Final Project: Data Flow Analysis

S. Patel, J. Collard, M. Barney

May 6, 2013

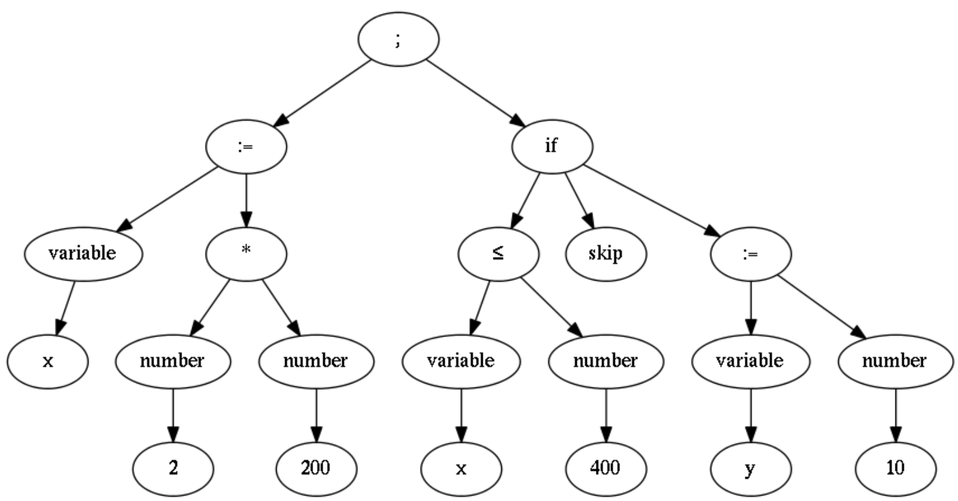
1 Introduction

This report contains our implementation of a scanner and parser for a basic programming language, and our data flow graph generation tools. Also included is a printer for the AST of a program that outputs `.gv` files to be used with a tool like dot to create graph-based images.

For example, the program:

```
x:= 2 * 200;
if x <= 400 then skip else y:=10 fi
```

yields the following AST:



Our implementation is divided up into several sections, roughly corresponding to the problems given in the specification, and each its own Haskell module.

We spent approximately 40 man hours on the project. This was slightly shorter than the previous assignments, but we feel that over the course of the semester, our ability to code together as a team has substantially improved.

In addition, since we were coding in Haskell, tasks like parsing, printing, and the even some of the algorithms have become familiar to us, and thus allowed us a faster development time.

2 Abstract Syntax

In this module we define the abstract syntax (AST) for statements written in a simple imperative language.

Its Backus-Naur form is as follows:

$$
\begin{aligned}
\text{Arithmetic expression } a &::= x \mid n \mid a_1 \ o_a \ a_2 \\
\text{Boolean expression } b &::= true \mid false \mid not \ b \mid b_1 \ o_b \ b_2 \mid a_1 \ o_r \ a_2 \\
\text{Statement } S &::= x \ := \ a \mid skip \mid S_1; S_2 \mid \\
& \quad if \ b \ then \ S_1 \ else \ S_2 \ fi \mid while \ b \ do \ S \ od
\end{aligned}
$$

The Haskell code will more or less precisely mirror the mathematical definition just given.

```
module AST where

import Data.Maybe

data AOP =
  Plus |
  Times |
  Minus deriving (Eq, Show, Enum)

data BOP =
  And |
  Or deriving (Eq, Show, Enum)

data REL =
  Equal |
  Less |
  Leq |
  Greater |
  Geq deriving (Eq, Show, Enum)

data Arith =
  Var String |
```

```
    Number Int |
    BinOp AOP Arith Arith deriving (Eq, Show)
data Boolean =
    T |
    F |
    Not Boolean |
    BoolOp BOP Boolean Boolean |
    RelOp REL Arith Arith deriving (Eq, Show)
data Statement =
    Assign String Arith |
    Skip |
    Seq Statement Statement |
    If Boolean Statement Statement |
    While Boolean Statement deriving (Eq, Show)
```

