dOvs - warmup project

Group 9
Miran Hasanagić - 20084902
Jakob Graugaard Laursen - 20093220
Steven Astrup Sørensen - 201206081

September 8, 2015

1 Introduction

This report describes the approach in order to develop a straight line interpreter. Additionally, it provides an overview of problems encountered and intersting experiments during development process. Finally, five test expressions are interpreted and shown.

In order to develop the straight line interpreter the group members met physically in order to both unterstand the problem, and furthermore outline and discuss the solutions.

2 How did we make the interpreter

You should not write how you wrote the interpreter, but why you wrote it that way

Basically the interpreter is developed as two mutual functions; one in order to interpreter a statement, and one for interpreting an expression. This was a natural choice as a consequence of the two **datatype**s `stm` and `exp` that had to be interpreted. Generally the suggestions in the book were followed. Since a `stm` can be viewed as having a TREE structure as described in the book, it was naturally to use mutual recursive functions in order to interpret an expression, such as `prog` from the book.

+

3 Problems experienced

If you encountered any problems describe how you solved them

During the development different parts raised different challenges. In the beginning time was spent in order to learn the needed parts of SML. In the start phase of writing the interpreter some of the errors from the compiler did not make sense, so we used print for debugging. When the first parts of the code where written, the SML interpreter complained and threw error messages that filled several terminal screens. Defeated, after about an hour or two of trying to make sense of the errors along with rewriting the code, we decided to throw away the code and start from scratch.

We then build the functions one line at a time, each time checking if the functions as currently written had the correct type. We learned a very useful technique, namely having a catch all base case, something like funct (_) = NONE, until we had covered all possible cases we were interested in. This approach, which essentially is Test Driven Development, it helped us to to write all the desired functionality.

After the main functionality of the interpreter was woking correctly, different experiments led to the improvement of different special cases. For example, the two exceptions `DivisionByZero` and `IdExpNotFound`. Both can be interpreted with valid syntax, but can not be interpreted to valid values. Hence they had to be handled. The syntax is ensured by the **datatype**s `stm` and `exp`. Hence if a `stm` uses non valid constructs the SML interpreter will give an error.

4 Additional statements tested

5 statements as source code, and as values of type 'stm' plus a description of test including the outcome.

This section presents five test expressions for the interpreter. It shows source code, the actual type 'stm' and the outcome. All test have been successful, so the outcome matches the expected outcome.

4.1 Test 1

```
source code: a:=2; b:= a+2; print(b+a,b-a,b*a,b/a)

val test1 = G.CompoundStm(
        G.AssignStm("a",G.NumExp 2),
        G.CompoundStm(
            G.AssignStm("b", G.OpExp(G.IdExp "a", G.Plus, G.NumExp 2)),
            G.PrintStm[G.OpExp(G.IdExp "b", G.Plus, G.IdExp "a"),
                                G.OpExp(G.IdExp "b", G.Minus, G.IdExp "a"),
                                G.OpExp(G.IdExp "b", G.Times, G.IdExp "a"),
                                G.OpExp(G.IdExp "b", G.Div, G.IdExp "a")]))

Description: Test the assign statment, and that the id's are stored correctly.
                It also test the 4 different operations,
                and the print of the expression list at the same line.

Outcome: 6 2 8 2
```

4.2 Test 2

```
source code: print((3-5)/2); print(a);

val test2 = G.CompoundStm(
        G.PrintStm[G.OpExp(G.OpExp(G.NumExp, G.Minus, G.NumExp),G.Div, G.NumExp)],
        G.PrintStm[IdExp "a"])

Description: Test order of the operations. Test that the ID is not
                found error. Also execution order is tested, since the first
                print should be printed.

Outcome: -1 (--newline--) Error: Using unassigned variable: a
```

4.3 Test 3

```
source code: a:=3; print((print(a+3), a*4)); print(4/2);

val test3 =
    G.CompoundStm(
        G.AssignStm("a", G.NumExp 3),
        G.CompoundStm(
            G.PrintStm[G.OpExp(G.IdExp "a", G.Plus, G.NumExp 1),
                        G.EseqExp(G.PrintStm[G.OpExp(G.IdExp "a",G.Plus, G.NumExp 3)],
                                    G.OpExp(G.IdExp "a", G.Times, G.NumExp 4))],
            G.PrintStm[G.OpExp(G.NumExp 4, G.Div, G.NumExp 2)]))

Description: Tests the EseqExp. Furthermore, it tests the nested print.
        It also tests the execution order of the nested print. The interpreter
        evaluates all values of a print statement, before we print the values,
        so print-by-value version.

Outcome: 6 (--newline--) 4 12 (--newline--) 2
```

4.4 Test 4

```
source code: Test4: a:=0; print(a); Print(2/a); Print(2+a);

val test4 = G.CompoundStm(
        G.AssignStm("a", G.NumExp 0),
```

```
        G.CompoundStm(
            G.PrintStm[G.IdExp "a"],
            G.CompoundStm(
                G.PrintStm[G.OpExp(G.NumExp 2, G.Div, G.IdExp "a")],
                G.PrintStm[G.OpExp(G.NumExp 2, G.Minus, G.NumExp 1)])))

Description: Test for division by zero error. Also execution order is
             tested, since we do not expect to see the last print output.

Outcome: 0 (--newline--) Error: Division by zero not allowed
```

4.5 Test 5

```
source code: a:=2; print(a) ; a:= 4; Print(a);

val test5 = G.CompoundStm(
        G.AssignStm("a", G.NumExp 2),
        G.CompoundStm(
            G.PrintStm[G.IdExp "a"],
            G.CompoundStm(
                G.AssignStm("a", G.NumExp 4),
                G.PrintStm[G.IdExp "a"])))

Description: Tests the reassignment of the same value. Hence it tests
             the update and lookup of id's functionality.

Outcome: 2 (--newline--) 4
```

5 Conclusion

This report described the development process of the straight line interpreter. This project was used as a “warum up” project for both learning programming in SML and general principles of interpreting a language. Both aspects provide valuable lessons in order to start work on the compiler project.