Deductive Parsing in Haskell

Jan van Eijck
CWI and ILLC, Amsterdam, Uil-OTS, Utrecht

February 23, 2004

Abstract

This paper contains the full code of an implementation in Haskell [2], in 'literate programming' style [3], of an approach to deductive parsing based on [4]. We focus on the case of the Earley [1] parsing algorithm for CF languages.

Keywords: Deductive parsing, context free grammars, Earley parsing algorithm, Haskell, literate programming.

1 General Data Structures

```
module DP where

import List
```

Terminal and nonterminal symbols:

```
data Symbol a b = T a | N b | D b deriving (Eq,Ord,Read)
```

The `D` nonterminal is useful for extending a grammar with a new start symbol.

Given show functions for the types `a` and `b`, we define a show function for `Symbol a b` as follows:

```
instance (Show a, Show b) => Show (Symbol a b) where
 show (T x)  = show x
 show (N x)  = show x
 show (D x)  = show x ++ "'"
```

The property of being a nonterminal:

```
nonterm :: Symbol a b -> Bool
nonterm (T _)  = False
nonterm  _     = True
```

The property of being a dummy symbol

```
dummy :: Symbol a b -> Bool
dummy (D _) = True
dummy  _    = False
```

Grammar rules:

```
data Rule a b = Rule (Symbol a b) [Symbol a b] deriving Eq
```

A show function for grammar rules.

```
instance (Show a, Show b) => Show (Rule a b) where
  show (Rule y zs) = show y ++ "==>" ++ show zs
```

Reading a grammar rule:

```
instance (Read a, Read b) => Read (Rule a b) where
  readsPrec p = \ r ->
        [ (Rule symbol rhs,u) | (symbol,s) <- reads r,
                                ("==>", t) <- lex   s,
                                (rhs,   u) <- reads t    ]
```

Example:

```
DP> read "N 'S' ==> [T 'a', N 'S', T 'a']" :: Rule Char Char
'S'==>['a','S','a']
```

Functions for accessing the left- and righthand sides of a rule.

```
lhs :: Rule a b -> (Symbol a b)
lhs (Rule x ys) = x

rhs :: Rule a b -> [Symbol a b]
rhs (Rule x ys) = ys
```

A grammar is a list of rules.

```
type Grammar a b = [Rule a b]
```

When reading in a grammar we adopt the convention that the lefthandside symbol of the first grammar rule is the start symbol.

```
startSymbol :: Grammar a b -> Symbol a b
startSymbol []     = error "empty grammar"
startSymbol (r:rs) = lhs r
```

Converting a list of strings to a grammar:

```
readGrammar :: (Read a, Read b) => [String] -> (Grammar a b)
readGrammar =
   map (read ::  (Read a, Read b) => String -> Rule a b)
```

Finally, a function for reading a grammar from a file.

```
getGrammar :: (Read a, Read b) => FilePath -> IO (Grammar a b)
getGrammar filename = do
      str <- readFile filename
      return (readGrammar (lines str))
```

2 Example Grammars

For concreteness sake, let us assume that terminal and nonterminal symbols are of type `Char`. Here is an example grammar, read in from file `grammar0` (it is assumed that the file `grammar0` is in the current directory):

```
DP> readFile "grammar0"
"N 'S' ==> [T 'a', N 'S', T 'b']\nN 'S' ==> [T 'a', T 'b']\n"
DP> getGrammar "grammar0" :: IO (Grammar Char Char)
['S'==>['a','S','b'],'S'==>['a','b']]
```

Here is another example grammar.

```
grammar1 :: Grammar Char Char
grammar1 = [Rule (N 'S') [T 'a', N 'S', T 'a'],
            Rule (N 'S') [T 'b', N 'S', T 'b'],
            Rule (N 'S') [T 'a'],
            Rule (N 'S') [T 'b']             ]
```

An example of a grammar with an epsilon rule:

```
grammar2 :: Grammar Char Char
grammar2 = [Rule (N 'S') [T 'a', N 'S', T 'a'],
            Rule (N 'S') [T 'b', N 'S', T 'b'],
            Rule (N 'S') [T 'a'],
            Rule (N 'S') [T 'b'],
            Rule (N 'S') []                    ]
```

A grammar for balanced parentheses:

```
grammar3 :: Grammar Char Char
grammar3 = [Rule (N 'S') [T '(', N 'S', T ')', N 'S'],
            Rule (N 'S') []                           ]
```

Here is a data type for derivation trees:

```
data Tree a b = Leaf a | Node b [Tree a b] deriving (Eq,Ord,Show)
```

An example derivation tree:

```
tree0 = Node 'S' [Leaf 'a', Leaf 'b']
```

Displaying a tree on the screen:

```
displayTree :: (Show a, Show b) => Tree a b -> IO()
displayTree tr = mapM_ putStrLn (showTree 0 tr) where
   showTree :: (Show a, Show b) => Int -> Tree a b -> [String]
   showTree i (Leaf x)    = [(map (\ _ -> ' ') [1..i]) ++ show x]
   showTree i (Node x ts) = ((map (\ _ -> ' ') [1..i]) ++ show x) :
      concat (map (showTree (i+5)) ts)
```