In addition, for bug testing, and for nice output, we wrote a pretty printer for ASTs, as follows:

```
pettyShowAOP :: AOP → String
pettyShowAOP aop = fromJust ∘ lookup aop $ ops where
    ops = zip [Plus .. Minus] ["+", "*", "-"]
pettyShowBOP :: BOP → String
pettyShowBOP And = "/\\"
pettyShowBOP Or = "\\/"
pettyShowREL :: REL → String
pettyShowREL rel = fromJust ∘ lookup rel $ rels where
    rels = zip [Equal .. Geq] ["==", "<", "<=", ">", ">="]
pettyShowArith :: Arith → String
pettyShowArith (Var s) = s
pettyShowArith (Number i) = show i
pettyShowArith (BinOp aop a1 a2) = pettyShowArith a1 ++ " "
    ++ pettyShowAOP aop ++ " "
    ++ pettyShowArith a2
pettyShowBool :: Boolean → String
pettyShowBool T = "true"
pettyShowBool F = "false"
pettyShowBool (Not b) = "not" ++ pettyShowBool b
pettyShowBool (BoolOp bop b1 b2) = pettyShowBool b1 ++ " "
    ++ pettyShowBOP bop ++ " "
```

$\quad +\!\!+\ \mathit{pettyShowBool}\ b2$

$\mathit{pettyShowBool}\ (\mathit{RelOp}\ \mathit{rel}\ a1\ \ a2) = \mathit{pettyShowArith}\ a1 +\!\!+ \texttt{" "}$

$\quad +\!\!+\ \mathit{pettyShowREL}\ \mathit{rel} +\!\!+ \texttt{" "}$

$\quad +\!\!+\ \mathit{pettyShowArith}\ a2$

$\mathit{pettyShowStatement} :: \mathit{Statement} \to \mathit{String}$

$\mathit{pettyShowStatement}\ (\mathit{Assign}\ s\ a) = s +\!\!+ \texttt{" := "} +\!\!+ \mathit{pettyShowArith}\ a$

$\mathit{pettyShowStatement}\ \mathit{Skip} = \texttt{"Skip"}$

$\mathit{pettyShowStatement}\ (\mathit{Seq}\ s1\ \ s2) = \mathit{pettyShowStatement}\ s1 +\!\!+ \texttt{";"} +\!\!+ [\texttt{'\textbackslash n'}]$

$\quad +\!\!+\ \mathit{pettyShowStatement}\ s2 +\!\!+ [\texttt{'\textbackslash n'}]$

$\mathit{pettyShowStatement}\ (\mathit{If}\ b\ s1\ \ s2) = \texttt{"if "} +\!\!+ \mathit{pettyShowBool}\ b +\!\!+ [\texttt{'\textbackslash n'}]$

$\quad +\!\!+ \texttt{"then "} +\!\!+ \mathit{pettyShowStatement}\ s1 +\!\!+ [\texttt{'\textbackslash n'}]$

$\quad +\!\!+ \texttt{"else "} +\!\!+ \mathit{pettyShowStatement}\ s2$

$\mathit{pettyShowStatement}\ (\mathit{While}\ b\ s) = \texttt{"while "} +\!\!+ \mathit{pettyShowBool}\ b +\!\!+ [\texttt{'\textbackslash n'}]$

