Aflevering 2

Thomas Vinther, 201303874 Jens Kristian Nielsen, 201303862

12. februar 2019

Opgave 27

Kode

```
  def unparse(e: AstNode): String = e match{
    // ...
    case VarExp(x) => s"$x"
    case BlockExp(vals,exp) =>
      var valString = ""
      var endTuborg = ""
      for(d <- vals){
      valString = valString + "{ val "+d.x+" = "+unparse(d.exp)+" ; "
      endTuborg = endTuborg+" }"
      }
      valString+unparse(exp)+endTuborg
    }
```

```
  def eval(e: Exp, venv: VarEnv): Int = e match {
    // ...
    case VarExp(x) =>
      trace(s"Variable $x found, lookup of variable value in environment gave "
          +venv(x))
      venv(x)
    case BlockExp(vals, exp) =>
      var venv1 = venv
      for (d <- vals)
        venv1 = venv1 + (d.x -> eval(d.exp, venv1))
        trace("Calculating variable values and adding to variable environment")
      eval(exp, venv1)
  }
```

Beskrivelse

Implementationen af `BlockExp` og `ValDecl` i fortolkeren relaterer til den operationelle semantik ved:
VarExp(x): en ast node af denne type laver et lookup i variable environmentet venv fra eval kaldet for at finde den værdi der er knyttet til x, dette svarer i semantikken til:

$$\frac{\text{venv}(x) = v}{\text{venv} \vdash x \Rightarrow v}$$

BlockExp(vals,exp): en ast node af denne type løber listen vals af deklarationer igennem og for hver deklaration d udfører den følgende:

$$\frac{\text{venv1} \vdash \text{d.exp} \Rightarrow v \qquad \text{venv1'} = \text{venv1}[\text{d.x} \mapsto v]}{\text{venv1} \vdash \text{val d.x} = \text{d.exp} \Rightarrow \text{venv1'}}$$

Hvorefter vi sætter venv1 = venv1', mærket var for sammenlignelighed med slide 40 og 41. Når vi har udført dette for alle ValDecl's d i vals listen har vi kørt venstre side af tælleren i følgende udtryk der

repræsenterer hele BlockExp(vals,exp) udtrykket:

$$\frac{\text{venv} \vdash \text{vals} \Rightarrow \text{venv1} \qquad \text{venv1} \vdash \text{exp} \Rightarrow v}{\text{venv} \vdash \{\text{vals} \ ; \ \text{exp}\} \Rightarrow v}$$

Når fortolkeren køres med argumenterne `-unparse -run -trace examples/ex21.s` fås outputtet:
((x+ val z = (y/x) ; (z*2))+12)
Please provide an integer value for the variable x: 2
Please provide an integer value for the variable y: 3
BinOpExp found, evaluating left and right expressions
BinOpExp found, evaluating left and right expressions
Variable x found, lookup of variable value in environment: 2
BinOpExp found, evaluating left and right expressions
Variable y found, lookup of variable value in environment: 3
Variable x found, lookup of variable value in environment: 2
Dividing expressions
Calculating variable values and adding to variable environment
BinOpExp found, evaluating left and right expressions
Variable z found, lookup of variable value in environment: 1
Integer 2 found
Multiplying expressions
Adding expressions
Integer 12 found
Adding expressions
Output: 16

Fra vores main, parses programmet først, dernæst unparses det og printes i konsollen, øverst i outputtet, da vi har slået unparse til i vores kørsel. Da vi også har slået vores trace funktion til får vi også printet hvilket slags udtryk der evalueres i den givne rækkefølge fra fortolkeren.
Først skabes det initial variabel enviroment hvor der indtastes værdier til de frie variable, i metoden makeInitialVarEnv fra fortolkeren, der benytter metoden freeVars fra vars, der finder de frie variable.
Dernæst evalueres udtrykket. Først findes et BinOpExp, der består af exp:(x+ val z = (y/x) ; (z*2)), operatoren + og exp:12. Derefter evalueres det venstre exp, hvor der igen findes et BinOpExp, exp: x, operatoren + og exp: val z = (y/x) ; (z*2) . Herefter evalueres venstre siden først, dette var x som er en variabel der bliver slået op i det tilhørende enviroment.
Det højre udtryk af BinOpExp, mere bestemt: val z = (y/x) ; (z*2) bliver dernæst evalueret, her evalueres venstresiden: val z = (y/x), hvor variabel x og y bliver slået op i deres tilhørende enviroment, dernæst udregnes de og værdien af z bliver lagt ind i z's enviroment. Dernæst bliver (z*2) evalueret som et BinOpExp, z bliver slået op i enviroment, og 2 bliver fundet som et heltal. Udregningen udføres "opad"dvs. z*2 som bliver lagt sammen med x, som var blevet slået op til 2. Vi er nu næsten tilbage til det første BinOpExp hvor højresiden evalueres, her finder vi heltallet 12 og operatoren plus og udregningen udføres. Til sidst bliver det samlede resultat printet fra Main.