The example tree gets displayed as follows:

```
DP> displayTree tree0
'S'
     'a'
     'b'
```

3 Earley Items, Axioms, Goals, Consequences

Earley items are of the form $i, A \rightarrow \alpha \bullet \beta, j$. They consist of a rule $A \rightarrow \alpha\beta$ with a $\bullet$ in its righthand side to indicate the part of the righthand side that was recognized so far, a pointer i to the parent node where the rule was invoked, and a pointer j to the position in the input that recognition has reached.

For good measure, we also include a derivation tree component, by putting a list of derivation trees as the last component of an Earley item.

```
data Item a b =
  Item Int (Symbol a b) [Symbol a b] [Symbol a b] Int [Tree a b]
  deriving (Eq,Ord)
```

A show function for items, using * for the dot, and suppressing the derivation tree component.

```
instance (Show a, Show b) => Show (Item a b) where
  show (Item i b symbols symbols' j ts) =
   "(" ++ show i ++ "," ++
   show b ++ "==>" ++ show symbols ++ "*" ++ show symbols'
   ++ "," ++ show j  ++ ")"
```

A function for extracting the list of derivation trees from an Earley item:

```
getTrees :: Item a b -> [Tree a b]
getTrees (Item i b symbols symbols' j ts) = ts
```

In the case of Earley parsing, there is one axiom. It has the form $0, S' \rightarrow \bullet S, 0$, where S is the start symbol of the grammar and S' is a new start symbol.

```
axioms :: Grammar a b -> [Item a b]
axioms grammar = [Item 0 (D x) [] [N x] 0 []]
  where
    N x = startSymbol grammar
```

In the case of Earley parsing, there is one goal. It has the form $0, S' \rightarrow S\bullet, n$, where S is the start symbol of the grammar, S' is the new start symbol used in the axiom, and n is the length of the input. Here is a function for recognizing goals:

```
goal :: (Eq a, Eq b) => Grammar a b -> [a] -> Item a b -> Bool
goal grammar tokens (Item i symbol symbols symbols' k trees) =
  i == 0
  &&
  dummy symbol
  &&
  symbols == [startSymbol grammar]
  &&
  symbols' == []
  &&
  k == length tokens
```

In the case of Earley parsing, there are three kinds of consequences, for scanning, prediction and completion.

```
consequences ::  (Eq a,Eq b) =>
   Grammar a b -> [a] -> Item a b -> [Item a b] -> [Item a b]
consequences grammar tokens trigger stored =
   scan tokens trigger
   ++
   predict grammar trigger
   ++
   complete trigger stored
```

The scanning rule for Earley parsing is the rule that shifts the bullet across a terminal. It has the form (derivation tree component omitted):

$$\frac{i, A \rightarrow \alpha \bullet w_j \beta, j}{i, A \rightarrow \alpha w_j \bullet \beta, j+1}$$

```
scan :: (Eq a,Eq b) => [a] -> Item a b -> [Item a b]
scan tokens (Item i a alpha [] j ts)            = []
scan tokens (Item i a alpha (symbol:beta) j ts)
   | j >= length tokens = []
   | otherwise          =
      [ Item i a (alpha ++ [symbol]) beta (j+1) (ts ++ [Leaf (tokens !! j)])
            | symbol == (T (tokens !! j))            ]
```

The prediction rule for Earley parsing is the rule that initializes a new rule $B \rightarrow \gamma$ on the basis of a premisse indicating that B is expected at the current point in the input. It has the form (derivation tree component omitted):

$$\frac{i, A \rightarrow \alpha \bullet B \beta, j}{j, B \rightarrow \bullet \gamma, j} B \rightarrow \gamma$$

```
predict :: (Eq a,Eq b) => Grammar a b -> Item a b -> [Item a b]
predict grammar (Item i a alpha [] j ts)            = []
predict grammar (Item i a alpha (symbol:beta) j ts) =
  [ Item j b [] gamma j [] | (Rule b gamma) <- grammar,
                             symbol == b                ]
```

The completion rule for Earley parsing is the rule that shifts the bullet across a non-terminal. It has two premisses, and it is of the following form (derivation tree component omitted):

$$\frac{i, A \to \alpha \bullet B\beta, k \quad k, B \to \gamma\bullet, j}{i, A \to \alpha B \bullet \beta, j}$$

In the implementation this is handled by distinguishing three cases:

- Trigger of the form $i, A \to \alpha \bullet B\beta, k$: look for other premisse on the chart.
- Trigger of the form $k, B \to \gamma\bullet, j$: look for other premisse on the chart.
- Otherwise, i.e., trigger s of the form $i, A \to \alpha \bullet w\beta, k$, with w a terminal: rule does not apply.