$\quad +\!\!+\ \mathit{pettyShowStatement}\ s$

Lastly, as previously mentioned, we also wrote a printer that outputs dot syntax for drawing pretty graphs for the abstract syntax of a program. For example, the output for the example program in §1 is as follows:

```
digraph graphname{
s_0 [label=";"];
s_0 -> s_1;
s_0 -> s_9;
s_1 [label=":="];
s_2 [label="variable"];
s_3 [label="x"];
s_1 -> s_2 -> s_3;
s_1 -> a_4;
a_4 [label="*"];
a_4 -> a_5;
a_4 -> a_7;
a_5 [label="number"];
a_6 [label="2"];
a_5 -> a_6;
a_7 [label="number"];
a_8 [label="200"];
a_7 -> a_8;
s_9 [label="if"];
s_9 -> b_10;
s_9 -> s_15;
```

```
s_9 -> s_16;
b_10 [label="<="];
b_10 -> a_11;
b_10 -> a_13;
a_11 [label="variable"];
a_12 [label="x"];
a_11 -> a_12;
a_13 [label="number"];
a_14 [label="400"];
a_13 -> a_14;
s_15 [label="skip"];
s_16 [label=":="];
s_17 [label="variable"];
s_18 [label="y"];
s_16 -> s_17 -> s_18;
s_16 -> a_19;
a_19 [label="number"];
a_20 [label="10"];
a_19 -> a_20;
}
```

```haskell
dotPrinter' :: (Num t, Show t) => Statement -> t -> ([Char], t)
dotPrinter' (While b s1) counter =
  let whilelabel = "s_" ++ (show counter)
      (boolean, counter') = dotPrinterBool b (counter + 1)
      (statement1, counter'') = dotPrinter' s1 (counter')
      string = (whilelabel ++ " [label=\"while\"];\n" ++
                whilelabel ++ " -> " ++ "b_" ++ (show (counter + 1))
                  ++ ";\n" ++
                whilelabel ++ " -> " ++ "s_" ++ (show (counter'))
                  ++ ";\n" ++
                boolean ++ statement1) in
  (string, (counter''))
dotPrinter' (If b s1 s2) counter =
  let iflabel = "s_" ++ (show counter)
      (boolean, counter') = dotPrinterBool b (counter + 1)
      (statement1, counter'') = dotPrinter' s1 (counter')
      (statement2, counter''') = dotPrinter' s2 (counter'')
      string = (iflabel ++ " [label=\"if\"];\n" ++
```

```haskell
        iflabel ++ " -> " ++ "b_" ++ (show (counter + 1))
          ++ ";\n" ++
        iflabel ++ " -> " ++ "s_" ++ (show (counter'))
          ++ ";\n" ++
        iflabel ++ " -> " ++ "s_" ++ (show (counter''))
          ++ ";\n" ++
        boolean ++ statement1 ++ statement2) in
  (string, (counter'''))
dotPrinter' (Seq s1 s2) counter =
  let seq = "s_" ++ (show counter)
      (statement1, counter') = dotPrinter' s1 (counter + 1)
      (statement2, counter'') = dotPrinter' s2 (counter')
      string = (seq ++ " [label=\";\"];\n" ++
        seq ++ " -> " ++ "s_" ++ (show (counter + 1))
          ++ ";\n" ++
        seq ++ " -> " ++ "s_" ++ (show (counter'))
          ++ ";\n" ++
        statement1 ++ statement2) in
  (string, (counter''))
dotPrinter' (Skip) counter =
  let s1 = "s_" ++ (show counter)
      string = (s1 ++ " [label=\"skip\"];\n") in
  (string, (counter + 1))
dotPrinter' (Assign name a) counter =
  let s1 = "s_" ++ (show counter)
      s2 = "s_" ++ (show (counter + 1))
      s3 = "s_" ++ (show (counter + 2))
      (arith, counter') = dotPrinterArith a (counter + 3)
      string = (s1 ++ " [label=\":=\"];\n" ++
        s2 ++ " [label=\"variable\"];\n" ++
        s3 ++ " [label=\"" ++ name ++ "\"];\n" ++
        s1 ++ " -> " ++ s2 ++ " -> " ++ s3 ++ ";\n" ++
        s1 ++ " -> " ++ "a_" ++ (show (counter + 3))
          ++ ";\n" ++ arith) in
      (string, counter')

dotPrinterArith :: (Num t, Show t) => Arith -> t -> ([Char], t)
dotPrinterArith (Var s) counter =
  let v1 = "a_" ++ (show counter)
      v2 = "a_" ++ (show (counter + 1))
```

```haskell
    string = (v1 ++ " [label=\"variable\"];\n" ++
      v2 ++ " [label=\"" ++ s ++ "\"];\n" ++
      v1 ++ " -> " ++ v2 ++ ";\n"
                 ) in
  (string, (counter + 2))
dotPrinterArith (Number i) counter =
  let n1 = "a_" ++ (show counter)
    n2 = "a_" ++ (show (counter + 1))
    string = (n1 ++ " [label=\"number\"];\n" ++
      n2 ++ " [label=\"" ++ (show i) ++ "\"];\n" ++
      n1 ++ " -> " ++ n2 ++ ";\n"
                 ) in
  (string, (counter + 2))
dotPrinterArith (BinOp aop a1 a2) counter =
  let op = "a_" ++ (show counter)
    (s1, counter') = dotPrinterArith a1 (counter + 1)
    (s2, counter'') = dotPrinterArith a2 counter'
    string = (op ++ " [label=\"" ++ (dotAOP aop)
       ++ "\"];\n" ++
      op ++ " -> " ++ "a_" ++ (show (counter + 1))
         ++ ";\n" ++
      op ++ " -> " ++ "a_" ++ (show counter')
         ++ ";\n" ++
      s1 ++ s2
                 ) in
  (string, (counter''))
dotPrinterBool :: (Num t, Show t) => Boolean -> t -> ([Char], t)
dotPrinterBool (T) counter =
  let b1 = "b_" ++ (show counter)
    string = (b1 ++ " [label=\"\"];\n") in
  (string, (counter + 1))
dotPrinterBool (F) counter =
  let b1 = "b_" ++ (show counter)
    string = (b1 ++ " [label=\"\"];\n") in
  (string, (counter + 1))
dotPrinterBool (Not b) counter =
  let b1 = "b_" ++ (show counter)
    (s1, counter') = dotPrinterBool b (counter + 1)
    string = (b1 ++ " [label=\"not\"];\n" ++
```

```
    b1 ++ " -> " ++ (show (counter + 1)) ++ ";\n" ++
    s1
                 ) in
  (string, counter')
dotPrinterBool (BoolOp bop b1 b2) counter =
  let op = "b_" ++ (show counter)
    (s1, counter') = dotPrinterBool b1 (counter + 1)
    (s2, counter'') = dotPrinterBool b2 counter'
    string = (op ++ " [label=\"" ++ (dotBOP bop) ++ "\"];\n" ++
      op ++ " -> " ++ "b_" ++ (show (counter + 1)) ++ ";\n" ++
      op ++ " -> " ++ "b_" ++ (show counter') ++ ";\n" ++
      s1 ++ s2
                 ) in
  (string, (counter''))
dotPrinterBool (RelOp rel a1 a2) counter =
  let op = "b_" ++ (show counter)
    (s1, counter') = dotPrinterArith a1 (counter + 1)
    (s2, counter'') = dotPrinterArith a2 counter'
    string = (op ++ " [label=\"" ++ (dotREL rel) ++ "\"];\n" ++
      op ++ " -> " ++ "a_" ++ (show (counter + 1)) ++ ";\n" ++
      op ++ " -> " ++ "a_" ++ (show counter') ++ ";\n" ++
      s1 ++ s2
                 ) in
  (string, (counter''))
dotAOP :: AOP -> [Char]
dotAOP (Plus)  = "+"
dotAOP (Minus) = "-"
dotAOP (Times) = "*"
dotBOP (And)   = ""
dotBOP (Or)    = ""
dotREL (Equal)   = "=="
dotREL (Less)    = "<"
dotREL (Leq)     = ""
dotREL (Greater) = ">"
dotREL (Geq)     = ""
dotPrinter :: Statement -> [Char]
dotPrinter x =
  ("digraph graphname{\n" ++ (fst (dotPrinter' x 0)) ++ "}")
```