Opgave 28

Kode

```
1  import scala.collection.mutable.ListBuffer
2
3  def simplify(exp: Exp): Exp = {
4    var expNew = exp
5  while(expNew != simplify1(expNew)) {
6    expNew = simplify1(expNew)
7    }
8    expNew
9  }
10
11 def simplifyDecl(vd: ValDecl): ValDecl = vd match{
```

```
12      case ValDecl(x,exp) => ValDecl(x,simplify(exp))
13    }
14
15    def simplify1(exp: Exp): Exp =
16      exp match{
17        case IntLit(c)=> IntLit(c)
18        case VarExp(x)=> VarExp(x)
19        case UnOpExp(op,e)=> UnOpExp(op,simplify(e))
20        case BlockExp(vals,e)=>
21          var vals1 = new ListBuffer[ValDecl]()
22            for (v <- vals){
23              vals1 += simplifyDecl(v)
24            }
25            val vals2 = vals1.toList
26            BlockExp(vals2,simplify(e))
27        case BinOpExp(IntLit(m),ModuloBinOp(),IntLit(n)) =>
28          if((0<=m)&&(m<n)) IntLit(m)
29          else BinOpExp(IntLit(m),ModuloBinOp(),IntLit(n))
30        case BinOpExp(IntLit(m),MaxBinOp(),IntLit(n)) =>
31          if(m == n) IntLit(m)
32          else BinOpExp(IntLit(m),MaxBinOp(),IntLit(n))
33        case BinOpExp(IntLit(m),MultBinOp(),IntLit(n)) =>
34          if ((m < 0) && (n < 0)) BinOpExp(IntLit(-m), MultBinOp(), IntLit(-n))
35          else if (m < 0) UnOpExp(NegUnOp(), BinOpExp(IntLit(-m), MultBinOp(),
              IntLit(n)))
36          else if (n < 0) UnOpExp(NegUnOp(), BinOpExp(IntLit(m), MultBinOp(),
              IntLit(-n)))
37          else if (n == 1) IntLit(m)
38          else if (m == 1) IntLit(n)
39          else if ((n == 0) || (m == 0)) IntLit(0)
40          else BinOpExp(IntLit(m), MultBinOp(), IntLit(n))
41        case BinOpExp(le, op, re) => op match {
42          case PlusBinOp() =>
43            if(le == IntLit(0)) simplify(re)
44          else if(re == IntLit(0)) simplify(le)
45            else BinOpExp(simplify(le),op,simplify(re))
46          case MinusBinOp() =>
47            if(le == re) IntLit(0)
48          else if (le == IntLit(0)) UnOpExp(NegUnOp(),simplify(re))
49            else re match {
50              case IntLit(m) =>{
51                if (m<0) BinOpExp(simplify(le),PlusBinOp(),IntLit(-m))
52                else BinOpExp(simplify(le),op,simplify(re))
53            }
54            BinOpExp(simplify(le),op,simplify(re))
55            }
56        case MultBinOp() =>
57            if(le == IntLit(1)) simplify(re)
58            else if(re == IntLit(1)) simplify(le)
59            else if((le == IntLit(0))||(re == IntLit(0))) IntLit(0)
60          else BinOpExp(simplify(le),op,simplify(re))
61          case DivBinOp() =>
62            if(le == IntLit(0)) IntLit(0)
63            else if(re == IntLit(0)) throw new IllegalArgumentException("Division
                by zero")
64          else if(le == re) IntLit(1)
65            else BinOpExp(simplify(le),op,simplify(re))
66          case ModuloBinOp() =>
67            if(re == IntLit(0)) throw new IllegalArgumentException("Modulation by
                zero")
68            else BinOpExp(simplify(le),op,simplify(re))
69        case MaxBinOp() => BinOpExp(simplify(le),op,simplify(re))
70      }
```

```
  def main(args: Array[String]): Unit = {
    assert((Interpreter.simplify(Parser.parse("3%5")))==(Parser.parse("3")))
    assert(Interpreter.simplify(Parser.parse("3-3"))==Parser.parse("0"))
    assert(Interpreter.simplify(Parser.parse("a/a"))==Parser.parse("1"))
    assert(Interpreter.simplify(Parser.parse("10*0"))==Parser.parse("0"))
    assert(Interpreter.simplify(Parser.parse("0*10"))==Parser.parse("0"))
    assert(Interpreter.simplify(Parser.parse("55+0"))==Parser.parse("55"))
    assert(Interpreter.simplify(Parser.parse("0-12"))==Parser.parse("-12"))
    assert(Interpreter.simplify(Parser.parse("5*1"))==Parser.parse("5"))
    assert(Interpreter.simplify(Parser.parse("0/4"))==Parser.parse("0"))
    assert(Interpreter.simplify(Parser.parse("5max5"))==Parser.parse("5"))
    assert(Interpreter.simplify(Parser.parse("(3*3-9)max(0*9)"))==Parser.parse(
        "(((3*3)-9)max0)"))
    assert(Interpreter.simplify(Parser.parse("(5*(a/(--a)))*(5*(1-1))"))==
        Parser.parse("0"))
    assert(Interpreter.simplify(Parser.parse("{val x=3*1;x*0}"))==Parser.parse(
        "{ val x = 3 ; 0 }"))
    assert(Interpreter.simplify(Parser.parse("{val x={val z = 7/7 ; z*1};z*x*0}
        "))==Parser.parse("{ val x = { val z = 1 ; z } ; 0 }"))
    assert(Interpreter.simplify(Parser.parse("55+0"))==Parser.parse("55"))
    assert(Interpreter.simplify(Parser.parse("5*1"))==Parser.parse("5"))
    assert(Interpreter.simplify(Parser.parse("0/4"))==Parser.parse("0"))
    assert(Interpreter.simplify(Parser.parse("5max5"))==Parser.parse("5"))
    assert(Interpreter.simplify(Parser.parse("a/a"))==Parser.parse("1"))
    assert(Interpreter.simplify(Parser.parse("12-12"))==Parser.parse("0"))
    assert(Interpreter.simplify(Parser.parse("(5*1)*(5*(1-1))"))==Parser.parse(
        "0"))
    assert(Interpreter.simplify(Parser.parse("(5*1)*(5*0)"))==Parser.parse("0")
        )
  }
```