Parser MP

CS 421

Revision 1.0

Assigned Friday, April 7, 2017
Due Monday, April 17, 2017

Objectives and Background

For this MP you will write a program that will take a grammar as input and determine whether or not the grammar is LL.

When you are finished with this MP, you will have accomplished two things:

1. You will have written a parser using HASKELL's `parsec` parser combinator library.
2. You will have written code to analyze a grammar.

The Types

There are three user-defined types you will use:

- `Symbol` — Encodes the symbols of the language, include epsilon (ϵ).

```haskell
data Symbol = Symbol String
            | Epsilon
   deriving (Eq,Generic)
```

- `ProductionSet` — Groups all the productions belonging to a specific nonterminal. We store the nonterminal as a string and the right hand side of the productions as a list of list.

```haskell
data ProductionSet = ProductionSet String [[Symbol]]
   deriving Eq
```

So, the grammar

$$\begin{array}{rcl} S \rightarrow & x\,A\,b \\ | & q \\ | & \epsilon \end{array}$$

would be encoded as

```haskell
ProductionSet "S" [[Symbol "x", Symbol "A", Symbol "b"]
                  ,[Symbol "q"]
                  ,[Epsilon]]
```

- `Grammar` — Encodes the grammar with three fields. The first is the list of production sets, assuming the first encodes the start symbol. The second field is the set of nonterminal symbols, and the thirds is the set of terminal symbols.

```haskell
data Grammar = Grammar [ProductionSet] (S.HashSet Symbol) (S.HashSet Symbol)
```

The instance `show` has been custom defined for these.

Part 1 --- The Parser

The first part of this MP is to write the parser. We have provided a pretty type `Parser` and a utility function `run`.

```haskell
type Parser = ParsecT String () Identity

run :: Parser a -> String -> a
run p s =
    case parse p "<stdin>" s of
        Right x -> x
        Left x  -> error (show x)
```

The grammar

Here is the grammar you need to parse:

$$\begin{aligned} \mathit{Grammar} ::= &\ \mathit{ProductionSet} \\ \mid &\ \mathit{ProductionSet}\ \mathit{Grammar} \\ \mathit{ProductionSet} ::= &\ \mathit{Symbol}\ \rightarrow\ \mathit{Production}\ \mathit{Newline}\ [\mathit{Bar}\ \mathit{Production}\ ...] \\ \mathit{Production} ::= &\ [\mathit{Symbol}...]\ \mathit{Newline} \\ \mid &\ \mathit{Epsilon}\ \mathit{Newline} \end{aligned}$$

The notation `Newline` means to match a literal end of line, and the `[Foo...]` notation means to match zero or more `Foo`'s. An epsilon, if it appears, can only appear on a line by itself. We will match the stings `"eps"` and `"ϵ"` as an epsilon.

To get you started, we provide three parsers, `inlinews :: Parser String` that reads non-newline whitespace, `stringws :: String -> Parser String` that matches a given string and consumes any trailing non-newline whitespace, and `ident :: Parser String` that reads a symbol that is not an epsilon. We give you the last one as an example of how you might need to use the `try` combinator later on.[1]

```
-- inlinews - parses spaces and tabs but not newlines
inlinews :: Parser String
inlinews = many (oneOf " \t") <?> "whitespace"

-- stringws - parses a string and consumes trailing whitespace
stringws :: String -> Parser String
stringws s = do _ <- string s
                _ <- inlinews
                return s

-- ident - parse a non-epsilon identifier, at least one upper or lowercase letter
ident :: Parser String
ident = do i <- try (do ii <- many1 (oneOf (['a'..'z'] ++ ['A'..'Z'])) <?> "an identifier"
                        if ii == "eps" then fail "eps not expected" else return ii)
           _ <- inlinews
           return i
```

We also give you the "other end" of the parser, `grammar :: Parser Grammar`:

```
grammar :: Parser Grammar
grammar = do p <- many1 productionSet
             return (Grammar p (terminals p) (nonTerminals p))
```

Here are the descriptions of the remaining parsers you will need to write.

- `realIdent :: Parser Symbol` — This is just like `ident`, but returns a `Symbol` instead of a string.
- `epsilon :: Parser Symbol` — Parse a single epsilon and the trailing non-newline whitespace.
- `epsilonLine :: Parser [Symbol]` — Parse an epsilon and a trailing newline.
- `tokenLine :: Parser [Symbol]` — Parse a line of non-epsilon tokens (like would come after the `->` or `|` in the grammar.) Also consumes the newline.
- `initialProduction :: Parser (String,[Symbol])` — Parse the first line in a grammar specification. In the grammar

$$\begin{aligned} S \rightarrow &\ x\ A\ b \\ \mid &\ q \\ \mid &\ \epsilon \end{aligned}$$

  the `initialProduction` is the $S \rightarrow x\ A\ b$ part.
- `continueProduction :: Parser [Symbol]` — Parse a continuation production. These begin with a bar (possibly with some leading whitespace). In the above grammar, lines 2 and 3 each are continuation productions.
- `productionSet :: Parser ProductionSet` — Parse a production set. The grammar above is a single production set.

[1] The `try` combinator allows you to "unconsume" input and backtrack when necessary.

LL Analyzer

The analysis part comes next. The first three functions take a list of productions and determine the class of symbols in it. A symbol is considered a non-terminal if it appears on the left hand side of any production. Otherwise we consider it a terminal symbol. The function `symbols` returns the intersection.

These functions are needed by the parser to produce the final grammar object, so these return an empty hash set so that you don't have to get them working before you test your parser.

- `nonTerminals :: [ProductionSet] -> S.HashSet Symbol`
- `symbols :: [ProductionSet] -> S.HashSet Symbol`
- `terminals :: [ProductionSet] -> S.HashSet Symbol`

In order to detect if a grammar is LL or not, you will need to find the First set of all the nonterminal symbols in the grammar. You will need two functions for this[2].

- `getFirstSet :: Grammar -> H.HashMap Symbol (S.HashSet Symbol)` This is the top-level function. You input your grammar and get a hashmap from symbols to corresponding first sets.
- `first :: S.HashSet Symbol -> [Symbol] -> S.HashSet Symbol` This function takes a first set for a grammar and a list of tokens (as you would find on the right hand side of a production) and returns the corresponding set of symbols that are the first set for that line. Typically `getFirstSet` will call this repeatedly as it generates a grammar's first set.

For this MP, we are going to have you return "augmented" first sets. That is, the first set will contain both nonterminal symbols *and* terminal symbols. The reason for including the nonterminal symbols is that makes it much easier to detect left recursion and mutual left recursion.

For example, consider this grammar:

$$\begin{aligned} S \to &\quad A\ b \\ \mid &\quad q \\ A \to &\quad x\ y \\ \mid &\quad \epsilon \end{aligned}$$

Then `getFirstSet` will return $\{S \mapsto \{A, x, b, q\}, A \mapsto \{x, \epsilon\}\}$.
If we apply the function `first` on that and the line `A b` we will get back the set $\{A, x, b\}$.
Finally, you need to write the function

- `isLL :: Grammar -> Bool` that is true when a grammar is LL. You are encouraged to write helper functions to check for specific things, such as `isLeftRecursive` or `hasCommonPrefix` if it makes life easier for you.

Tests

We will provide test cases by Wednesday of next week.

[2] You are always welcome to write your own helper functions if you want.