3 Scanner and Parser

We decided to use the Parsec library for parsing files containing the simple imperative language.

This gave us a great amount of flexibility for parsing input files. For example, our implementation supports programs containing the unicode characters for ,,,, and , representing their respective operations.

Furthermore, the error messaging system for Parsec, when scanning fails, is actually quite beautiful, and very detailed.

A second benefit is that the parsers for booleans, arithmetic expressions, and statements, again almost exactly mirrors the abstract syntax for those expressions, which makes it that much easier to debug and to read.

Our scanner and parser supports fully parenthesized arithmetic expressions *or* precedence rules for plus, times, and minus (times has highest precedence). As a result, we are capable of scanning and parsing arbitrarily complex expressions, but a design decision was made to keep our control flow graph and reaching definitions restricted to "simple" expressions with single variables.

Lastly, we strictly obey the BNF for statements when parsing. A consequence is that we are strict with respect to the use of semicolons for sequencing. In other words:

```
skip; skip
```

is a valid program accepted by our scanner and parser.

```
skip; skip;
```

however, is not.

```
module Input (sparse) where
import AST
import Text.ParserCombinators.Parsec
expr  = term `chainl1` addop
term  = factor `chainl1` mulop
varParser :: GenParser Char st Arith
varParser = do
  v <- many1 letter
  spaces
  return (Var v)
```

```
numParser :: GenParser Char st Arith
numParser = do
  n ← many1 digit
  spaces
  return (Number ((read n) :: Int))

factor =
  varParser <|> numParser <|>
    do
    char '('
    spaces
    n ← expr
    spaces
    char ')'
    spaces
    return n

addop = do {char '+'; spaces; return (BinOp Plus)} <|>
         do {char '-'; spaces; return (BinOp Minus)}

mulop = do {char '*'; spaces; return (BinOp Times)}

arithmetic = do
  e ← expr
  return e

optionalParens p = between (char '(') (char ')') p <|> p
```

