Project 2: Grammar Analysis and Parsing

S. Patel, J. Collard, M. Barney

April 21, 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.

Our design decisions with respect to the final problems for strong-LL(1) and non-strong-LL(1) table driven parsers is given in §6.1

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]
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, Show)
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, Show)
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

```
nonTerminals :: RHS nt t -> [nt]
nonTerminals (NonT nt rhs) = nt : nonTerminals rhs
nonTerminals (Term _ rhs) = nonTerminals rhs
nonTerminals Empty = []
```

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

```
terminals :: RHS nt t -> [t]
terminals (Term t rhs) = t : terminals rhs
```

```haskell
terminals (NonT _ rhs) = terminals rhs
terminals Empty = []

simpleGrammar :: Grammar String String
simpleGrammar = [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))
```

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)
```

```haskell
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.

```haskell
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 ⇒ Grammar nt t → S.Set nt
computeReachable [] = S.empty
computeReachable ps = go (S.singleton ∘ nonterminal ∘ head $ ps)
  (concat ∘ replicate (length ps) $ ps) where
  go marked [] = marked
  go marked ((Production nt rhs) : prs) = if S.member nt marked
    then go marked' prs
```

$\textbf{else}\ go\ marked\ prs$
$\textbf{where}\ marked' = S.union\ marked \circ S.fromList \circ nonTerminals\ \$\ rhs$

eliminateUnreachable removes all unreachable Non Terminals from a Grammar.

$eliminateUnreachable :: Ord\ nt \Rightarrow Grammar\ nt\ t \rightarrow Grammar\ nt\ t$
$eliminateUnreachable\ g = cleanGrammar\ \textbf{where}$
$\quad reachable = computeReachable\ \$\ g$
$\qquad$ -- unnecessary? By definition, the unreachable non-terminals cannot be in any
$\qquad$ -- other production list.
$\qquad$ -- 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.

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

eliminateNonGenerating removes all non Generating Non Terminals from a Grammar.

$eliminateNonGenerating :: (Ord\ nt, Ord\ t) \Rightarrow Grammar\ nt\ t \rightarrow Grammar\ nt\ t$
$eliminateNonGenerating\ g = cleanGrammar\ \textbf{where}$
$\quad generating = computeGenerating\ g$
$\quad cleanProductions = Filterable.filter\ (`S.member`generating)\ g$
$\quad cleanGrammar = Prelude.filter$
$\qquad (\lambda(Production\ nt\ rhs) \rightarrow S.member\ nt\ generating)\ cleanProductions$

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.

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

{-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.

$$\mathbf{module}\ Nullable\ (nullable)\ \mathbf{where}$$
$$\mathbf{import}\ ContextFreeGrammar$$
$$\mathbf{import}\ qualified\ Data.Set\ as\ S$$
$$\mathbf{import}\ Prelude\ hiding\ (drop)$$
$$\mathbf{type}\ Set = S.Set$$
$$nullable :: (Ord\ nt) \Rightarrow Grammar\ nt\ t \rightarrow Set\ nt$$
$$nullable = nullable'\ S.empty$$
$$nullable' :: (Ord\ nt) \Rightarrow Set\ nt \rightarrow Grammar\ nt\ t \rightarrow Set\ nt$$
$$nullable'\ set\ grammar = set''\ \mathbf{where}$$
$$\quad set'' = \mathbf{if}\ nulls \equiv set\ \mathbf{then}\ set\ \mathbf{else}\ set'$$
$$\quad set' = nullable'\ nulls\ (S.fold\ drop\ grammar\ nulls)$$
$$\quad nulls = S.fromList \circ map\ nonterminal \circ filter\ isNull\ \$\ grammar$$
$$isNull :: Production\ nt\ t \rightarrow Bool$$
$$isNull\ (Production\ \_\ Empty) = True$$
$$isNull\ \_ = False$$
$$simpleGrammar :: Grammar\ String\ String$$

