Final Project: Data Flow Analysis

S. Patel, J. Collard, M. Barney

May 10, 2013

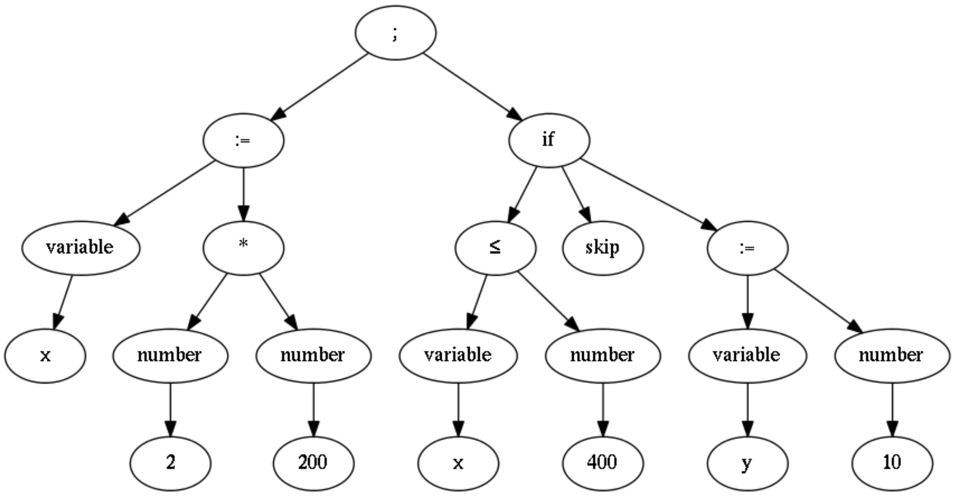
1 Introduction

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

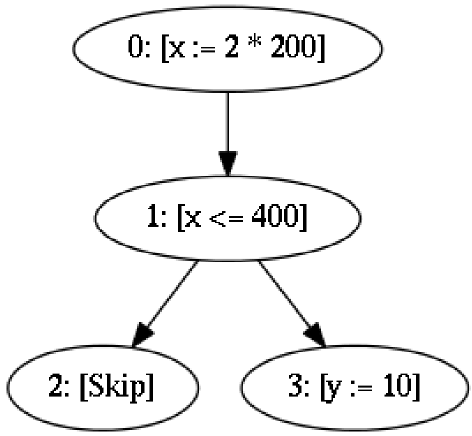
For example, the program:

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

yields the following AST:



whereas its CFG is:



Our implementation is divided up into several sections, roughly corresponding to the problems given in the specification, and each its own Haskell module. Each group member was more or less given a module to work on, but there was coding across modules happening on occasion.

We spent approximately 50 man hours on the project. We feel that over the course of the semester, our ability to code together as a team substantially improved, and we were able to work together in an efficient and robust manner.

Unfortunately, due to time constraints, we were unable to complete the optional assignments, but we are very happy with our final implementation, and this report.

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**

```haskell
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:

```haskell
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 ++ " "
  ++ pettyShowBool b2
pettyShowBool (RelOp rel a1 a2) = pettyShowArith a1 ++ " "
  ++ pettyShowREL rel ++ " "
  ++ pettyShowArith a2

pettyShowStatement :: Statement -> String
pettyShowStatement (Assign s a) = s ++ " := " ++ pettyShowArith a
pettyShowStatement Skip = "Skip"
pettyShowStatement (Seq s1 s2) = pettyShowStatement s1 ++ ";" ++ ['\n']
  ++ pettyShowStatement s2 ++ ['\n']
pettyShowStatement (If b s1 s2) = "if " ++ pettyShowBool b ++ ['\n']
  ++ "then " ++ pettyShowStatement s1 ++ ['\n']
  ++ "else " ++ pettyShowStatement s2
pettyShowStatement (While b s) = "while " ++ pettyShowBool b ++ ['\n']
  ++ 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;
}
```

$$\begin{aligned}
&\mathit{dotPrinter'} :: (\mathit{Num}\ t, \mathit{Show}\ t) \Rightarrow \mathit{Statement} \rightarrow t \rightarrow ([\mathit{Char}], t)\\
&\mathit{dotPrinter'}\ (\mathit{While}\ b\ s1)\ \mathit{counter} =
\end{aligned}$$

```haskell
  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" ++
        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))
      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 → String
dotPrinter x =
  ("digraph graphname{\n" ⧺ (fst (dotPrinter′ x 0)) ⧺ "}")

dotPrint = putStrLn ∘ dotPrinter
```

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.

