CS 421 — First and Follow Sets

Objectives

The purpose of this handout is to give a formal definition of the first and follow set algorithms, as well as a Haskell implementation. The implementation is described throughout the text as parts of it come up, and the whole of the code is given at the end. You are likely to have better results if you type in the code yourself, rather than downloading it from somewhere or cutting and pasting.

Introduction

As you saw before, a grammar G is a tuple (N, T, P), where N is a set of *non-terminal symbols*, T is a set of *terminal symbols*, and P is a set of *productions* mapping each non-terminal symbol to one or more lists of symbols, possibly including the empty string, which we reperesent with ϵ.

Headers

We need the following imports and language directives.

```haskell
{-# LANGUAGE DeriveGeneric #-}

import qualified Data.HashMap.Strict as H
import qualified Data.HashSet as S

import GHC.Generics (Generic)
import Data.Hashable
```

We are using the efficient hash map and hash set implementation, and to allow the `hash` function to work on our types, we need the generics and hashable modules.

If you are using `stack`, you need to add `hashable` and `unordered-containers` to the cabal file under the `executable` heading. For example, if the project is called `grmsets`, you will have a block like this one:

```
executable grmsets-exe
  hs-source-dirs:      app
  main-is:             Main.hs
  ghc-options:         -threaded -rtsopts -with-rtsopts=-N
  build-depends:       base
                     , grmsets
                     , hashable
                     , unordered-containers
  default-language:    Haskell2010
```

The `build-depends` block is the important one, the other options may be different.

Representations

We use a symbol type with three constructors to represent the three kinds of symbols: terminals, non-terminals, and epsions. We also tell the compiler that we intend to hash this type.

```
data Symbol = Term String
            | NTerm String
            | Eps
     deriving (Show,Eq,Generic)

instance Hashable Symbol
```

A production takes a non-terminal to a list of symbols. A non-terminal can have more than one production.

```
data Prod = Prod Symbol [Symbol]
     deriving (Show,Eq)
```

We use a hash map to represent our sets. Each non-terminal has its own first and follow set.

```
data Prod = Prod Symbol [Symbol]
     deriving (Show,Eq)

type GSet = H.HashMap Symbol (S.HashSet Symbol)
```

The representation of this grammar

$$
\begin{array}{rcl}
S & \rightarrow & A\,\mathtt{x} \\
 & | & B\,\mathtt{y} \\
 & | & \mathtt{z} \\
A & \rightarrow & \mathtt{1}\,C\,B \\
 & | & \mathtt{2}\,B \\
B & \rightarrow & \mathtt{3}\,B \\
 & | & C \\
C & \rightarrow & \mathtt{4} \\
 & | & \epsilon
\end{array}
$$

will be this:

```
gr = [Prod (NTerm "S") [NTerm "A", Term "x"]
     ,Prod (NTerm "S") [NTerm "B", Term "y"]
     ,Prod (NTerm "S") [Term "z"]
     ,Prod (NTerm "A") [Term "1", NTerm "C", NTerm "B"]
     ,Prod (NTerm "A") [Term "2", NTerm "B"]
     ,Prod (NTerm "B") [Term "3", NTerm "B"]
     ,Prod (NTerm "B") [NTerm "C"]
     ,Prod (NTerm "C") [Term "4"]
     ,Prod (NTerm "C") [Eps]
     ]
```

We will frequently want to update entries into our hash table, so we have this utility function to allow that. It takes a function for modifying the entry, a default value in case the entry is not present, a lookup key, and the hash table. It returns a new hash table.

```
updateDefault :: (Eq k, Hashable k) => (v -> v) -> v -> k
                 -> H.HashMap k v -> H.HashMap k v
updateDefault f d k m =
  case H.lookup k m of
    Nothing -> H.insert k (f d) m
    Just v2 -> H.insert k (f v2) m
```

You can simplify this code a bit by using `lookupDefault`.

Here is an example of how it works:

```
Main> updateDefault (+1) 0 "x" H.empty
fromList [("x",1)]
Main> updateDefault (+1) 0 "x" $ H.insert "x" 10 H.empty
fromList [("x",11)]
```

First Sets

Let α be an arbitrary (possibly empty) string of symbols. The first set of a string of symbols α is the set of terminal symbols that could appear in the initial position of a valid parse of α.

For example, if we have a grammar

$$\begin{aligned} S \rightarrow & \quad x \\ | & \quad y\ z \end{aligned}$$

Then we have $first(z) = \{z\}$, $first(S) = \{x, y\}$, and $first(z\ S) = \{z\}$.

To define a first set, let α be an aribrary string of symbols, N be a non-terminal symbol whose first set does not contain ϵ, N' be a non-terminal symbol whose first set contains ϵ, and x be a terminal symbol.

