Assigment 1: Warmup project

dOvs/Compilers efterår 2017

Andreas Salling Heglingegård[1], Andreas Sørensen[2], and Christian Zhuang-Qing Nielsen[3]

[1]201405359, 201405359@post.au.dk
[2]201508158, au542705@uni.au.dk
[3]201504624, christian@czn.dk

September 13, 2017

Contents

1 How did we make the interpreter **2**

2 Problems experienced **2**

3 Additional statements tested **2**

1 How did we make the interpreter

We found that the dynamically changing environment was more useful than the static one so we followed the approach from the book. Here we made every result in *interpStm* a list where every element was a tuple of an *id* and an *int* (i.e. [(id1, int1), (id2, int2), ...].
For *interpExp* every result was a tuple witht he first element being an *int* and the second one being a list of tubles of *id* and *int* (i.e. (int, [(id1, inty1), (id2, inty2), ...]))

This approach meant that every time a new variable was created it was dynamically added to our table (the list). Meanwhile, existing variable receiving new values would become new elements in the table. This way we could just look at the current value of a variable by looking at the newest version (the leftmost one). This resulted in an increased memory use if we assigned the same variable multiple values throughout the program. We deemed this acceptable.

Finally we should say that our interpreter only interprets syntactically correct programs. Specifically we cannot interpret a program containing the id-expression of an unassigned id.

val progfail = G.printStm(G.idExp "a") returns an error (because a value has not been assigned to "a").

2 Problems experienced

When we created *printEnv*, we had trouble figuring out how to make it work with another argument. To solve the problem, we added an extra argument to the function.

We spent a lot of time on figuring out haw to solve $interpStm(G.PrintStm(explist : G.explist)$, we ended up creating a function which returns a int list with the numbers it should print. When we create the int list we should append in the end of the list, we did not know how to do it, therefore we have to reverse the list before we print it.

3 Additional statements tested

Here we look at 5 statements. Above each statement we see them in the straight-line language. All outputs are from interp.

Code 1: $\text{print}(1, 2, (\text{print}(3, 4), 5), 6, 7)$

```
1 val prog1 =
2   G.PrintStm [G.NumExp 1,G.NumExp 2,G.EseqExp (G.PrintStm
3       [G.NumExp 3, G.NumExp 4],G.NumExp 5), G.NumExp 6,
            G.NumExp 7]
```

In Code 1 we test if we have call-by-value or by reference. This we can see because if we were using reference, then the `G.EseqExp` would be run when it was needed, but as we can see in the output

```
3 4
1 2 5 6 7
```

this expression is run first, and the print is run before we print the surrounding statement.

Code 2: $a := 1337; \text{print}(a, (a := 30, a))$

```
1 val prog2 =
2   G.CompoundStm (
3     G.AssignStm ("a", G.NumExp 1337)
4     ,G.PrintStm[G.IdExp "a", G.EseqExp (G.AssignStm ("a",
          G.NumExp 30), G.IdExp "a")])
```

In Code 2 we test whether we actually run through the expressions in the print from left to right, as we actually switched the direction after testing Code 1. This ended up as an error in this test, and we ended up reversing the output for printing instead. The real output is

```
1337 30
a = 30
```

Code 3: $a := 1337; b := 42; a := (a/b); \text{print}(a, b, (c := (a * 2), c))$

```
1 val prog3 =
2   G.CompoundStm (
3     G.CompoundStm (
4       G.CompoundStm (
5         G.AssignStm ("a", G.NumExp 1337)
6         , G.AssignStm ("b", G.NumExp 42))
7       , G.AssignStm ("a", G.OpExp (G.IdExp "a", G.Div, G.IdExp
            "b")))
8     , G.PrintStm [G.IdExp "a", G.IdExp "b", G.EseqExp
          (G.AssignStm ("c", G.OpExp (G.IdExp "a", G.Times,
          G.NumExp 2)), G.IdExp "c")])
```

The output of Code 3 is

```
31 42 62
c = 62
a = 31
b = 42
```

Code 4: $a := (30 + (1337 - 42))$

```
1 val prog4 =
2   G.AssignStm ("a", G.OpExp (G.NumExp 30, G.Plus,
3     G.OpExp (G.NumExp 1337, G.Minus, G.NumExp 42)))
```

The output of Code 4 is

```
a = 1325
```

Code 5: $a := 420; b := 8; c := 17; a := (a/b); a := (a + c); \text{print}(a)$

```
1 val prog5 =
2   G.CompoundStm (
3     G.CompoundStm (
```

```
4       G.CompoundStm (
5         G.CompoundStm (
6           G.CompoundStm (
7       G.AssignStm ("a", G.NumExp 420)
8       , G.AssignStm ("b", G.NumExp 8))
9       , G.AssignStm ("c", G.NumExp 17))
10      , G.AssignStm ("a", G.OpExp(G.IdExp "a", G.Div, G.IdExp
            "b")))
11    , G.AssignStm ("a", G.OpExp(G.IdExp "a", G.Plus, G.IdExp
        "c")))
12     , G.PrintStm [G.IdExp "a"])
```

The output of Code 5 is

```
69
a = 69
c = 17
b = 8
```