```haskell
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 {try (string ">=" <|> string " "); return $ RelOp Geq} <|>
  do {try (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, ViewPatterns #-}
module ControlFlow where
import AST
import Control.Applicative
import Control.Monad.State
import Data.List (intercalate)
import qualified Data.Map as M
import qualified Data.Set as S
import Input
import Test.HUnit hiding (State)
```

A block is any part of a program that must be labeled. For this language, it's either a boolean expression, an assignment, or a skip.

```
type Block = Either Statement Boolean
```

A Control Flow Graph is a collection of labeled blocks and edges.

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

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

formatCFGasDot takes a *ControlFlowGraph* and formats it as a DOT graph string. For example:

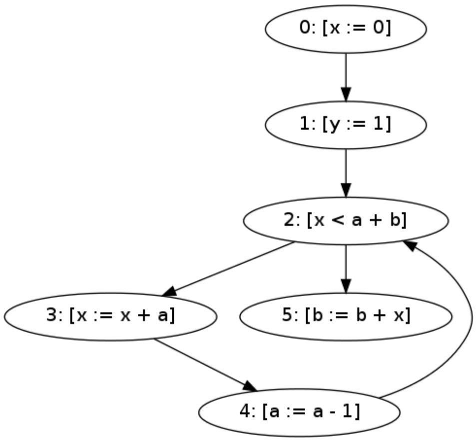
'formatCFGasDOT simpleGraph' returns:

```
digraph {
l_0 [label="0: [x := 0]"]
l_1 [label="1: [y := 1]"]
l_2 [label="2: [x < a + b]"]
l_3 [label="3: [x := x + a]"]
l_4 [label="4: [a := a - 1]"]
l_5 [label="5: [b := b + x]"]
l_0 -> l_1
l_1 -> l_2
l_2 -> l_3
l_2 -> l_5
l_3 -> l_4
l_4 -> l_2
}
```

Once compiled:

```
dot -Tpng "simpleGraph.dot" > "simpleGraph.png"
```

Produces the image:




```haskell
formatCFGasDOT :: ControlFlowGraph -> String
formatCFGasDOT cfg = "digraph {\n" ++ nodes ++ "\n" ++ edges ++ "\n}"
  where
    nodes = formatLabels (labels cfg)
    edges = intercalate "\n" . zipWith formatEdges outks $
      (map (outE M.!) outks)
    outE = outEdges cfg
    outks = M.keys outE

formatLabels :: M.Map Int Block -> String
formatLabels (M.assocs -> labels) = intercalate "\n" .
  map (uncurry formatLabel) $ labels

formatEdges :: Int -> S.Set Int -> String
formatEdges from toSet = intercalate "\n" . map (formatEdge from) $
  S.toList toSet

formatEdge :: Int -> Int -> String
formatEdge from to = "l_" ++ (show from) ++ " -> l_" ++ (show to)