Then the first set of a string of symbols is defined as:

$$\begin{aligned} first(x\ \alpha) &= \{x\} \\ first(N\ \alpha) &= first(N) \\ first(N'\ \alpha) &= (first(N') - \epsilon) \cup first(\alpha) \\ first(\epsilon) &= \{\epsilon\} \end{aligned}$$

In Haskell, we define this as

```
first :: [Symbol] -> GSet -> S.HashSet Symbol
first []  fset = S.singleton Eps
first (Term t : _) fset = S.singleton (Term t)
first (Eps : syms) fset = first syms fset
first (NTerm t : syms) fset =
  let fs_t = H.lookupDefault S.empty (NTerm t) fset
   in if S.member Eps fs_t
         then S.union (S.delete Eps fs_t) (first syms fset)
      else fs_t
```

To calculate the first set of a grammar we iterate over the productions $N \rightarrow \alpha$ and update our definitions $first(N) \leftarrow first(N) \cup first(\alpha)$.

To represent the update of an individual production, we have `updateFirst`:

```
updateFirst :: GSet -> Prod -> GSet
updateFirst fset (Prod nt syms) fset =
   updateDefault (S.union (first syms fset)) S.empty nt fset
```

Given our grammar `gr` above, the first round of updating would yield

$$\begin{aligned} first(S) &= \{z\} \\ first(A) &= \{1, 2\} \\ first(B) &= \{3\} \\ first(C) &= \{4, \epsilon\} \end{aligned}$$

Because the non-terminals do not have anything in their first set definitions yet, we only see the terminal symbols that come at the beginning of a production.

Here is our function `propagateFirst`, which takes a grammar and a first set and updates the first set.

```
propagateFirst :: [Prod] -> GSet -> GSet
propagateFirst grm fset = Prelude.foldl updateFirst fset grm
```

Running it once on an empty hashmap gives us the result we got above.

```
Main> propagateFirst gr H.empty
fromList [(NTerm "A",fromList [Term "2",Term "1"]),
          (NTerm "C",fromList [Term "4",Eps]),
          (NTerm "S",fromList [Term "z"]),
          (NTerm "B",fromList [Term "3"])]
```

If we run this another round, the B and S symbols get updated.

```
Main> propagateFirst gr it
fromList [(NTerm "A",fromList [Term "2",Term "1"]),
          (NTerm "C",fromList [Term "4",Eps]),
          (NTerm "S",fromList [Term "z",Term "3",Term "2",Term "1"]),
          (NTerm "B",fromList [Term "4",Term "3",Eps])]
```

The third time we `propagateFirst` we reach a steady state. Further runs of this code will not change anything.

```
Main> propagateFirst gr it
fromList [(NTerm "A",fromList [Term "2",Term "1"]),
          (NTerm "C",fromList [Term "4",Eps]),
          (NTerm "S",fromList [Term "4",Term "z",Term "3",Term "y",Term "2",Term "1"]),
          (NTerm "B",fromList [Term "4",Term "3",Eps])]
```

Since `propagateFirst` always reaches a steady state, we can define our first set as the fix-point of `propagateFirst`.

```
fix f x = if x == result
             then x
          else fix f result
  where result = f x

firstSet grm = fix (propagateFirst grm) H.empty
```

Now we can calculate our first sets!

```
Main> firstSet gr
fromList [(NTerm "A",fromList [Term "2",Term "1"]),
          (NTerm "C",fromList [Term "4",Eps]),
          (NTerm "S",fromList [Term "4",Term "z",Term "3",Term "y",Term "2",Term "1"]),
          (NTerm "B",fromList [Term "4",Term "3",Eps])]
```

Follow Sets

The follow set of a non-terminal symbol N tells us the terminal symbols that could come *after* a valid parse of N. These are used in the LR parsing algorithm to determine when it is allowable to perform the reduce action. You can think of it as data we need to answer the question "am I done with this symbol?"

Suppose we have a production $N \to \alpha M \beta$, where N and M are non-terminal symbols, and α and β are arbitrary strings of symbols. There are three rules.

- If $\epsilon \notin first(\beta)$, then $follow(M) \subset first(\beta) - \epsilon$, since β follows M. (We never add ϵ to a follow set.)
- If $\epsilon \in first(\beta)$, then $follow(M) \subset follow(N)$, since M appears at the end of an N production.
- Finally, for the start symbol S and end of input symbol \$, we have $\$ \in follow(S)$.