$$
\begin{array}{l}
\mathit{simpleGrammar} = [\,a\,]\ \textbf{where} \\
\quad a = \mathit{Production}\ \texttt{"A"}\ (\mathit{Term}\ \texttt{"ab"}\ \mathit{Empty}) \\
\mathit{simpleGrammar2} :: \mathit{Grammar}\ \mathit{String}\ \mathit{String} \\
\mathit{simpleGrammar2} = [\,a, a', b, b', c\,]\ \textbf{where} \\
\quad a = \mathit{Production}\ \texttt{"A"}\ (\mathit{Term}\ \texttt{"ab"}\ \mathit{Empty}) \\
\quad a' = \mathit{Production}\ \texttt{"A"}\ \mathit{Empty} \\
\quad b = \mathit{Production}\ \texttt{"B"}\ (\mathit{NonT}\ \texttt{"A"}\ (\mathit{NonT}\ \texttt{"A"}\ \mathit{Empty})) \\
\quad b' = \mathit{Production}\ \texttt{"B"}\ (\mathit{NonT}\ \texttt{"A"}\ (\mathit{Term}\ \texttt{"b"}\ \mathit{Empty})) \\
\quad c = \mathit{Production}\ \texttt{"C"}\ (\mathit{Term}\ \texttt{"cdef"}\ \mathit{Empty})
\end{array}
$$

5.2 First

In this section, we compute the first set for a context-free grammar.

$$
\begin{array}{l}
\textbf{module}\ \mathit{First}\ (\mathit{first})\ \textbf{where} \\
\textbf{import}\ \mathit{ContextFreeGrammar} \\
\textbf{import}\ \mathit{Control.Monad} \\
\textbf{import}\ \mathit{Control.Monad.State} \\
\textbf{import}\ \mathit{Data.Functor} \\
\textbf{import}\ \mathit{Data.List} \\
\textbf{import}\ \mathit{qualified}\ \mathit{Data.Map}\ \mathit{as}\ M \\
\textbf{import}\ \mathit{Data.Maybe} \\
\textbf{import}\ \mathit{Nullable} \\
\textbf{import}\ \mathit{ScanAndParse} \\
\textbf{import}\ \mathit{qualified}\ \mathit{Data.Set}\ \mathit{as}\ S \\
\textbf{import}\ \mathit{Test.HUnit}\ \mathit{hiding}\ (\mathit{State})
\end{array}
$$

first is the interface function exported for general use. Given a Grammar, first computes the First Set for each Production, and returns the Sets in a Map from a Non-terminal to it's First Set.

$$
\begin{array}{l}
\mathit{first} :: (\mathit{Ord}\ \mathit{nt}, \mathit{Ord}\ t) \\
\quad \Rightarrow \mathit{Grammar}\ \mathit{nt}\ t \\
\quad \rightarrow M.\mathit{Map}\ \mathit{nt}\ (S.\mathit{Set}\ (\mathit{Terminal}\ t)) \\
\mathit{first}\ g = \mathit{firsts} \circ \mathit{execState}\ \mathit{state} \circ \mathit{FS}\ M.\mathit{empty} \circ \mathit{nullable}\ \$\ g\ \textbf{where} \\
\quad \mathit{state} = \mathit{mapM}\ \mathit{first}' \circ \mathit{concat} \circ \mathit{replicate}\ (\mathit{length}\ g)\ \$\ g
\end{array}
$$

The FirstState Data Type stores the map of Sets of First Terminals that is modified and returned at the end of a call to first. It also stores the set of Non-terminals which are nullable.

```haskell
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
```

first' does the work for the first function. Given a production, first' will calculate the set of first terminals and store it in the implicit FirstState.

```haskell
first' :: (Ord nt, Ord t) => Production nt t -> Environment nt t ()
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 is a helper function which, given a RHS will return the first sets of every terminal and non-terminal until a non nullable terminal/non-terminal is found.

```haskell
firstRHS :: (Ord nt, Ord t)
  => RHS nt t
  -> Environment nt t [S.Set (Terminal t)]
firstRHS Empty = return []
firstRHS (Term y _) = return [S.singleton . Terminal $ y]
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
```

getFirsts is a helper function which given a Non-terminal will return its current first set.