formatLabel :: Int -> Block -> String
formatLabel i (Left state) = "l_" ++ (show i) ++
  " [label=\"" ++ (show i) ++ ": ["
   ++ (pettyShowStatement state) ++ "]\"]"
formatLabel i (Right bool) = "l_" ++ (show i) ++ " [label=\"" ++
  (show i) ++ ": [" ++ (pettyShowBool bool)
   ++ "]\"]"
```

The *decorate* function takes a program, and returns a map from a label int to a block.

$$\mathit{decorate} :: \mathit{Statement} \rightarrow \mathit{M.Map\ Int\ Block}$$
$$\mathit{decorate} = \mathit{M.fromList} \circ \mathit{flip\ evalState}\ 0 \circ \mathit{decorate'}$$

If a statement is an assignment, label it.

$$\mathit{decorate'} :: \mathit{Statement} \rightarrow \mathit{State\ Int}\ [(\mathit{Int}, \mathit{Block})]$$
$$\mathit{decorate'}\ \mathit{a@(Assign\ s\ arith)} = (:[\,]) < \$ > (, \mathit{Left\ a}) < \$ > \mathit{getIncrement}$$

If a statement is a *Skip*, label it.

$$\mathit{decorate'}\ \mathit{Skip} = (:[\,]) < \$ > (, \mathit{Left\ Skip}) < \$ > \mathit{getIncrement}$$

If a statement is a sequence of statements, then label each one.

$$\mathit{decorate'}\ (\mathit{Seq\ s1\ s2}) = \mathit{decorate2\ s1\ s2}$$

If a statement is an if then else, label the then and else statements, as well as the boolean expression.

$$\mathit{decorate'}\ \mathit{con@(If\ bool\ s1\ s2)} = (:) < \$ > (, \mathit{Right\ bool})$$
$$\quad < \$ > \mathit{getIncrement}$$
$$\quad < * > \mathit{decorate2\ s1\ s2}$$

If a statement is a while loop, label the boolean expression and the statement in the loop.

$$\mathit{decorate'}\ \mathit{whl@(While\ bool\ s)} = (:) < \$ > (, \mathit{Right\ bool}) < \$ >$$
$$\quad \mathit{getIncrement} < * > \mathit{decorate'}\ s$$

Decorate two statements and collect their results.

$$\mathit{decorate2} :: \mathit{Statement} \rightarrow \mathit{Statement} \rightarrow \mathit{State\ Int}\ [(\mathit{Int}, \mathit{Block})]$$
$$\mathit{decorate2\ s1\ s2} = (+\!\!+) < \$ > (\mathit{decorate'}\ \mathit{s1}) < * > (\mathit{decorate'}\ \mathit{s2})$$

Get the next label and then increment it for the next get call.

$$\mathit{getIncrement} :: \mathit{Num\ s} \Rightarrow \mathit{State\ s\ s}$$
$$\mathit{getIncrement} = \mathit{get} \ggg\!= \lambda i \rightarrow \mathit{put}\ (i + 1) \gg \mathit{return}\ i$$

Computes a control flow graph for a program.

$$
\begin{aligned}
&\mathit{controlFlowGraph} :: \mathit{Statement} \to \mathit{ControlFlowGraph}\\
&\mathit{controlFlowGraph}\ g = \mathit{CFG}\ \mathit{dec}\ \mathit{outs}\ \mathit{ins}\ \mathbf{where}\\
&\quad \mathit{dec} = \mathit{decorate}\ g\\
&\quad \mathit{outs} = \mathit{computeSuccessors}\ g\ (\mathit{M.keysSet}\ \mathit{dec})\\
&\quad \mathit{ins} = \mathit{computePredecessors}\ \mathit{outs}
\end{aligned}
$$

basicBlocks calculates the first level of basic blocks for a program and returns them in a list.

$$
\begin{aligned}
&\mathit{basicBlocks} :: \mathit{Statement} \to [\mathit{Statement}]\\
&\mathit{basicBlocks}\ (\mathit{Seq}\ s1\ s2) = \mathit{basicBlocks}\ s1 \mathbin{+\!\!+} \mathit{basicBlocks}\ s2\\
&\mathit{basicBlocks}\ s = [s]
\end{aligned}
$$

computeSuccessors is a wrapper for *computeSuccessors'* which filters out nonexistant labels.

$$
\begin{aligned}
&\mathit{computeSuccessors}\ s\ \mathit{dec} = \mathit{M.map}\ (\mathit{S.filter}\ (\mathit{flip}\ \mathit{S.member}\ \mathit{dec}))\\
&\quad (\mathit{computeSuccessors'}\ 0\ (\mathit{basicBlocks}\ s))
\end{aligned}
$$

computeSuccessors computes the successor sets for a block. If a block is an assignment or skip, then the successor is the next block.

