Project 2: Grammar Analysis and Parsing

S. Patel, J. Collard, M. Barney

April 17, 2013

1 Introduction

This report contains our implementation of a scanner and parser for context free grammars, a series of hygiene functions for sterilizing the grammar, and finally a parser *for* the grammar specified in the context free grammar.

It is divided up into several sections, roughly corresponding to the problems given in the specification, each a Haskell module. The work was split up evenly amongst the group members, and approximately 40 man hours went into the final preparation of this document, the source code, unit testing, and related work.

2 Context Free Grammar

In this section we provide the context free grammar data type.

At its heart, a grammar it consists of a list of productions, where each production consists of a constructor and two arguments; the first a paramaterized nonterminal, and the second a paramaterized right hand side.

An *RHS* is either empty, a terminal, which takes two arguments — the paramaterized object representing a terminal, and another *RHS*; or a nonterminal, which similarly takes two arguments.

```
{-# LANGUAGE FlexibleInstances, MultiParamTypeClasses #-}
module ContextFreeGrammar
(Grammar, Production (..), RHS (..), module Dropable,
nonTerminals, terminals, Terminal (..)) where

import Dropable
import Filterable
import Prelude hiding (drop, filter)

type Grammar nt t = [Production nt t]
```

```haskell
data Terminal t = Epsilon | EOF | Terminal t deriving (Show, Eq, Ord)

instance (Eq nt) => Dropable nt (Grammar nt t) where
  drop x grammar = map (drop x) grammar

instance Filterable (nt -> Bool) (Grammar nt t) where
  filter pred grammar = map (filter pred) grammar

data Production nt t = Production {nonterminal :: nt,
  rhs :: RHS nt t} deriving (Eq, Ord)

instance Show (Production String String) where
  show (Production nt rhs) = nt ++ " ->" ++ show rhs

instance Show (Production Char Char) where
  show (Production nt rhs) = show nt ++ " -> " ++ show rhs

instance (Eq nt) => Dropable nt (Production nt t) where
  drop x (Production nt rhs) = Production nt (drop x rhs)

instance Filterable (nt -> Bool) (Production nt t) where
  filter pred (Production nt rhs) = Production nt (filter pred rhs)

data RHS nt t = Empty
  | Term t (RHS nt t)
  | NonT nt (RHS nt t) deriving (Eq, Ord)

instance Show (RHS String String) where
  show Empty = ""
  show (Term t rhs) = " " ++ t ++ (show rhs)
  show (NonT nt rhs) = " " ++ nt ++ (show rhs)

instance Show (RHS Char Char) where
  show Empty = ""
  show (Term t rhs) = show t ++ (show rhs)
  show (NonT nt rhs) = show nt ++ (show rhs)

instance (Eq nt) => Dropable nt (RHS nt t) where
  drop x (NonT nt rhs)
    | x == nt = drop x rhs
    | otherwise = (NonT nt (drop x rhs))
  drop x (Term t rhs) = Term t (drop x rhs)
  drop _ Empty = Empty

instance Filterable (nt -> Bool) (RHS nt t) where
  filter _ Empty = Empty
  filter pred (Term t rhs) = (Term t (filter pred rhs))
  filter pred (NonT nt rhs) =
    if pred nt then (NonT nt (filter pred rhs)) else (filter pred rhs)
```

nonTerminals takes the RHS of a Production and returns a list of all Non Terminals

$$
\begin{array}{l}
\mathit{nonTerminals} :: \mathit{RHS}\ \mathit{nt}\ t \rightarrow [\mathit{nt}] \\
\mathit{nonTerminals}\ (\mathit{NonT}\ \mathit{nt}\ \mathit{rhs}) = \mathit{nt} : \mathit{nonTerminals}\ \mathit{rhs} \\
\mathit{nonTerminals}\ (\mathit{Term}\ \_\ \mathit{rhs}) = \mathit{nonTerminals}\ \mathit{rhs} \\
\mathit{nonTerminals}\ \mathit{Empty} = [\,]
\end{array}
$$