The boolean parsers are as follows:

```
notParser = do
  string "not" <|> string "" <|> string "~"
  spaces
  b ← boolean
  return $ Not b

andParser = do
  string "/\\" <|> string ""
  spaces
  b2 ← boolean
  return $ (λx → BoolOp And x b2)

orParser = do
  string "\\/" <|> string ""
  spaces
  b2 ← boolean
```

```
  return $ (λx → BoolOp Or x b2)
relation =
  do {string ">=" <|> string ""; return $ RelOp Geq} <|>
  do {string "<=" <|> string ""; return $ RelOp Leq} <|>
  do {string ">"; return $ RelOp Greater} <|>
  do {string "<"; return $ RelOp Less} <|>
  do {string "=="; return $ RelOp Equal}
relopParser = do
  a1 ← arithmetic
  spaces
  relop ← relation
  spaces
  a2 ← arithmetic
  return $ relop a1 a2
tfParser =
  do {string "true"; spaces; return T} <|>
  do {string "false"; spaces; return F}
boolean = do
  b ← notParser <|> tfParser <|> relopParser
  bexpr ← optionMaybe $ andParser <|> orParser
  case bexpr of
    Nothing → return b
    Just bFun → return $ bFun b
```

And finally, the statement parsers, for constructing the AST for a given program.

```
assignment = do
  identifier ← many1 letter
  spaces
  string ":="
  spaces
  expression ← arithmetic
  return $ Assign identifier expression
skip = do
  string "skip"
  spaces
  return Skip
ifstatement = do
```

```
  string "if"
  many1 space
  b ← boolean
  string "then"
  many1 space
  s1 ← statement
  string "else"
  many1 space
  s2 ← statement
  string "fi"
  return $ If b s1 s2
whilestatement = do
  string "while"
  many1 space
  b ← boolean
  string "do"
  many1 space
  s ← statement
  string "od"
  return $ While b s
  -- assignment must be last to preserve keywords
statement' = skip <|> ifstatement <|> whilestatement <|> assignment
statement :: GenParser Char st Statement
statement = do
  s1 ← statement'
  seq ← optionMaybe (char ';' ≫ spaces ≫ statement)
  case seq of
    Nothing → return s1
    Just s2 → return $ Seq s1 s2
sparse = parse statement "(syntax error)"
```