```
getFirsts :: Ord nt ⇒ nt → Environment nt t (S.Set (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
  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\\39.txt"
  "for first with 39"
  expected
  first where
  expected = M.fromList [("T", S.fromList [Epsilon, Terminal "a", Terminal "b"]),
    ("R", S.fromList [Epsilon, Terminal "b"])]
```

```
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)
 {- END TESTS - -}
```

5.3 Follow

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

```
module Follow (follow) where
import ContextFreeGrammar
import Control.Monad.State
import Data.Functor
import qualified Data.Map as M
import Data.Maybe
import qualified Data.Set as S
import First
import Nullable
import ScanAndParse
import Test.HUnit hiding (State)
```

A GrammarState holds data that the follow' function requires to work.

```
data GrammarState nt t = GS {
  grammar :: Grammar nt t,
  follows :: M.Map nt (S.Set (Terminal t)),
  firsts :: M.Map nt (S.Set (Terminal t))
}
type Environment nt t a = State (GrammarState nt t) a
```

follow is the interface function exported for general use. Given a Grammar, follow computes the Follow Set for each Production, and returns the Sets in a Map from a Non-terminal to it's Follow Set.

```
follow :: (Ord nt, Ord t)
   ⇒ Grammar nt t
   → M.Map nt (S.Set (Terminal t))
follow [ ] = M.empty
follow g@((Production s rhs) : ps) = fs where
  state = mapM follow′ ∘ concat ∘ replicate (length g) $ g
  fs = follows ∘ execState state ∘ GS g initial ∘ first $ g
  initial = M.singleton s (S.singleton EOF)
```

follow' is where the main work of the follow function is done. For a given production, follow' will add an entry into the GrammarState passed along. This function is meant to be mapM'd across the Grammar you want to compute the follow sets of.

```
follow′ :: (Ord nt, Ord t)
   ⇒ Production nt t
   → Environment nt t ()
follow′ (Production a _) = do
  fllostate ← get
  let g = grammar fllostate
      fllow = follows fllostate
      ps = filter (elem a ∘ nonTerminals ∘ rhs) g
  sets ← forM ps $ λ(Production x rhs) → do
    case after a rhs of
      Empty → return ∘ fromMaybe S.empty ∘ M.lookup x $ fllow
      Term t _ → return ∘ S.singleton ∘ Terminal $ t
      NonT nt rest → do
        let frstb = (firsts fllostate)M.! nt
          fllowx = case S.member Epsilon frstb of
            False → S.empty
            True → fromMaybe S.empty ∘ M.lookup x $ fllow
        return $ S.union (S.delete Epsilon frstb) (fllowx)
  let s = fromMaybe S.empty (M.lookup a fllow)
      newS = S.union s (S.unions sets)
  put fllostate {follows = M.insert a newS fllow}
```

after is a helper function which removes all Terminals and Non-terminals from a RHS until a specific Non-terminal is reached. Then the rest of the RHS is returned.

```
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
 {– 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
  grammar ← sparse < $ > readFile file
  assertEqual forF e (f grammar)
testFollow = makeTestM "testFollow"
  "tests\\39.txt"
  "for first with 39"
  expected
  follow where
  expected = M.fromList [("R", S.fromList [EOF, Terminal "c"]),
    ("T", S.fromList [EOF, Terminal "c"])]
tests = TestList [testFollow]
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)
 {– END TESTS - -}
```

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
```

```haskell
import ContextFreeGrammar
import Filterable
import Nullable
import First
import Follow
import System.Environment
import Data.List as L
import qualified Data.Map as M
import qualified Data.Set as S
import ScanAndParse
import BadHygiene
type Table = M.Map (String, String) (Production String String)
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)
  -- need to generate S' -> S$
getNewStart :: Grammar String String -> Grammar String String
getNewStart [] = error "getNewStart run on empty grammar --- a new low point"
getNewStart g@(Production nt rhs : ps) =
    (Production (nt ++ "'") (NonT nt (Term "$" Empty))) : g
isRHSNullable :: Ord a => S.Set a -> RHS a t -> Bool
isRHSNullable _ Empty = True
isRHSNullable _ (Term t _) = False
isRHSNullable m (NonT nt rhs) =
  if S.member nt m then
    isRHSNullable m rhs
  else False
firstRHS _ Empty = S.empty
firstRHS i (Term t rhs) = S.singleton t
firstRHS i (NonT nt rhs) =
```

```
    if (S.member nt (nulls i)) then
      S.union firstsnt (firstRHS i rhs)
    else
      firstsnt
    where firstsnt = ((firsts i) M.! nt)
  data GrammarInfo = GI {
    firsts :: M.Map String (S.Set String),
    follows :: M.Map String (S.Set String),
    nulls :: S.Set String
  } deriving Show
```