terminals takes the RHS of a Production and returns a list of all Terminals

$$
\begin{array}{l}
\mathit{terminals} :: \mathit{RHS}\ \mathit{nt}\ t \rightarrow [t] \\
\mathit{terminals}\ (\mathit{Term}\ t\ \mathit{rhs}) = t : \mathit{terminals}\ \mathit{rhs} \\
\mathit{terminals}\ (\mathit{NonT}\ \_\ \mathit{rhs}) = \mathit{terminals}\ \mathit{rhs} \\
\mathit{terminals}\ \mathit{Empty} = [\,]
\end{array}
$$

$$
\begin{array}{l}
\mathit{simpleGrammar} :: \mathit{Grammar}\ \mathit{String}\ \mathit{String} \\
\mathit{simpleGrammar} = [a, b, c, d]\ \textbf{where} \\
\quad a = \mathit{Production}\ \texttt{"A"}\ (\mathit{Term}\ \texttt{"a"}\ \mathit{Empty}) \\
\quad b = \mathit{Production}\ \texttt{"B"}\ (\mathit{NonT}\ \texttt{"B"}\ \mathit{Empty}) \\
\quad c = \mathit{Production}\ \texttt{"C"}\ (\mathit{Term}\ \texttt{"a"}\ (\mathit{NonT}\ \texttt{"B"}\ \mathit{Empty})) \\
\quad d = \mathit{Production}\ \texttt{"D"}\ (\mathit{NonT}\ \texttt{"B"}\ (\mathit{Term}\ \texttt{"a"}\ \mathit{Empty}))
\end{array}
$$

3 Scanner and Parser for context-free grammars

In this section we provide code for a simple scanner and parser for a textual representation of a context free grammar.

The grammar for the concrete representation follows the suggestion in the assignment, with one minor difference:

```
Grammar -> Grammar Production
Grammar -> Production
Production -> UpperSymbol Arrow RHS
RHS -> RHS Symbol
RHS ->
Symbol -> UpperSymbol
Symbol -> AlphaNumSymbol
```

In other words, non-terminals are restricted to their first letter being upper case, terminals are sequences of alphanumeric characters where the first character cannot be upper-case, and right hand side terminals and non-terminals are delimited by spaces.

A couple helper functions are initially defined, in addition to the grammar token data structure, which is as follows:

```haskell
module ScanAndParse (sparse) where

import ContextFreeGrammar
import Data.Char (isUpper, isSpace, isAlphaNum, isAlpha, isDigit)

data GrammarToken =
  Symbol String |
  ArrowToken |
  NewLineToken deriving (Show, Eq)

alphanumeric = takeWhile isAlphaNum

drop' _ [] = []
drop' i (x : xs) =
  if i <= 0 then (x : xs)
  else
    drop' (i - 1) xs
```

The scanner is a simple function that checks for two special characters, the arrow, `->` and the newline character, `\n`, scans symbols for nonterminals or terminals, and returns their appropriate tokens.

If a non alphanumeric character is found, the scanner returns an error.

```haskell
scan :: String -> [GrammarToken]
scan []                = []
scan ('-' : '>' : cs)  = ArrowToken : scan cs
scan ('\n' : cs)       = NewLineToken : scan cs
scan (c : cs) | isSpace c = scan cs
scan s@(c : cs) | isAlphaNum c =
  let name = alphanumeric s
    len = length name in
      (Symbol name) : scan (drop' len s)
scan s@(c : cs)        =
  error ("lexical error; " ++ c : " is an unrecognized character.")
```

The parser generates a list of productions, i.e., a "grammar", from a list of grammar tokens. The helper function, *parseRHS*, will throw a syntax error if an arrow token is found on the right hand side.

