type of `e2` is the type of the entire let-expression.

Hints: To implement `Eval`, use `REnv` methods `AllocateLocal` and `PopEnv`. To implement `Check`, use `TEnv` methods `DeclareLocal` and `PopEnv`. To implement `Compile`, use `CEnv` methods `DeclareLocal` and `PopEnv`; and generate `SWAP` and `INCSP(-1)` stack machine instructions to remove the let-bound variable from the stack.

Make sure that complex expressions get parsed and evaluated correctly. For instance, each of these expressions should evaluate to 20:

```
let x = 2 in let y = x+3 in y*4 end end
let x = let y = 2 in y+3 end in x*4 end
let x = 2 in let x = x+3 in x*4 end end
let x = let x = 2 in x+3 end in x*4 end
```

If you have solved the previous exercises, you may check this by evaluating:

```
  let x = 2 in let y = x+3 in y*4 end end == 20
& let x = let y = 2 in y+3 end in x*4 end == 20
& let x = 2 in let x = x+3 in x*4 end end == 20
& let x = let x = 2 in x+3 end in x*4 end == 20
```

Also, these let-expressions should be type-correct:

```
let x = 2+3 in x > 7 end
let x = 2==2 in x & x end
```

whereas this one is not type-correct:

```
let x = 2+2 in x & x end
```

Exercise 3.5 Add conditional expressions in the lexer and parser specification, and define abstract syntax for the conditional expressions in Expressions.cs, for instance `IfElseExpression : Expression`. In this subproblem you can ignore the `Eval`, `Check`, and `Compile` methods; just let them throw Exception("Not implemented").

For the concrete syntax you may choose an SML/F#-style syntax such as:

```
if e1 then e2 else e3
```

or a C/C++/Java/C#-style syntax such as:

```
e1 ? e2 : e3
```

In the latter case you'll probably find parsing much easier if you always require parentheses around the expression, as in

```
( e1 ? e2 : e3 )
```

Exercise 3.6 Implement `Check` and `Eval` methods for conditional expressions as in Exercise 3.5.