A production N $\rightarrow \alpha$ is in the table (N,a) *iff* a is in first(α) $\vee$ (nullable(α) $\wedge$ a is in follow(α)). This condition is readily translateable into our code:

```
validEntry gi (Production nterm α) term = firstalpha ∨ (nullablea ∧ follown)
  where
    firstalpha = S.member (term) (firstRHS gi α)
    nullablea = isRHSNullable (nulls gi) α
    follown = S.member (term) ((follows gi) M.! nterm)
```

We'll also need some conversion from the Terminal data types used by the first and follow functions:

```
toTerminal [] = []
toTerminal ("$" : ss) = EOF : toTerminal ss
toTerminal (s : ss) = Terminal s : toTerminal ss
fromTerminal' :: Terminal String → String
fromTerminal' Epsilon = ""
fromTerminal' EOF     = "$"
fromTerminal' (Terminal t) = t
fromTerminalSet ts = S.map fromTerminal' ts
```

Lastly, we build a table for a (strong) LL(1) parser, returning nothing if there is more than one production per entry in the table. In the event that we have more than one entry per spot in the table, we will run a different table building function, whose entries are *lists*, to represent the different possible production rules that apply for a given input token and a non-terminal.

```
buildTable' :: GrammarInfo → Grammar String String →
  [String] → Table → Maybe Table
buildTable' _ [] _ acc = Just acc
```

```
buildTable' gi (p@(Production nt _) : ps) terms acc =
  case fromList acc kvs of
    Nothing → Nothing
    Just a → buildTable' gi ps terms a
  where
    valids = L.filter (validEntry gi p) terms
    kvs = map (λv → ((nt, v), p)) valids
fromList :: Table →
  [((String, String), Production String String)] → Maybe Table
fromList acc [] = Just acc
fromList acc ((k@(nt, t), p) : ks) = case M.lookup k acc of
  Nothing → fromList (M.insert k p acc) ks
  Just _ → Nothing
buildTable grammar =
  let terms = getFeature terminals grammar in
  let gi = GI (M.map fromTerminalSet $ first grammar)
    (M.map fromTerminalSet $ follow grammar) (nullable grammar) in
  let table = buildTable' gi grammar terms M.empty in
  table
```

Finally, we conclude with code for building a so-called "ambiguous" table. As mentioned above, this table building function is run when we try to build a table for a strong LL(1) parser. If that fails, we build a table which allows multiple entries at each index, which we represent by lists.

```
  -- an ambiguous table
type TableA = M.Map (String, String) ([Production String String])
fromListA :: TableA →
  [((String, String), Production String String)] → TableA
fromListA acc [] = acc
fromListA acc ((k@(nt, t), p) : ks) = case M.lookup k acc of
  Nothing → fromListA (M.insert k [p] acc) ks
  Just _ → fromListA (M.adjust (p:) k acc) ks
buildTableA' :: GrammarInfo → Grammar String String →
  [String] → TableA → TableA
buildTableA' _ [] _ acc = acc
buildTableA' gi (p@(Production nt _) : ps) terms acc =
  buildTableA' gi ps terms (fromListA acc kvs)
    where
      valids = L.filter (validEntry gi p) terms
```