$$
\begin{aligned}
&\mathit{computeSuccessors'}\ i\ ((\mathit{Assign}\ \_\ \_) : \mathit{ss}) =\\
&\mathit{M.union}\ m\ (\mathit{computeSuccessors'}\ (i+1)\ \mathit{ss})\ \mathbf{where}\\
&\quad m = \mathit{M.singleton}\ i\ (\mathit{S.singleton}\ (i+1))\\
&\mathit{computeSuccessors'}\ i\ (\mathit{Skip} : \mathit{ss}) =\\
&\mathit{M.union}\ m\ (\mathit{computeSuccessors'}\ (i+1)\ \mathit{ss})\ \mathbf{where}\\
&\quad m = \mathit{M.singleton}\ i\ (\mathit{S.singleton}\ (i+1))
\end{aligned}
$$

If the block is an "if" statement, then compute successors of the "then" and "else", and adjust the last blocks in both the "then" and "else" to be succeeded by the next basic block.

The boolean expression in the "if" statement is succeeded by both the "then" and "else" blocks.

$$
\begin{aligned}
&\mathit{computeSuccessors'}\ i\ ((\mathit{If}\ \_\ s1\ s2) : \mathit{ss}) =\\
&\mathit{M.unions}\ [m, m1', m2', \mathit{computeSuccessors'}\ c2\ \mathit{ss}]\ \mathbf{where}\\
&\quad m = \mathit{M.singleton}\ i\ (\mathit{S.fromList}\ [i+1, i+2])\\
&\quad c1 = \mathit{countBlocks}\ s1\\
&\quad c2 = \mathit{countBlocks}\ s2\\
&\quad b1 = \mathit{basicBlocks}\ s1
\end{aligned}
$$

$$
\begin{aligned}
&\quad b2 = basicBlocks\ s2\\
&\quad m1 = computeSuccessors'\ (i+1)\ b1\\
&\quad m1' = M.adjust\ (\backslash\_ \rightarrow S.singleton\ (i + c1 + c2 + 1))\ (length\ b1 + i)\ m1\\
&\quad m2 = computeSuccessors'\ (i + c1 + 1)\ b2\\
&\quad m2' = M.adjust\ (\backslash\_ \rightarrow S.singleton\ (i + c1 + c2 + 1))\ (length\ b2 + i)\ m2
\end{aligned}
$$

If the block is a While loop then it's boolean expression is succeeded by the first basic block in the loop, and the next first block after the loop. The successor of the last basic block in the loop is adjusted to be succeeded by the loop expression.

$$
\begin{aligned}
&computeSuccessors'\ i\ ((While\ \_\ s) : ss) =\\
&M.unions\ [m, m1', computeSuccessors'\ (c + i + 1)\ ss]\ \mathbf{where}\\
&\quad m = M.singleton\ i\ (S.fromList\ [i + 1, c + i + 1])\\
&\quad c = countBlocks\ s\\
&\quad bs = basicBlocks\ s\\
&\quad m1 = computeSuccessors'\ (i + 1)\ bs\\
&\quad m1' = updateLast\ m1\ (length\ bs + i)\ i\ (last\ bs)\\
&computeSuccessors'\ \_\ [\,] = M.empty\\
&updateLast\ m\ i\ v\ (If\ \_\ s1\ s2) = i2\ \mathbf{where}\\
&\quad i1 = M.adjust\ (\backslash\_ \rightarrow S.singleton\ v)\ (i + 1)\ m\\
&\quad i2 = M.adjust\ (\backslash\_ \rightarrow S.singleton\ v)\ (i + 2)\ i1\\
&updateLast\ m\ i\ v\ \_ = M.adjust\ (\backslash\_ \rightarrow S.singleton\ v)\ i\ m
\end{aligned}
$$

countBlocks returns the number of total blocks (not basic) in a program.

$$
\begin{aligned}
&countBlocks\ (Seq\ s1\ s2) = countBlocks\ s1 + countBlocks\ s2\\
&countBlocks\ (If\ \_\ s1\ s2) = 1 + (countBlocks\ s1 + countBlocks\ s2)\\
&countBlocks\ (While\ \_\ s) = 1 + countBlocks\ s\\
&countBlocks\ \_ = 1
\end{aligned}
$$

computePredecessors takes a collection of successor edges and returns a collection of predecessor edges. The predecessors of some block *b* is equivalent to the set of blocks with b as a successor.

$$
\begin{aligned}
&computePredecessors :: M.Map\ Int\ (S.Set\ Int)\\
&\quad \rightarrow M.Map\ Int\ (S.Set\ Int)\\
&computePredecessors\ outs = M.fromList \circ map\ go \circ M.keys\ \$\ outs\ \mathbf{where}\\
&\quad go\ i = (i, labelsWith\ i)\\
&\quad labelsWith\ i = S.fromList\ [j \mid (j, set) \leftarrow M.toList\ outs, S.member\ i\ set]\\
&displayLabeledGraph :: M.Map\ Int\ Block \rightarrow IO\ ()
\end{aligned}
$$

