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)
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 ++ " "
```

```haskell
    ++ 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
```

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.

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

```haskell
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
```

```haskell
  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)"
```

```
 {-# LANGUAGE ViewPatterns #-}
module ReachingDefinition (formatEquations, ReachingDefinitions, reachingDefinitions) wh
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 entry exit where
  (entry, exit) = reachingDefinitions′ False 1 (S.fromList lbls)
    (initEntry, initExits) cfg
    where
      initEntry = M.union (M.singleton 0 initialSet) empties
      initExits = M.union (M.singleton 0 initialSet′) empties
      vars = S.toList ∘ determineVars $ cfg
      initialSet = S.fromList ∘ map (λstr → (str, Nothing)) $ vars
      initialSet′ = initialSet ‘S.difference‘ kill ‘S.union‘ gen
      (_, kill, gen) = getExitEquation (S.fromList lbls) 0
        (labels cfgM.! 0)
      lbls = M.keys ∘ labels $ cfg
      empties = M.unions ∘ map ((flip M.singleton) S.empty) $ lbls
reachingDefinitions′ :: Bool → Int → S.Set Int →
    (EntryDefs, ExitDefs) →
    ControlFlowGraph → (EntryDefs, ExitDefs)
reachingDefinitions′ toStop l lbls (entry, exit) cfg =
  if stop then
    if isLoop then (entry″, exit″)
    else (entry′, exit′) else (entry″, exit″)
  where
    currEntry = (entryM.! l)
    (_, entEq) = entryEquation l cfg
    nextEntry = (S.unions exitSets)
    exitSets = map (exitM.!) (S.toList entEq)
    currExit = exitM.! l
    (_, kill, gen) = getExitEquation lbls l (labels cfgM.! l)
    nextExit = nextEntry ‘S.difference‘ kill ‘S.union‘ gen
    nextLabels = (S.toList (outEdges cfgM.! l))
    stop = null nextLabels ∨
      (currEntry ≡ nextEntry ∧ currExit ≡ nextExit ∧ isLoop)
    entry′ = M.insert l nextEntry entry
    exit′ = M.insert l nextExit exit
    entry″ = M.unions ∘ map fst $ branches
    exit″ = M.insert l
      ((entry″M.! l) ‘S.difference‘ kill ‘S.union‘ gen) exits
    exits = M.unions ∘ map snd $ branches
```

```
    rdef l = reachingDefinitions' stop l lbls (entry', exit') cfg
    branches = if toStop then [] else map rdef nextLabels
    isLoop = (S.size (inEdges cfgM.! l)) > 1
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
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
```

```haskell
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 (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),
```

$(5, S.empty)]$
$inEdges = M.fromList\ [(0, S.empty),$
$(1, S.singleton\ 0),$
$(2, S.fromList\ [1, 4]),$
$(3, S.singleton\ 2),$
$(4, S.singleton\ 3),$
$(5, S.singleton\ 2)]$

4 Main module

The main module puts everything together.

```
module Main where
import System.Environment
import AST
import Input
main = do
  [file] ← getArgs
  contents ← readFile file
  case sparse contents of
    Right ast → print ast
    Left err → print err
```