$$kvs = map\ (\lambda v \rightarrow ((nt, v), p))\ valids$$

$$buildTableA\ grammar =$$

$$\textbf{let}\ terms = getFeature\ terminals\ grammar\ \textbf{in}$$

$$\textbf{let}\ gi = GI\ (M.map\ fromTerminalSet\ \$\ first\ grammar)$$

$$(M.map\ fromTerminalSet\ \$\ follow\ grammar)\ (nullable\ grammar)\ \textbf{in}$$

$$\textbf{let}\ table = buildTableA'\ gi\ grammar\ terms\ M.empty\ \textbf{in}$$

$$table$$

6.1 Table-driven Parser module

In this section we provide a simple table-driven parser function, *parseWithTable*. It takes a start non-terminal symbol, a table created in the *Table* module, an input string, and returns a boolean indicating whether the string was successfully parsed or not.

Since we are just creating a parser, but do not know the intended use of the parser, we decided to simply return a boolean reporting whether the parse was successful or not. If necessary, it could be easily modified for other accomodations.

Another design decision was to emit Haskell code as a module for either strong LL(1) or non strong LL(1) parsers, to run on a given text file as input for that the context free grammar it was generated for.

For the strong LL(1) case, our parser behaves as required. The more interesting problem was writing, in Haskell, a parser for non strong LL(1) grammers which attempt to parse in the LL(1) manner, and "dynamically detect" if the grammar isn't even LL(1).

Our solution ended up being a "minor" modification of the strong LL(1) parser; we decided that the necessary behavior for parsing a non-deterministic choice in the table was to essentially map the parser over the list, and then take the union of the solutions that the parsers recursively return.

In other words, we try all the cases, or as the handout stated: "if we have no means to decide which right-hand side to select, we have to try them all."

Surprisingly, by changing the helper function slightly for checking whether the leftmost symbol on the right hand side of a production matches the production rule (i.e., whether it was left recursive or not), we were able to parse a whole new class of grammars which caused our program to loop infinitely before.

In the end, our implementation for non-strong LL(1) parsers emits a module which marks it as non-strong, and parses, if it has to, in an LL(k) manner. We would have liked to have written a parser that failed, or was unable to perform for idiosyncratic grammars, but due to a lack of time, we were at a loss on how to subvert the power of Haskell.

Another interesting feature would have been to check *how* more than one entry was added to the table. In other words, how the following logical statement was falsified: a production N $\rightarrow \alpha$ is in the table (N,a) *iff* a is in first(α) $\vee$ (nullable(α) $\wedge$ a is in follow(α)).

There are three possible ways it can be made false: the left disjunct is false, and the left conjunct is false; the left disjunct is false, and the right conjunct is false; and the left disjunct is false and both conjuncts in the conjunction are false.

These three different possibilities correspond to bullet points on pg. 248 of the handout, and give hints at how to "dynamically" detect features about the given grammar.