```
displayLabeledGraph = mapM_ (putStrLn ∘ showBlock) ∘ M.toList where
  showBlock (i, Left s) = "[" ++ (pettyShowStatement s) ++ "]" ++ (show i)
  showBlock (i, Right b) = "[" ++ (pettyShowBool b) ++ "]" ++ (show i)
parseGraph :: FilePath → IO ControlFlowGraph
parseGraph fp = do
  eG ← sparse <$> (readFile fp)
  return $ case eG of
    Right s → controlFlowGraph s
    Left e → error $ "A horrible parse error occurred: " ++ (show e)
makeTest :: String → FilePath → ControlFlowGraph → Test
makeTest name fp expected = TestLabel name $ TestCase $ (expected@ =?) =<< parseGraph fp
ifExampleTest = makeTest "ifexample" "tests\\ifexample.txt" expectedIf
expectedIf = CFG labels outs ins where
  labels = M.fromList [(0, Left (Assign "x" (BinOp Times (Number 2) (Number 200)))),
    (1, Right (RelOp Leq (Var "x") (Number 400))),
    (2, Left Skip), (3, Left (Assign "y" (Number 10)))]
  outs = M.fromList [(0, S.singleton 1), (1, S.fromList [2, 3]),
    (2, S.empty), (3, S.empty)]
  ins = M.fromList [(0, S.empty), (1, S.singleton 0), (2, S.singleton 1), (3, S.singleton 1)]
whileTest = makeTest "while" "tests\\while.txt" expectedWhile
expectedWhile = CFG labels outs ins where
  labels = M.fromList [(0, Left (Assign "y" (Var "x"))),
    (1, Left (Assign "z" (Number 1))),
    (2, Right (RelOp Greater (Var "y") (Number 0))),
    (3, Left (Assign "z" (BinOp Times (Var "z") (Var "y")))),
    (4, Left (Assign "y" (BinOp Minus (Var "y") (Number 1)))),
    (5, Left (Assign "y" (Number 0)))]
  outs = 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)]
  ins = 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)]
whileTest2 = makeTest "while" "tests\\while2.txt" expectedWhile2
expectedWhile2 = CFG labels outs ins where
  labels = M.fromList [(0, Right (RelOp Greater (Var "y") (Number 0))),
    (1, Left (Assign "z" (BinOp Times (Var "z") (Var "y"))))]
  outs = M.fromList [(0, S.singleton 1), (1, S.singleton 0)]
  ins = M.fromList [(0, S.singleton 1), (1, S.singleton 0)]
printd f = mapM_ print ∘ M.toList ∘ f
```