So, for the grammar rule $A \to 1\ C\ B$, we have $follow(C) = first(B) \cup follow(A)$, since $\epsilon \in first(B)$.
As in first sets, we iteratively update our follow set according to these rules.
The first time we apply these rules to our grammar `gr`, we get

$$\begin{aligned} follow(S) &= \{\$\} \\ follow(A) &= \{x\} \\ follow(B) &= \{x, y\} \\ follow(C) &= \{3, 4, x, y\} \end{aligned}$$

It just so happens that this is the complete follow set for this grammar.
Here is the code to perform an update.

```
updateFollow :: GSet -> GSet -> Prod -> GSet
updateFollow fiset foset (Prod nt []) = foset
updateFollow fiset foset (Prod nt (NTerm n : syms)) =
  let fs_n = first syms fiset
      rest = updateFollow fiset foset (Prod nt syms)
   in if S.member Eps fs_n
         then updateDefault
                (\fs -> fs `S.union` H.lookupDefault S.empty nt foset
                          `S.union` (S.delete Eps fs_n))
                S.empty
                (NTerm n)
                rest
         else updateDefault
                (S.union (S.delete Eps fs_n))
                S.empty
                (NTerm n)
                rest
updateFollow fiset foset (Prod nt (_ : syms)) =
   updateFollow fiset foset (Prod nt syms)
```

It works by walking down the right hand side of a production and considering each element. The last clause handles the case that the symbol is a terminal or ϵ; it just considers the rest of the elements. The second clause handles the case that the symbol is a non-terminal. In this case we take the first set of the remaining string and union that with the non-terminal's follow set, also adding the production's follow set if there is an ϵ. We then check the rest of the symbols (this is the purpose of `rest`). We can use the fix-point operator with this to define `followSet`.

Source Code

Here is the complete source code.

```haskell
{-# LANGUAGE DeriveGeneric #-}

module Main where
import qualified Data.HashMap.Strict as H
import qualified Data.HashSet as S

import GHC.Generics (Generic)
import Data.Hashable

-- Add these next three lines if you are using stack
import Lib
main :: IO ()
main = someFunc

-- The Types

data Symbol = Term String
            | NTerm String
            | Eps
     deriving (Show,Eq,Generic)

instance Hashable Symbol

data Prod = Prod Symbol [Symbol]
     deriving (Show,Eq)

type GSet = H.HashMap Symbol (S.HashSet Symbol)

updateDefault :: (Eq k, Hashable k) => (v -> v) -> v -> k -> H.HashMap k v -> H.HashMap k v
updateDefault f d k m =
  case H.lookup k m of
    Nothing -> H.insert k (f d) m
    Just v2 -> H.insert k (f v2) m

gr = [Prod (NTerm "S") [NTerm "A", Term "x"]
     ,Prod (NTerm "S") [NTerm "B", Term "y"]
     ,Prod (NTerm "S") [Term "z"]
     ,Prod (NTerm "A") [Term "1", NTerm "C", NTerm "B"]
     ,Prod (NTerm "A") [Term "2", NTerm "B"]
     ,Prod (NTerm "B") [Term "3", NTerm "B"]
     ,Prod (NTerm "B") [NTerm "C"]
     ,Prod (NTerm "C") [Term "4"]
     ,Prod (NTerm "C") [Eps]
     ]

first :: [Symbol] -> GSet -> S.HashSet Symbol
```

```
first []  fset = S.singleton Eps
first (Term t : _) fset = S.singleton $ Term t
first (Eps : syms) fset = first syms fset
first (NTerm t : syms) fset =
  let fs_t = H.lookupDefault S.empty (NTerm t) fset
   in if S.member Eps fs_t
         then S.union (S.delete Eps fs_t) $ first syms fset
      else fs_t

updateFirst :: GSet -> Prod -> GSet
updateFirst fset (Prod nt syms) =
  updateDefault (S.union (first syms fset)) S.empty nt fset

propagateFirst grm fset = Prelude.foldl updateFirst fset grm

fix f x = if x == result
            then x
          else fix f result
  where result = f x

firstSet grm = fix (propagateFirst grm) H.empty

follow sym foset = H.lookupDefault S.empty sym foset

updateFollow :: GSet -> GSet -> Prod -> GSet
updateFollow fiset foset (Prod nt []) = foset
updateFollow fiset foset (Prod nt (NTerm n : syms)) =
  let fs_n = first syms fiset
      rest = updateFollow fiset foset (Prod nt syms)
   in if S.member Eps fs_n
         then updateDefault
                (\fs -> fs `S.union` H.lookupDefault S.empty nt foset
                           `S.union` (S.delete Eps fs_n))
                S.empty
                (NTerm n)
                rest
         else updateDefault
                (S.union (S.delete Eps fs_n))
                S.empty
                (NTerm n)
                rest
updateFollow fiset foset (Prod nt (_ : syms)) =
   updateFollow fiset foset (Prod nt syms)

propagateFollow fiset grm foset =
  Prelude.foldl (updateFollow fiset) foset grm

followSet grm = fix (propagateFollow (firstSet grm) grm) H.empty
```