Lastly, it would have been better for us to write actual functions which directly eliminate left recursion, and perform left factorization, rather than implementing these features dynamically, but again, we didn't have the time and opted for a more "hackish" approach.

```
module Parser where
import ContextFreeGrammar
import Data.Char
import System.Environment
import qualified Data.Map as M
import ScanAndParse
import BadHygiene
import Table
getStart :: Grammar String String → String
getStart ((Production nt _): _) = nt
rhsString =
  "rhsToList :: RHS String String -> [String]\n" ++
  "rhsToList Empty = []\n" ++
  "rhsToList (Term t rhs) = t : rhsToList rhs\n" ++
  "rhsToList (NonT nt rhs) = nt : rhsToList rhs"
parseWithTableString =
  "\nparseWithTable start t s = parse' [start] s where\n" ++
  "  parse' [] cs = True\n" ++
  "  parse' (top:rest) s@(c:cs)\n" ++
  "    | isUpper (head top) = case M.lookup (top,[c]) t of\n" ++
```

```
"      Nothing -> False\n" ++
"      Just (Production _ rhs) -> parse' (rhsToList rhs ++ rest) s\n" ++
"    | c == head top = parse' rest cs\n" ++
"    | otherwise = False\n"
generateStrong grammar table = do
  putStrLn "module LLParser (parse) where "
  putStrLn "import qualified Data.Map as M"
  putStrLn "import ContextFreeGrammar"
  putStrLn "import Data.Char"
  putStr "\ntable = M."
  putStrLn . show $ table
  putStrLn parseWithTableString
  putStrLn rhsString
  putStr $ "parse s = parseWithTable "
    ++ "\"" ++ (getStart grammar) ++ "\""
    ++ " table " ++ "(s++\"$\")" ++ "\n"
parseTableAString =
  "\nnparseWithTable start t s = parse' [start] s where" ++
  "  parse' [] cs= True\n" ++
  "  parse' (top:rest) s@(c:cs)\n" ++
  "    | isUpper (head top) = case M.lookup (top,[c]) t of\n" ++
  "      Nothing -> False\n" ++
  "      Just (Production _ rhs) -> parse' (rhsToList rhs ++ rest) s\n" ++
  "    | c == head top = parse' rest cs\n" ++
  "    | otherwise = False\n"
parseWithTableAString =
  "\nparseWithTableA start t s = parse' [start] s where\n" ++
  "  parse' [] cs = True\n" ++
  "  parse' _ []  = False\n" ++
  "  parse' st@(top:rest) s@(c:cs)\n" ++
  "    | isUpper (head top) = case M.lookup (top,[c]) t of\n" ++
  "      Nothing -> False\n" ++
  "      Just [Production _ rhs] -> parse' (rhsToList rhs ++ rest) s\n" ++
  "      Just ps -> or . map (helper rest s) $ ps\n" ++
  "    | c == head top = parse' rest cs\n" ++
  "    | otherwise = False\n" ++
  "  helper rest str (Production nt rhs) = \n" ++
  "    case nt == head newStack of\n" ++
  "         False -> parse' newStack str\n" ++
```

```
  "          True -> parse' (tail newStack) str\n" ++
  "    where newStack = rhsToList rhs ++ rest\n"
rhsToList :: RHS String String -> [String]
rhsToList Empty = []
rhsToList (Term t rhs) = t : rhsToList rhs
rhsToList (NonT nt rhs) = nt : rhsToList rhs
generate grammar table = do
  putStrLn "module NonLLParser (parse) where "
  putStrLn "import qualified Data.Map as M"
  putStrLn "import ContextFreeGrammar"
  putStrLn "import Data.Char"
  putStr "\ntableA = M."
  putStrLn . show $ table
  putStrLn parseWithTableAString
  putStrLn rhsString
  putStr $ "parse s = parseWithTableA "
    ++ "\"" ++ (getStart grammar) ++ "\""
    ++ " tableA " ++ "(s++\"$\")" ++ "\n"
```

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.

Different tables are built depending on whether the function for building a strong LL(1) parse table returns *Nothing* or not.

```
module Main where
import ContextFreeGrammar
import ScanAndParse
import BadHygiene
import Table
import Parser
import qualified Data.Map as M
import System.Environment
main = do
  [file] <- getArgs
  contents <- readFile file
```

$$
\begin{array}{l}
\textbf{let}\ grammar = sparse\ contents \\
\textbf{let}\ grammar' = eliminateUseless \circ getNewStart\ \$\ grammar \\
\textbf{let}\ table = buildTable\ grammar' \\
\textbf{case}\ table\ \textbf{of} \\
\quad Nothing \rightarrow \textbf{do} \\
\quad\quad generate\ grammar'\ (buildTableA\ grammar') \\
\quad Just\ t \rightarrow \\
\quad\quad generateStrong\ grammar'\ t
\end{array}
$$

8 Conclusion and Sample Output

This concludes our implementation. We have provided some sample output from the various stages of our implementation. Two different simple context-free grammars are considered, `39.txt` and `39_ambiguous.txt`.