4 Control Flow Diagrams

In this section we compute the control flow graph for a given AST.

```
{-# LANGUAGE TupleSections #-}
module ControlFlow where
import AST
```

```
import Control.Applicative
import Control.Monad.State
import qualified Data.Map as M
import qualified Data.Set as S

type Block = Either Statement Boolean

data ControlFlowGraph = CFG { labels :: M.Map Int Block,
  outEdges :: M.Map Int (S.Set Int),
  inEdges :: M.Map Int (S.Set Int) } deriving (Show, Eq)

decorate :: Statement → M.Map Int Block
decorate = M.fromList ∘ flip evalState 0 ∘ decorate′

decorate′ :: Statement → State Int [(Int, Block)]
decorate′ a@(Assign s arith) = (:[ ]) < $ > (, Left a) < $ > getIncrement

decorate′ Skip = (:[ ]) < $ > (, Left Skip) < $ > getIncrement

decorate′ (Seq s1 s2) = decorate2 s1 s2

decorate′ con@(If bool s1 s2) = (:) < $ > (, Right bool)
  < $ > getIncrement
  < ∗ > decorate2 s1 s2

decorate′ whl@(While bool s) = (:) < $ > (, Right bool) < $ > getIncrement < ∗ > decorate′ s

decorate2 :: Statement → Statement → State Int [(Int, Block)]
decorate2 s1 s2 = (++) < $ > (decorate′ s1) < ∗ > (decorate′ s2)

getIncrement :: Num s ⇒ State s s
getIncrement = get ≫= λi → put (i + 1) ≫ return i

getIncrement2 :: Num s ⇒ State (s, a) (s, a)
getIncrement2 = get ≫= λ(i, s) → put ((i + 1), s) ≫ return (i, s)

displayLabeledGraph :: M.Map Int Block → IO ()
displayLabeledGraph = mapM_ (putStrLn ∘ showBlock) ∘ M.toList where
  showBlock (i, Left s) = "[" ++ (pettyShowStatement s) ++ "]" ++ (show i)
  showBlock (i, Right b) = "[" ++ (pettyShowBool b) ++ "]" ++ (show i)

ast :: Statement
ast = Seq (Assign "x" (BinOp Plus (Number 5) (Number 3))) s where
  s = Seq (Assign "y" (Number 3)) s2
  s2 = Seq (While (RelOp Less (Var "y") (Var "x")) s4) s3
  s3 = Skip
  s4 = Assign "y" (BinOp Plus (Var "y") (Number 1))

ast2 :: Statement
ast2 = Seq (Assign "x" (BinOp Plus (Number 5) (Number 3))) s where
```

```
  s = Seq (Assign "y" (Number 3)) s2
  s2 = Seq (While (RelOp Less (Var "y") (Var "x")) s4) s3
  s3 = Skip
  s4 = If T (Assign "y" (BinOp Plus (Var "y") (Number 1))) s5
  s5 = Assign "y" (BinOp Plus (Var "y") (Number 2))
cfg :: ControlFlowGraph
cfg = CFG (decorate ast) out ins where
  ins = M.fromList [(0, S.empty), (1, set 0), (2, flist [1, 3]),
    (3, set 2), (4, set 2)]
  out = M.fromList [(0, set 1), (1, set 2), (2, flist [3, 4]),
    (3, set 2), (4, S.empty)]
  set = S.singleton
  flist = S.fromList
cfg2 :: ControlFlowGraph
cfg2 = CFG (decorate ast2) out ins where
  ins = M.fromList [(0, S.empty), (1, set 0), (2, flist [1, 4, 5]),
    (3, set 2), (4, set 3), (5, set 3), (6, set 2)]
  out = M.fromList [(0, set 1), (1, set 2), (2, flist [3, 6]),
    (3, flist [4, 5]), (4, set 2), (5, set 2), (6, S.empty)]
  set = S.singleton
  flist = S.fromList
controlFlowGraph :: Statement → ControlFlowGraph
controlFlowGraph g = init where
  init = CFG dec outs ins
  dec = decorate g
  outs = snd $ computeSuccessors 0 1 dec g
  ins = computePredecessors outs
computeSuccessors :: Int
  → Int
  → M.Map Int Block
  → Statement
  → (Int, M.Map Int (S.Set Int))
computeSuccessors i n dec (Seq s1 s2) = (n2, M.union m1 m2) where
  (n1, m1) = computeSuccessors i n dec s1
  (n2, m2) = computeSuccessors (n1 + 1) (n1 + 2) dec s2
computeSuccessors i n dec (If _ s1 s2) = (n2, M.unions [m, m1, m2]) where
  m = M.singleton i (S.fromList [i + 1, i + 2])
  (n1, m1) = computeSuccessors (i + 1) n dec s1
```

```
    (n2, m2) = computeSuccessors (i + 2) n dec s2
computeSuccessors i n dec (While _ s) = (n1, M.union m m1) where
    m = M.singleton i set
    set = case M.lookup n1 dec of
        Nothing → S.singleton n
        _ → S.fromList [i + 1, n1 + 1]
    (n1, m1) = computeSuccessors n i dec s
computeSuccessors i n dec _ = (i, M.singleton i set) where
    set = case M.lookup n dec of
        Nothing → S.empty
        _ → S.singleton n
computePredecessors :: M.Map Int (S.Set Int)
    → M.Map Int (S.Set Int)
computePredecessors outs = M.fromList ∘ map go ∘ M.keys $ outs where
    go i = (i, labelsWith i)
    labelsWith i = S.fromList [j | (j, set) ← M.toList outs, S.member i set]
```

