Final Project: Data Flow Analysis

S. Patel, J. Collard, M. Barney

May 5, 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.

It is divided up into several sections, roughly corresponding to the problems given in the specification, each a Haskell module.

2 Abstract Syntax

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

```
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)
```

As can be seen, the abstract syntax, thanks to Haskell's recursive data types, almost exactly mirrors the Backus-Naur form given in the assignment.

In addition, 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 ++ " "
    ++ 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, We also wrote a printer that outputs dot syntax for drawing pretty graphs for the abstract syntax of a program.

```
dotPrinter' :: (Num t, Show t) => Statement -> t -> ([Char], t)
dotPrinter' (Assign s a) counter =
  let (arith, counter') = dotPrinterArith a counter
      s1 = "s_" ++ (show counter')
      s2 = "s_" ++ (show (counter' + 1))
      string = (s1 ++ " [label=\":=\"];\n" ++
        s2 ++ " [label=\"" ++ s ++ "\"];\n" ++
        s1 ++ " -> " ++ s2 ++ ";\n" ++
        s1 ++ " -> " ++ "a_" ++ (show counter') ++ ";\n" ++
        arith ++ ";\n") in
      (string, counter' + 2)

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))
```

```haskell
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 (s1, counter') = dotPrinterArith a1 counter
      (s2, counter'') = dotPrinterArith a2 counter'
      op = "a_" ++ (show (counter'' + 1))
      string = (op ++ " [label=\"" ++ (dotAOP aop) ++ "\"];\n" ++
        s1 ++ s2 ++
        op ++ " -> " ++ "a_" ++ (show counter) ++ ";\n" ++
        op ++ " -> " ++ "a_" ++ (show counter')
               ) in
  (string, (counter'' + 1))
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)) ++ "}")
main' = do
putStrLn $ dotPrinter (Assign "x" (BinOp Plus ((BinOp Times (Var "y") (Number 3))) (Nu
```

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. MENTION UNICODE

```
module Input (sparse) where

import AST
import Text.ParserCombinators.Parsec

type Program = [Statement]

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

  -- assignment must be last to preserve keywords
statement′ = skip < | > ifstatement < | > whilestatement < | > assignment

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

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
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
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 ">"; return $ RelOp Greater} <|>
  do {string "<"; return $ RelOp Less} <|>
  do {string "=="; return $ RelOp Equal} <|>
  do {string ">=" <|> string ""; return $ RelOp Geq} <|>
  do {string "<=" <|> string ""; return $ RelOp Leq}
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
whilestatement = do
  string "while"
  many1 space
  b ← boolean
  string "do"
  many1 space
```

```
  s ← statement
  string "od"
  return $ While b s
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)
```

```haskell
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,
```

```haskell
  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 (\str -> (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
```

```haskell
    keys = M.keys exits
    defs = zipWith formatExitDef keys (map (exits M.!) 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
entryEquations :: ControlFlowGraph -> [EntryEquation]
entryEquations cfg = zip lbls sets where
    x = inEdges cfg
    lbls = M.keys . labels $ cfg
    sets = map (x M.!) 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
```

```haskell
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 (map M.! 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
```

```haskell
main = do
  [file] <- getArgs
  contents <- readFile file
  case sparse contents of
    Right ast -> print ast
    Left err -> print err
```