8.1 Strong LL(1) Parser for 39.txt

8.1.1 Grammar

```
T -> R
T -> a T c
R ->
R -> b R
```

8.1.2 Table

```
(("R","$"),Production {nonterminal = "R", rhs = Empty})
(("R","b"),Production {nonterminal = "R", rhs = Term "b" (NonT "R" Empty)})
(("R","c"),Production {nonterminal = "R", rhs = Empty})
(("T","$"),Production {nonterminal = "T", rhs = NonT "R" Empty})
(("T","a"),Production {nonterminal = "T"
          , rhs = Term "a" (NonT "T" (Term "c" Empty))})
(("T","b"),Production {nonterminal = "T", rhs = NonT "R" Empty})
(("T","c"),Production {nonterminal = "T", rhs = NonT "R" Empty})
(("T'","$"),Production {nonterminal = "T'", rhs = NonT "T" (Term "$" Empty)})
(("T'","a"),Production {nonterminal = "T'", rhs = NonT "T" (Term "$" Empty)})
(("T'","b"),Production {nonterminal = "T'", rhs = NonT "T" (Term "$" Empty)})
```

8.1.3 Code

```
module LLParser (parse) where
import qualified Data.Map as M
import ContextFreeGrammar
import Data.Char

parseWithTable start t s = parse' [start] s where
  parse' [] cs = True
  parse' (top:rest) s@(c:cs)
    | isUpper (head top) = case M.lookup (top,[c]) t of
      Nothing -> False
      Just (Production _ rhs) -> parse' (rhsToList rhs ++ rest) s
    | c == head top = parse' rest cs
    | otherwise = False

rhsToList :: RHS String String -> [String]
rhsToList Empty = []
rhsToList (Term t rhs) = t : rhsToList rhs
rhsToList (NonT nt rhs) = nt : rhsToList rhs
parse s = parseWithTable "T'" table (s++"$")
```

8.2 LL(1) Parser for 39_ambiguous.txt

8.2.1 Grammar

```
T -> R
T -> a T c
R ->
R -> R b R
```

8.2.2 Table

```
(("R","$"),[Production {nonterminal = "R", rhs = Empty}])
(("R","b"),[Production {nonterminal = "R"
          , rhs = NonT "R" (Term "b" (NonT "R" Empty))}
          ,Production {nonterminal = "R", rhs = Empty}])
(("R","c"),[Production {nonterminal = "R", rhs = Empty}])
(("T","$"),[Production {nonterminal = "T", rhs = NonT "R" Empty}])
(("T","a"),[Production {nonterminal = "T"
          , rhs = Term "a" (NonT "T" (Term "c" Empty))}])
```

```
(("T","b"),[Production {nonterminal = "T", rhs = NonT "R" Empty}])
(("T","c"),[Production {nonterminal = "T", rhs = NonT "R" Empty}])
(("T'","$"),[Production {nonterminal = "T'", rhs = NonT "T" (Term "$" Empty)}])
(("T'","a"),[Production {nonterminal = "T'", rhs = NonT "T" (Term "$" Empty)}])
(("T'","b"),[Production {nonterminal = "T'", rhs = NonT "T" (Term "$" Empty)}])
```

8.2.3 Code

```
module NonLLParser (parse) where
import qualified Data.Map as M
import ContextFreeGrammar
import Data.Char

parseWithTableA start t s = parse' [start] s where
  parse' [] cs = True
  parse' _ []  = False
  parse' st@(top:rest) s@(c:cs)
    | isUpper (head top) = case M.lookup (top,[c]) t of
      Nothing -> False
      Just [Production _ rhs] -> parse' (rhsToList rhs ++ rest) s
      Just ps -> or . map (helper rest s) $ ps
    | c == head top = parse' rest cs
    | otherwise = False
  helper rest str (Production nt rhs) =
    case nt == head newStack of
         False -> parse' newStack str
         True -> parse' (tail newStack) str
    where newStack = rhsToList rhs ++ rest

rhsToList :: RHS String String -> [String]
rhsToList Empty = []
rhsToList (Term t rhs) = t : rhsToList rhs
rhsToList (NonT nt rhs) = nt : rhsToList rhs
parse s = parseWithTableA "T'" tableA (s++"$")
```