5 Reaching Definitions

In this section we compute the reaching definitions for a given AST and its control flow graph.

```
{-# LANGUAGE ViewPatterns #-}
module ReachingDefinition (formatEquations,
    ReachingDefinitions,
    reachingDefinitions,
    formatReachingDefinitions) where
import AST
import ControlFlow
import Data.List (intercalate)
import qualified Data.Map as M
import qualified Data.Set as S
type ReachingDefinition = S.Set (String, Maybe Int)
data ReachingDefinitions = RDS {entry :: M.Map Int ReachingDefinition,
    exit :: M.Map Int ReachingDefinition}
type EntryDefs = M.Map Int ReachingDefinition
type ExitDefs = M.Map Int ReachingDefinition
```

```
type KillSet = ReachingDefinition
type GenSet = ReachingDefinition
type ExitEquation = (Int, KillSet, GenSet)

type EntryEquation = (Int, S.Set Int)

reachingDefinitions :: ControlFlowGraph → ReachingDefinitions
reachingDefinitions cfg = RDS entries exits where
  (entries, exits) = reachingDefinitions′ (empties, empties) cfg
  empties = M.unions ∘ map ((flip M.singleton) S.empty) $ lbls
  lbls = M.keys ∘ labels $ cfg

reachingDefinitions′ :: (EntryDefs, ExitDefs) → ControlFlowGraph →
    (EntryDefs, ExitDefs)
reachingDefinitions′ (entries, exits) cfg =
  if entries ≡ entries′ ∧ exits ≡ exits′
    then (entries′, exits′)
    else reachingDefinitions′ (entries′, exits′) cfg where
  (entries′, exits′) = pass cfg (entries, exits)

pass :: ControlFlowGraph → (EntryDefs, ExitDefs) →
  (EntryDefs, ExitDefs)
pass cfg (entries, exits) =
  pass′ 0 vars lbls (S.empty) cfg (entries, exits) where
  lbls = S.fromList ∘ M.keys ∘ labels $ cfg
  vars = determineVars cfg

pass′ :: Int → S.Set String → S.Set Int →
  S.Set Int → ControlFlowGraph →
  (EntryDefs, ExitDefs) → (EntryDefs, ExitDefs)
pass′ l vars lbls marked cfg (entries, exits) =
  if S.null nextLabels then (entries′, exits′)
    else (entries″, exits″) where
  (_, kill, gen) = getExitEquation lbls l (labels cfgM.! l)
  (_, entEq) = entryEquation l cfg
  exitSets = map (exitsM.!) (S.toList entEq)
  nextEntry = if l ≡ 0 then initialEntry vars else S.unions exitSets
  nextExit = nextEntry `S.difference` kill `S.union` gen
  nextLabels = (outEdges cfgM.! l) `S.difference` marked
  entries′ = M.insert l nextEntry entries
  exits′ = M.insert l nextExit exits
  recurse n = pass′ n vars lbls (S.insert l marked)
    cfg (entries′, exits′)
```

```
  branches = S.toList ∘ S.map recurse $ nextLabels
  entries'' = mergeSets ∘ map fst $ branches
  exits'' = mergeSets ∘ map snd $ branches
mergeSets :: [M.Map Int ReachingDefinition]
    → M.Map Int ReachingDefinition
mergeSets maps = sets where
  set i = S.unions ∘ map (M.! i) $ maps
  lbls = head ∘ map M.keys $ maps
  sets = M.unions ∘ zipWith (M.singleton) lbls ∘ map set $ lbls
initialEntry :: S.Set String → ReachingDefinition
initialEntry = S.map (λstr → (str, Nothing))
formatReachingDefinitions :: ReachingDefinitions → String
formatReachingDefinitions (RDS entries exits) =
  (formatEntryDefs entries) ++ "\n" ++ (formatExitDefs exits)
formatEntryDefs :: EntryDefs → String
formatEntryDefs entries = intercalate "\n" defs where
  keys = M.keys entries
  defs = zipWith formatEntryDef keys (map (entriesM.!) keys)
formatEntryDef :: Int → ReachingDefinition → String
formatEntryDef l def = "RD(" ++ (show l) ++ ") = " ++
    (formatReachingDef def)
formatReachingDef :: ReachingDefinition → String
formatReachingDef (S.toList → defs) =
  "{" ++ (intercalate "," ∘ map formatElement $ defs) ++ "}"
formatExitDefs :: ExitDefs → String
formatExitDefs exits = intercalate "\n" defs where
  keys = M.keys exits
  defs = zipWith formatExitDef keys (map (exitsM.!) keys)
formatExitDef :: Int → ReachingDefinition → String
formatExitDef l def = "RD(" ++ (show l) ++ ") = " ++
  (formatReachingDef def)
formatEquations :: ControlFlowGraph → String
formatEquations cfg = entries ++ "\n" ++ exits where
  entries = intercalate "\n" ∘ map (formatEntryE vars) ∘
    entryEquations $ cfg
  exits = intercalate "\n" ∘ map formatExitE ∘ exitEquations $ cfg