```haskell
tests :: Test
tests = TestList [ifExampleTest,
   whileTest,
   whileTest2,
   TestLabel "ast" $ (controlFlowGraph ast) ~?= cfg,
   TestLabel "ast2" $ (controlFlowGraph ast2) ~?= cfg2]

doTestsPass :: IO Bool
doTestsPass = do
   counts ← runTestTT tests
   let errs = errors counts
      fails = failures counts
   return $ (errs ≡ 0) ∧ (fails ≡ 0)

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
```

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,
  ReachingDefinition,
  reachingDefinitions,
  formatReachingDefinitions) where
import AST
import ControlFlow
import Data.List (intercalate)
import qualified Data.Map as M
import qualified Data.Set as S
```

A *ReachingDefinition* is a set of *String* variable names to *Maybe Int* where *Just l* is the last known label assignment and *Nothing* indicates that it is unknown when the element was last assigned.

```
type ReachingDefinition = S.Set (String, Maybe Int)
```

A *ReachingDefinitions* contains two maps from *Int* to *ReachingDefinitions*. The *Int* key is the label and the *ReachingDefinition* is the definition associated with that label.

```
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)
```

Given a *ControlFlowGraph*, *reachingDefinitions* returns the *ReachingDefinitions* for the provided *ControlFlowGraph*. It is assumed that for each key in labels, there is also a key in *outEdges* and *inEdges*. If this condition is not met, it is unknown what the result of this function will be.

$$
\begin{array}{l}
\mathit{reachingDefinitions} :: \mathit{ControlFlowGraph} \rightarrow \mathit{ReachingDefinitions} \\
\mathit{reachingDefinitions}\ \mathit{cfg} = \mathit{RDS}\ \mathit{entries}\ \mathit{exits}\ \mathbf{where} \\
\quad (\mathit{entries}, \mathit{exits}) = \mathit{reachingDefinitions}'\ (\mathit{empties}, \mathit{empties})\ \mathit{cfg} \\
\quad \mathit{empties} = \mathit{M.unions} \circ \mathit{map}\ ((\mathit{flip}\ \mathit{M.singleton})\ \mathit{S.empty})\ \$\ \mathit{lbls} \\
\quad \mathit{lbls} = \mathit{M.keys} \circ \mathit{labels}\ \$\ \mathit{cfg}
\end{array}
$$

Given a *ControlFlowGraph*, *formatEquations* returns a human readable *String* showing the entry, $RD\circ(x)$, and exit, $RD\bullet(x)$, equations for each label in the *ControlFlowGraph*. For example given the following simple graph:

```
simpleGraph:
0: [x := 0]
1: [y := 1]
while 2: [x < a + b] do
  3: [x := x + a]
  4: [a := a - b]
od
5: [b := b + x]
```

the command:

$$\mathit{putStrLn} \circ \mathit{formatEquations}\ \$\ \mathit{simpleGraph}$$

yields:

$$
\begin{aligned}
RD\circ(0) &= \{(a,?),(b,?),(x,?),(y,?)\}\cup\{\}\\
RD\circ(1) &= RD\bullet(0)\\
RD\circ(2) &= RD\bullet(1)\cup RD\bullet(4)\\
RD\circ(3) &= RD\bullet(2)\\
RD\circ(4) &= RD\bullet(3)\\
RD\circ(5) &= RD\bullet(2)\\
RD\bullet(0) &= RD\circ(0)\setminus\{(x,?),(x,0),(x,1),(x,2),\\
&\quad (x,3),(x,4),(x,5)\}\cup\{(x,0)\}\\
RD\bullet(1) &= RD\circ(1)\setminus\{(y,?),(y,0),(y,1),\\
&\quad (y,2),(y,3),(y,4),(y,5)\}\cup\{(y,1)\}\\
RD\bullet(2) &= RD\circ(2)\setminus\{\}\cup\{\}\\
RD\bullet(3) &= RD\circ(3)\setminus\{(x,?),(x,0),(x,1),\\
&\quad (x,2),(x,3),(x,4),(x,5)\}\cup\{(x,3)\}\\
RD\bullet(4) &= RD\circ(4)\setminus\{(a,?),(a,0),(a,1),\\
&\quad (a,2),(a,3),(a,4),(a,5)\}\cup\{(a,4)\}\\
RD\bullet(5) &= RD\circ(5)\setminus\{(b,?),(b,0),(b,1),\\
&\quad (b,2),(b,3),(b,4),(b,5)\}\cup\{(b,5)\}
\end{aligned}
$$

```
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
```