The function *parse* will throw an error if multiple non-terminals occur on the left-hand side, or an arrow is missing.

```
parseRHS :: [GrammarToken] → ((RHS String String), [GrammarToken])
parseRHS [ ] =
  (Empty, [ ])
parseRHS (NewLineToken : rhs) =
  (Empty, rhs)
parseRHS (ArrowToken : rhs) =
  error "syntax error; arrow token found on right hand side"
parseRHS ((Symbol (c : cs)) : rhs) =
  let (term, rhs') = parseRHS rhs in
  if isUpper c then
    ((NonT (c : cs) term), rhs')
  else
    ((Term (c : cs) term), rhs')

parse :: [GrammarToken] → Grammar String String
parse [ ] = [ ]
parse (NewLineToken : p) = parse p
parse ((Symbol s) : ArrowToken : rhs) =
  let (production, rhs') = parseRHS rhs in
  (Production s (production)) : parse rhs'
parse ((Symbol s) : rhs) =
  error "Missing arrow or multiple non-terminals on left-hand side."

sparse = parse ∘ scan
```

4 Hygiene Module

In this module, we perform basic hygiene checks on the grammar, remove unreachable non terminals, etc.

```
module BadHygiene (computeReachable,
  eliminateUnreachable,
  computeGenerating,
  eliminateNonGenerating,
  eliminateUseless,
  isEmptyGrammar) where

import ContextFreeGrammar
import qualified Data.Set as S
import Filterable
import ScanAndParse
```

{-BEGIN CLEANING FUNCTIONS -}

computeReachable finds the Set of all Non Terminals of a Grammar that can be reached from the start node.

$computeReachable :: Ord\ nt \Rightarrow Grammar\ nt\ t \rightarrow S.Set\ nt$
$computeReachable\ [\,] = S.empty$
$computeReachable\ ps = go\ (S.singleton \circ nonterminal \circ head\ \$\ ps)\ (concat \circ replicate\ (length$
$\quad go\ marked\ [\,] = marked$
$\quad go\ marked\ ((Production\ nt\ rhs) : prs) = \textbf{if}\ S.member\ nt\ marked$
$\qquad \textbf{then}\ go\ marked'\ prs$
$\qquad \textbf{else}\ go\ marked\ prs$
$\qquad \textbf{where}\ marked' = S.union\ marked \circ S.fromList \circ nonTerminals\ \$\ rhs$

eliminateUnreachable removes all unreachable Non Terminals from a Grammar.

$eliminateUnreachable :: Ord\ nt \Rightarrow Grammar\ nt\ t \rightarrow Grammar\ nt\ t$
$eliminateUnreachable\ g = cleanGrammar\ \textbf{where}$
$\quad reachable = computeReachable\ \$\ g$
-- unnecessary? By definition, the unreachable non-terminals cannot be in any
-- other production list.
-- cleanProductions = Filterable.filter (‘S.member‘ reachable) g
$\quad cleanGrammar = Prelude.filter\ (\lambda(Production\ nt\ rhs) \rightarrow S.member\ nt\ reachable)\ g$

computeGenerating finds the Set of all Non Terminals of a Grammar that can produce a string of Terminals.

$computeGenerating :: (Ord\ nt, Ord\ t) \Rightarrow Grammar\ nt\ t \rightarrow S.Set\ nt$
$computeGenerating\ [\,] = S.empty$
$computeGenerating\ ps = go\ S.empty\ (concat \circ replicate\ (length\ ps)\ \$\ ps)\ \textbf{where}$
$\quad allTerms = S.fromList \circ concatMap\ (terminals \circ rhs)\ \$\ ps$
$\quad go\ markedNT\ [\,] = markedNT$
$\quad go\ markedNT\ ((Production\ nt\ rhs) : prs) = \textbf{if}\ (all\ (`S.member`allTerms) \circ terminals\ \$\ rhs)$
$\qquad (all\ (`S.member`markedNT) \circ nonTerminals\ \$\ rhs)$
$\qquad \textbf{then}\ go\ (S.insert\ nt\ markedNT)\ prs$
$\qquad \textbf{else}\ go\ markedNT\ prs$

eliminateNonGenerating removes all non Generating Non Terminals from a Grammar.