  vars = determineVars cfg
```

```haskell
entryEquations :: ControlFlowGraph → [EntryEquation]
entryEquations cfg = zip lbls sets where
  x = inEdges cfg
  lbls = M.keys ∘ labels $ cfg
  sets = map (xM.!) lbls

entryEquation :: Int → ControlFlowGraph → EntryEquation
entryEquation l cfg = (l, (inEdges cfg)M.! l)

formatEntryE :: S.Set String → EntryEquation → String
formatEntryE (S.toList → vars) (l, es)
  | l ≡ 0 = "RD(0) = {" ++ intercalate ", "
    (map formatVar vars) ++ "}   " ++ (formatEntries es)
  | otherwise = "RD(" ++ (show l) ++ ") = " ++ (formatEntries es)

formatEntries :: S.Set Int → String
formatEntries (S.toList → es)
  | null es = "{}"
  | otherwise = intercalate "   " ∘ map format $ es
    where
      format i = "RD(" ++ (show i) ++ ")"

formatVar :: String → String
formatVar s = "(" ++ s ++ ", ?)"

formatExitE :: ExitEquation → String
formatExitE (l, kill, gen) = "RD(" ++ (show l) ++ ") = " ++
  "RD(" ++ (show l) ++ ") " ++
  " {" ++ (formatDef kill) ++ "} " ++
  " {" ++ (formatDef gen) ++ "}"

formatDef :: ReachingDefinition → String
formatDef (S.toList → elems) = intercalate ", " ∘
  map formatElement $ elems

formatElement :: (String, Maybe Int) → String
formatElement (str, Nothing) = "(" ++ str ++ ", ?)"
formatElement (str, Just x) = "(" ++ str ++ ", " ++ (show x) ++ ")"

exitEquations :: ControlFlowGraph → [ExitEquation]
exitEquations cfg = [getExitEquation set i (mapM.! i) | i ← lbls]
  where
    map = labels cfg
    set = S.fromList lbls
    lbls = M.keys map
```

```
getExitEquation :: S.Set Int → Int → Block → ExitEquation
getExitEquation labels l block = (l, killSet labels block,
        genSet l block)

killSet :: S.Set Int → Block → KillSet
killSet labels (Left (Assign var _)) = S.union
(S.singleton (var, Nothing)) ∘ S.fromList ∘
zipWith (λs i → (s, Just i)) (repeat var) ∘ S.toList $ labels
killSet _ _ = S.empty

genSet :: Int → Block → GenSet
genSet l (Left (Assign var _)) = S.singleton (var, Just l)
genSet _ _ = S.empty

determineVars :: ControlFlowGraph → S.Set String
determineVars (labels → M.elems → cfg) = S.unions ∘ map getVars $ cfg

getVars :: Block → S.Set String
getVars (Left (Assign label arith)) = S.singleton label `S.union`
  (getArithVars arith)
getVars (Right bool) = getBoolVars bool
getVars _ = S.empty

getBoolVars :: Boolean → S.Set String
getBoolVars (BoolOp _ b0 b1) = S.union (getBoolVars b0)
  (getBoolVars b1)
getBoolVars (RelOp _ a0 a1) = S.union (getArithVars a0)
  (getArithVars a1)
getBoolVars _ = S.empty

getArithVars :: Arith → S.Set String
getArithVars (Var label) = S.singleton label
getArithVars (BinOp _ a0 a1) = S.union (getArithVars a0)
  (getArithVars a1)
getArithVars _ = S.empty

simpleGraph :: ControlFlowGraph
simpleGraph = CFG labels outEdges inEdges where
  labels = M.fromList [(0, Left (Assign "x" (Number 0))),
    (1, Left (Assign "y" (Number 0))),
    (2, Right (RelOp Less (Var "x")
      (BinOp Plus (Var "a") (Var "b")))),
    (3, (Left (Assign "x"
      (BinOp Plus (Var "x") (Var "a"))))),
    (4, (Left (Assign "a"
```

```
      (BinOp Minus (Var "a") (Number 1))))),
    (5, (Left (Assign "b"
      (BinOp Plus (Var "b") (Var "x")))))]
  outEdges = M.fromList [(0, S.singleton 1),
    (1, S.singleton 2),
    (2, S.fromList [3, 5]),
    (3, S.singleton 4),
    (4, S.singleton 2),
    (5, S.empty)]
  inEdges = M.fromList [(0, S.empty),
    (1, S.singleton 0),
    (2, S.fromList [1, 4]),
    (3, S.singleton 2),
    (4, S.singleton 3),
    (5, S.singleton 2)]
```

6 Main module

The main module puts everything together.

```
module Main where

import System.Environment
import AST
import Input
import ControlFlow
import ReachingDefinition

main = do
  [file] ← getArgs
  contents ← readFile file
  case sparse contents of
    Right ast → putStrLn ∘ dotPrinter $ ast
    Left err → print err
```

7 Example: while.txt Program

Given the following simple program:

```
y := x;
```

```
z := 1;
while y > 0 do
      z := z * y;
      y := y - 1
od;
y := 0
```

After scanning and parsing, our dot printer gives its abstract syntax as: