Aflevering 2

Studerende 1, 2017xxxxx   Studerende 2, 2017xxxxx

X. YYYY 20ZZ

Opgave 27

Kode

```scala
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
  }
```

```scala
def eval(e: Exp, venv: VarEnv): Int = e match {
  // ...
  case VarExp(x) =>
    trace("Variable found, lookup of variable value in environment")
    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

```
((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 found, lookup of variable value in environment
BinOpExp found, evaluating left and right expressions
Variable found, lookup of variable value in environment
Variable found, lookup of variable value in environment
Dividing expressions
Calculating variable values and adding to variable environment
BinOpExp found, evaluating left and right expressions
Variable found, lookup of variable value in environment
Integer 2 found
Multiplying expressions
Adding expressions
Integer 12 found
Adding expressions
Output: 16
```

Opgave 28

Kode

```scala
import scala.collection.mutable.ListBuffer

def simplify(exp: Exp): Exp = {
 var expNew = exp
while(expNew != simplify1(expNew)) {
 expNew = simplify1(expNew)
 }
 expNew
}

def simplifyDecl(vd: ValDecl): ValDecl = vd match{
 case ValDecl(x,exp) => ValDecl(x,simplify(exp))
}

def simplify1(exp: Exp): Exp =
 exp match{
  case IntLit(c)=> IntLit(c)
  case VarExp(x)=> VarExp(x)
 case UnOpExp(op,e)=> UnOpExp(op,simplify(e))
 case BlockExp(vals,e)=>
  var vals1 = new ListBuffer[ValDecl]()
   for (v <- vals){
    vals1 += simplifyDecl(v)
   }
   val vals2 = vals1.toList
   BlockExp(vals2,simplify(e))
  case BinOpExp(IntLit(m),ModuloBinOp(),IntLit(n)) =>
   if((0<=m)&&(m<n)) IntLit(m)
   else BinOpExp(IntLit(m),ModuloBinOp(),IntLit(n))
  case BinOpExp(IntLit(m),MaxBinOp(),IntLit(n)) =>
   if(m == n) IntLit(m)
   else BinOpExp(IntLit(m),MaxBinOp(),IntLit(n))
  case BinOpExp(IntLit(m),MultBinOp(),IntLit(n)) =>
   if ((m < 0) && (n < 0)) BinOpExp(IntLit(-m), MultBinOp(), IntLit(-n))
```

```
      else if (m < 0) UnOpExp(NegUnOp(), BinOpExp(IntLit(-m), MultBinOp(), IntLit(n)))
      else if (n < 0) UnOpExp(NegUnOp(), BinOpExp(IntLit(m), MultBinOp(), IntLit(-n)))
      else if (n == 1) IntLit(m)
      else if (m == 1) IntLit(n)
      else if ((n == 0) || (m == 0)) IntLit(0)
      else BinOpExp(IntLit(m), MultBinOp(), IntLit(n))
    case BinOpExp(le, op, re) => op match {
      case PlusBinOp() =>
        if(le == IntLit(0)) simplify(re)
        else if(re == IntLit(0)) simplify(le)
        else BinOpExp(simplify(le),op,simplify(re))
      case MinusBinOp() =>
        if(le == re) IntLit(0)
        else if (le == IntLit(0)) UnOpExp(NegUnOp(),simplify(re))
        else re match {
          case IntLit(m) =>{
            if (m<0) BinOpExp(simplify(le),PlusBinOp(),IntLit(-m))
            else BinOpExp(simplify(le),op,simplify(re))
          }
        BinOpExp(simplify(le),op,simplify(re))
        }
      case MultBinOp() =>
        if(le == IntLit(1)) simplify(re)
        else if(re == IntLit(1)) simplify(le)
        else if((le == IntLit(0))||(re == IntLit(0))) IntLit(0)
        else BinOpExp(simplify(le),op,simplify(re))
      case DivBinOp() =>
        if(le == IntLit(0)) IntLit(0)
        else if(re == IntLit(0)) throw new IllegalArgumentException("Division by zero")
        else if(le == re) IntLit(1)
        else BinOpExp(simplify(le),op,simplify(re))
      case ModuloBinOp() =>
        if(re == IntLit(0)) throw new IllegalArgumentException("Modulation by zero")
        else BinOpExp(simplify(le),op,simplify(re))
      case MaxBinOp() => BinOpExp(simplify(le),op,simplify(re))
}
```

```
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 }"))
}
```