$eliminateNonGenerating :: (Ord\ nt, Ord\ t) \Rightarrow Grammar\ nt\ t \rightarrow Grammar\ nt\ t$
$eliminateNonGenerating\ g = cleanGrammar\ \textbf{where}$

$generating = computeGenerating\ g$
$cleanProductions = Filterable.filter\ (`S.member`generating)\ g$
$cleanGrammar = Prelude.filter\ (\lambda(Production\ nt\ rhs) \rightarrow S.member\ nt\ generating)\ cleanPr$

eliminateUseless removes all non Generating and unreachable Non Terminals from a Grammar.

$eliminateUseless :: (Ord\ nt, Ord\ t) \Rightarrow Grammar\ nt\ t \rightarrow Grammar\ nt\ t$
$eliminateUseless = eliminateUnreachable \circ eliminateNonGenerating$

isEmptyGrammar determines if a Grammar will produce any strings at all.

$isEmptyGrammar :: (Ord\ t, Ord\ nt) \Rightarrow Grammar\ nt\ t \rightarrow Bool$
$isEmptyGrammar\ [\,] = True$
$isEmptyGrammar\ g = \neg \circ elem\ nt \circ map\ nonterminal\ \$\ g'$ **where**
$\quad g' = eliminateNonGenerating\ g$
$\quad (Production\ nt\ \_) = head\ g$
{-END CLEANING FUNCTIONS -}

5 Nullable, First, and Follow

In this section, we provide several modules for computing the nullable, first and follow sets of a given context free grammar, respectively.

5.1 Nullable

Here we compute whether a production is nullable or not.

module *Nullable* (*nullable*) **where**

import *ContextFreeGrammar*
import *qualified Data.Set as S*

import *Prelude hiding* (*drop*)

type $Set = S.Set$

$nullable :: (Ord\ nt) \Rightarrow Grammar\ nt\ t \rightarrow Set\ nt$
$nullable = nullable'\ S.empty$

