Project 2: Grammar Analysis and Parsing

S. Patel, J. Collard, M. Barney

April 7, 2013

1 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) where
import Dropable
import Filterable
import Prelude hiding (drop, filter)
type Grammar nt t = [Production nt t]
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 rh
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))
```

2 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 -> LowerSymbol
```

In other words, non-terminals are restricted to being upper case (in our case, only the first letter needs to be capitalized), terminals are lower case (the first letter), neither can begin with a numeral, 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:

```
module ScanAndParse 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.

```
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) | isAlpha c =
```

```
    let name = alphanumeric s
        len = length name in
      (Symbol name) : scan (drop' len s)
  scan s@(c : cs) | isDigit c =
    error "lexical error; symbols cannot begin with numerals."
  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 : []) = []
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."
```

3 Main module

The main module puts everything together, takes an 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
  putStrLn $ show contents
```