Given the *ReachingDefinitions* of a *ControlFlowGraph*, *formatReachingDefinitions* returns a human readable *String* showing the entry, $RD\circ(x)$, and exit, $RD\bullet(x)$, *ReachingDefinition* for each label. For example:

```
putStrLn ∘ formatReachingDefinitions ∘ reachingDefinitions $ simpleGraph
```

gives:

$$
\begin{aligned}
RD\circ(0) &= \{(a,?),(b,?),(x,?),(y,?)\}\\
RD\circ(1) &= \{(a,?),(b,?),(x,0),(y,?)\}\\
RD\circ(2) &= \{(a,?),(a,4),(b,?),(x,0),(x,3),(y,1)\}\\
RD\circ(3) &= \{(a,?),(a,4),(b,?),(x,0),(x,3),(y,1)\}\\
RD\circ(4) &= \{(a,?),(a,4),(b,?),(x,3),(y,1)\}\\
RD\circ(5) &= \{(a,?),(a,4),(b,?),(x,0),(x,3),(y,1)\}\\
RD\bullet(0) &= \{(a,?),(b,?),(x,0),(y,?)\}\\
RD\bullet(1) &= \{(a,?),(b,?),(x,0),(y,1)\}\\
RD\bullet(2) &= \{(a,?),(a,4),(b,?),(x,0),(x,3),(y,1)\}\\
RD\bullet(3) &= \{(a,?),(a,4),(b,?),(x,3),(y,1)\}\\
RD\bullet(4) &= \{(a,4),(b,?),(x,3),(y,1)\}\\
RD\bullet(5) &= \{(a,?),(a,4),(b,5),(x,0),(x,3),(y,1)\}
\end{aligned}
$$

```
formatReachingDefinitions :: ReachingDefinitions → String
formatReachingDefinitions (RDS entries exits) =
  (formatEntryDefs entries) ++ "\n" ++ (formatExitDefs exits)
simpleGraph :: ControlFlowGraph
simpleGraph = CFG labels outEdges inEdges where
  labels = M.fromList [(0, Left (Assign "x" (Number 0))),
    (1, Left (Assign "y" (Number 1))),
    (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)]
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))
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)
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
```

6 Main module

For the compiled binary, if run with a valid text file containing a program written in the simple imperative language, we decided to simply write the ast and cfg dot graphs to files, and print the reaching definitions analysis to stdout; otherwise a syntax error message is printed to stdout.

```
module Main where

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

main = do
  [file] ← getArgs
  contents ← readFile file
  let result = sparse contents
  case result of
    Right ast → do
      writeFile "ast.gv" (dotPrinter ast)
      let cfg = controlFlowGraph ast
      writeFile "cfg.gv" (formatCFGasDOT cfg)
      putStrLn ∘ formatReachingDefinitions ∘
```