$nullable' :: (Ord\ nt) \Rightarrow Set\ nt \rightarrow Grammar\ nt\ t \rightarrow Set\ nt$
$nullable'\ set\ grammar = set''$ **where**
$\quad set'' = $ **if** $nulls \equiv set$ **then** set **else** set'

```haskell
  set' = nullable' nulls (S.fold drop grammar nulls)
  nulls = S.fromList ∘ map nonterminal ∘ filter isNull $ grammar
isNull :: Production nt t → Bool
isNull (Production _ Empty) = True
isNull _ = False
simpleGrammar :: Grammar String String
simpleGrammar = [a] where
  a = Production "A" (Term "ab" Empty)
simpleGrammar2 :: Grammar String String
simpleGrammar2 = [a, a', b, b', c] where
  a = Production "A" (Term "ab" Empty)
  a' = Production "A" Empty
  b = Production "B" (NonT "A" (NonT "A" Empty))
  b' = Production "B" (NonT "A" (Term "b" Empty))
  c = Production "C" (Term "cdef" Empty)
```

```haskell
module First (first) where
import ContextFreeGrammar
import Control.Applicative
import Control.Monad
import Control.Monad.State
import Data.Functor
import Data.List
import qualified Data.Map as M
import Data.Maybe
import Nullable
import ScanAndParse
import qualified Data.Set as S
import Test.HUnit hiding (State)
first :: (Ord nt, Ord t) ⇒ Grammar nt t → M.Map nt (S.Set (Terminal t))
first g = firsts ∘ execState state ∘ FS M.empty ∘ nullable $ g where
  state = mapM first' ∘ concat ∘ replicate (length g) $ g
    -- state = mapM first' g
data FirstState nt t = FS {
  firsts :: M.Map nt (S.Set (Terminal t)),
  nulls :: S.Set nt
}
type Environment nt t a = State (FirstState nt t) a
```

```haskell
first' (Production nt rhs) = do
  fs ← get
  let mp = firsts fs
  case rhs of
    Empty → case M.lookup nt mp of
      Nothing → put fs {firsts = M.insert nt (S.singleton Epsilon) mp}
      Just _ → put fs {firsts = M.adjust (S.insert Epsilon) nt mp}
    _ → do
      sets ← firstRHS rhs
      let s = fromMaybe S.empty (M.lookup nt mp)
      put fs {firsts = M.insert nt (S.unions (s : sets)) mp}
firstRHS :: (Ord nt, Ord t) ⇒ RHS nt t → Environment nt t [S.Set (Terminal t)]
firstRHS Empty = return []
firstRHS (Term y _) = firstT y ≫= λs → return [s]
firstRHS (NonT y ys) = do
  nlls ← gets nulls
  case S.member y nlls of
    True → do
      set ← getFirsts y
      sets ← firstRHS ys
      return (set : sets)
    False → getFirsts y ≫= λs → return [s]
firstT :: Ord t ⇒ t → Environment nt t (S.Set (Terminal t))
firstT t = return ∘ S.singleton ∘ Terminal $ t
getFirsts nt = do
  set ← (M.lookup nt) < $ > gets firsts
  case set of
    Nothing → return S.empty
    Just set → return set
 {- BEGIN TESTS - -}
makeTestM :: (Eq a, Show a)
  ⇒ String
  → FilePath
  → String
  → a
  → (Grammar String String → a)
  → Test
makeTestM name file forF e f = TestLabel name ∘ TestCase $ do
```

```haskell
  grammar <- fmap sparse . readFile $ file
  assertEqual forF e (f grammar)
testFirst = makeTestM "testFirst"
  "tests\\test1.txt"
  "for first with test1"
  expected
  first where
  expected = M.fromList [("A", S.singleton . Terminal $ "a"),
    ("B", S.fromList [Terminal "b",
      Terminal "a",
      Epsilon]),
    ("C", S.fromList [Terminal "a",
      Terminal "b",
      Epsilon]),
      ("D", S.fromList [Terminal "a",
        Terminal "b",
        Epsilon])]
testFirst2 = makeTestM "testFirst2"
  "tests\\ir.txt"
  "for first with ir"
  expected
  first where
  expected = M.fromList [p, is, i]
  p = ("Program", S.fromList [Epsilon, Terminal "comma"])
  is = ("Instructions", S.fromList [Epsilon, Terminal "comma"])
  i = ("Instruction", S.fromList [Epsilon])
tests = TestList [testFirst,
  testFirst2]
runTests :: IO Counts
runTests = runTestTT tests
doTestsPass :: IO Bool
doTestsPass = do
  counts <- runTests
  let errs = errors counts
      fails = failures counts
  return $ (errs == 0) && (fails == 0)
```

5.2 Follow

In this section, we implement a function *follow* which calculates the follow set for our data structure of production grammars.

```
{-# LANGUAGE ViewPatterns #-}
module Follow where

import ContextFreeGrammar
import Control.Monad.State
import qualified Data.Map as M
import Data.Maybe
import qualified Data.Set as S
import First
import Nullable

data GrammarState nt t = GS {
  grammar :: Grammar nt t,
  firsts :: M.Map nt (S.Set (Terminal t))
}

follow :: (Ord nt, Ord t) => Grammar nt t -> M.Map nt (S.Set (Terminal t))
follow [] = M.empty
follow g@((Production nt rhs) : ps) = M.adjust (S.insert EOF) nt fMap where
  fMap = M.fromList $ zip (map nonterminal g) sets
  sets = evalState (mapM (follow'' . nonterminal) g) (GS g (first g))

follow' :: (Ord nt, Ord t) =>
  Grammar nt t -> Production nt t -> (nt, S.Set (Terminal t))
follow' g (Production a rhs) = (a, ⊥) where
  xs = getProductionsWith a g
  firsts = first g

follow'' :: (Ord nt, Ord t) =>
  nt -> State (GrammarState nt t) (S.Set (Terminal t))
follow'' a = do
  g <- gets grammar
  fs <- gets firsts
  let ps = getProductionsWith a g
  sets <- forM ps $ \(Production x (after a -> beta)) -> do
    case beta of
      Empty -> follow'' x
      NonT b _ -> do
        let firstb = fsM.! b
```

```
        case S.member Epsilon firstb of
          True → do
            folb ← follow′′ b
            let fb2 = S.delete Epsilon firstb
            return $ S.union folb fb2
          False → return firstb
      Term t _ → return ∘ S.singleton ∘ Terminal $ t
  return ∘ S.unions $ sets

getProductionsWith :: (Ord nt, Ord t) ⇒ nt → Grammar nt t → [Production nt t]
getProductionsWith nt ps = filter (elem nt ∘ nonTerminals ∘ rhs) ps

after :: (Eq nt) ⇒ nt → RHS nt t → RHS nt t
after nt Empty = Empty
after nt (Term t rhs) = after nt rhs
after nt (NonT nt2 rhs) = if nt ≡ nt2 then rhs else after nt rhs

simpleGrammar :: Grammar String String
simpleGrammar = [s, s′, b, a, c] where
  s = Production "S" (NonT "A" (NonT "B" Empty))
  s′ = Production "S" (Term "x" Empty)
  b = Production "B" (Term "b" Empty)
  a = Production "A" (Term "a" (NonT "A" Empty))
  c = Production "C" (Term "d" Empty)
```

6 Generating a Parse Table

In this section we generate a parse table for a given grammar, assuming it has been properly scanned, parsed, and thoroughly cleansed.

```
module Table where

import ContextFreeGrammar
import qualified Data.Map as M
import Filterable
import Nullable
import First
import Follow
import System.Environment

import Data.List

import ScanAndParse
import BadHygiene
```

```
type Table nt t = M.Map nt (M.Map t (Production nt t))
foo1 :: Grammar String String
foo1 = [a, b, c, d] where
  a = Production "A" (Term "a" Empty)
  b = Production "B" (NonT "B" Empty)
  c = Production "C" (Term "a" (NonT "B" Empty))
  d = Production "D" (NonT "B" (Term "a" Empty))
getFeature feature productions = loop productions []
  where
    loop [] acc = sort ∘ nub ∘ concat $ acc
    loop ((Production s rhs) : xs) acc = loop xs ((feature rhs) : acc)
buildTable grammar =
  let terms = getFeature terminals grammar in
  let nterms = getFeature nonTerminals grammar in
  terms
main = do
contents ← readFile "tests/39.txt"
let g′ = eliminateUseless ∘ sparse $ contents
let g″ = follow ∘ sparse $ contents
putStrLn $ show (getFeature terminals g′)
putStrLn $ show g″
```

7 Main module

The main module puts everything together, takes a textual representation of a context-free grammar as input, scans, parses, and performs the rest of the duties that are required.

```
module Main where
import ContextFreeGrammar
import ScanAndParse
import BadHygiene
import System.Environment
main = do
  -- [file] ¡- getArgs
  -- contents ¡- readFile file
  contents ← readFile "tests/39.txt"
```

-- contents ¡- readFile ”tests/ir.txt”
let $g = \mathit{sparse}\ \mathit{contents}$
let $g' = \mathit{eliminateUseless} \circ \mathit{sparse}\ \$\ \mathit{contents}$
$\mathit{putStrLn}\ \$\ \mathit{show}\ g$
$\mathit{putStrLn}\ \$\ \mathit{show}\ g'$