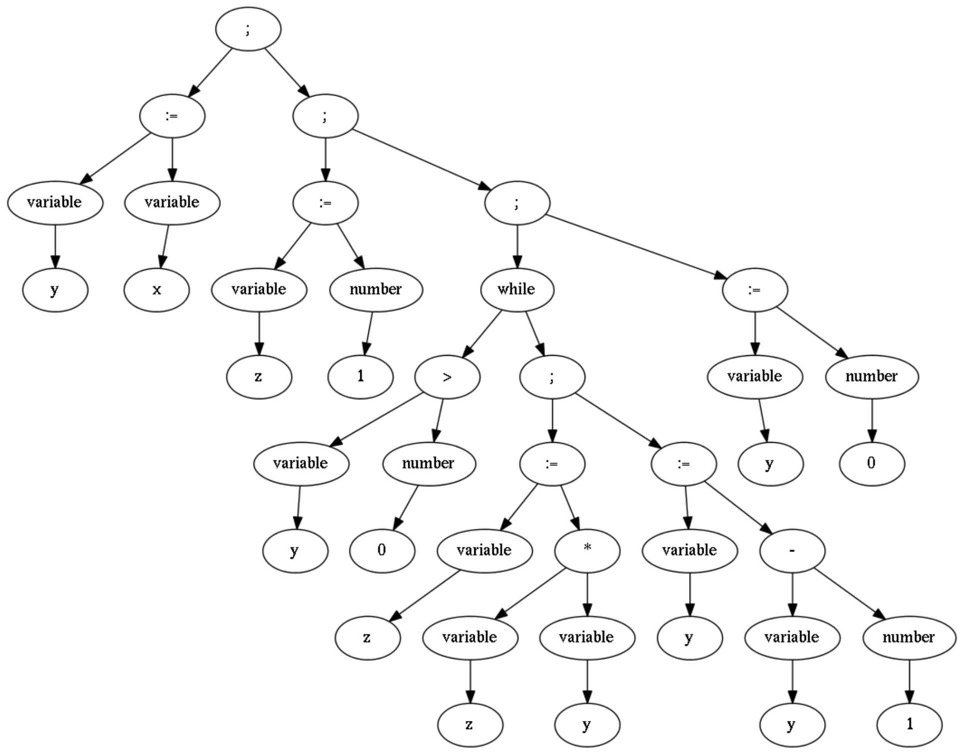
reachingDefinitions \$ *cfg*
Left err $\rightarrow$ *print err*

7 Example: while.txt Program

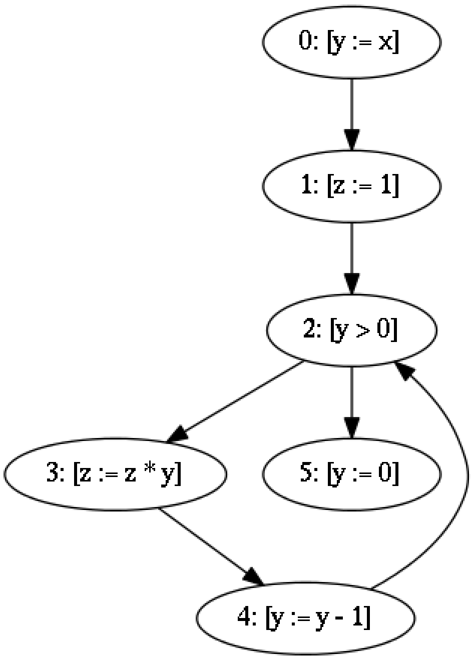
Given the following simple program:

```
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:



The control flow graph for the program is:



And finally, the entry and exit points for reaching definitions are:

$$
\begin{aligned}
RD\circ(0) &= \{(x,?),(y,?),(z,?)\} \\
RD\circ(1) &= \{(x,?),(y,0),(z,?)\} \\
RD\circ(2) &= \{(x,?),(y,0),(z,1),(z,3)\} \\
RD\circ(3) &= \{(x,?),(y,0),(z,1),(z,3)\} \\
RD\circ(4) &= \{\} \\
RD\circ(5) &= \{(x,?),(y,0),(z,1),(z,3)\} \\
RD\bullet(0) &= \{(x,?),(y,0),(z,?)\} \\
RD\bullet(1) &= \{(x,?),(y,0),(z,1)\} \\
RD\bullet(2) &= \{(x,?),(y,0),(z,1),(z,3)\} \\
RD\bullet(3) &= \{(x,?),(y,0),(z,3)\} \\
RD\bullet(4) &= \{\} \\
RD\bullet(5) &= \{(x,?),(y,5),(z,1),(z,3)\}
\end{aligned}
$$