The third case uses a catch-all pattern.

```
complete :: (Eq a, Eq b) => Item a b -> [Item a b] -> [Item a b]
complete (Item i a alpha (N x:beta) k ts) stored =
  [ Item i a (alpha++[N x]) beta j (ts ++ [Node x ts'])|
         (Item k' symbol gamma [] j ts') <- stored,
          k == k',
          symbol == N x                                    ]
complete (Item k (N x) gamma [] j ts) stored =
  [ Item i a (alpha++[N x]) beta j (ts' ++ [Node x ts])|
         (Item i a alpha (symbol:beta) k' ts') <- stored,
          k == k',
          symbol == N x                                    ]
complete item stored = []
```

This completes the Earley-specific part of the story.

4 Chart and Agenda

A chart plus agenda is a pair of item lists. Call this datatype a store.

```
type Store a b = ([Item a b],[Item a b])
```

The idea is to use the agenda for those items that have been proved, but whose direct consequences have not yet been derived, and the chart for the proved items the consequences of which have also been computed.

We start out with an empty chart and with a list of all axioms on the agenda.

```
initStore :: (Eq a, Eq b) => Grammar a b -> [a] -> Store a b
initStore grammar tokens = ([], axioms grammar)
```

Next, we tackle the items on the agenda one by one:

- add their consequences to the agenda.
- move them from the agenda to the chart (as their consequences have been computed).

```
exhaustAgenda :: (Eq a, Eq b) =>
      Grammar a b -> [a] -> Store a b -> Store a b
exhaustAgenda grammar tokens (chart,[]) = (chart,[])
exhaustAgenda grammar tokens (chart,agenda@(trigger:rest)) =
  exhaustAgenda grammar tokens (newchart,newagenda)
  where
    newchart = chart ++ [trigger]
    conseq   = consequences grammar tokens trigger chart
    new      = conseq \\ (chart ++ agenda)
    newagenda = rest ++ new
```

Check whether a goal item has been found, and return the list of goal items.

```
goalFound :: (Eq a, Eq b) => Grammar a b -> [a] -> [Item a b] -> [Item a b]
goalFound grammar tokens store = filter gl store
  where gl = goal grammar tokens
```

If a parse is successful, it is nice to display the chart:

```
display :: Show a => [a] -> IO()
display [] = return ()
display (x:xs) = do print x
                    display xs
```

Rather than displaying the whole chart, we will display only the records of the nodes that have been successfully created. To that end, we prune the chart using the following filter:

```
pruned :: (Eq a, Eq b) => [Item a b] -> [Item a b]
pruned = filter (\ (Item i s symbols symbols' j ts) -> symbols' == [])
```

Parsing is now a matter of initializing the store, exhausting the agenda, and checking whether a goal item has been found in the chart. As output we allow either a parsetree or a chart, depending on a boolean trigger.

```
data OutputKind = Tree | Chart deriving Eq
```

```
parse :: (Eq a, Show a, Eq b, Show b) =>
   Grammar a b -> [a] -> OutputKind -> IO()
parse grammar tokens output =
  if goals /= []
     then if output == Tree
             then displayTree ptree
             else display (pruned chart)
     else putStrLn "no parse"
  where
    goals  = goalFound grammar tokens chart
    ptree  = head (getTrees (head goals))
    init   = initStore grammar tokens
    result = exhaustAgenda grammar tokens init
    chart  = fst result
```

5 Trying it out

```
DP> parse grammar1 "abba" Chart
no parse

DP> parse grammar1 "aba" Chart
(0,'S'==>['a']*[],1)
(0,'S''==>['S']*[],1)
```

```
(1,'S'==>['b']*[],2)
(2,'S'==>['a']*[],3)
(0,'S'==>['a','S','a']*[],3)
(0,'S''==>['S']*[],3)

DP> parse grammar1 "aba" Tree
'S'
     'a'
     'S'
          'b'
     'a'

DP> parse grammar2 "abba" Chart
(0,'S'==>[]*[],0)
(0,'S'==>['a']*[],1)
(0,'S''==>['S']*[],0)
(1,'S'==>[]*[],1)
(0,'S''==>['S']*[],1)
(1,'S'==>['b']*[],2)
(2,'S'==>[]*[],2)
(2,'S'==>['b']*[],3)
(3,'S'==>[]*[],3)
(1,'S'==>['b','S','b']*[],3)
(3,'S'==>['a']*[],4)
(4,'S'==>[]*[],4)
(0,'S'==>['a','S','a']*[],4)
(0,'S''==>['S']*[],4)

DP> parse grammar2 "abba" Tree
'S'
     'a'
     'S'
          'b'
          'S'
          'b'
     'a'

DP>  parse grammar3 "(()())" Chart
(0,'S'==>[]*[],0)
(0,'S''==>['S']*[],0)
(1,'S'==>[]*[],1)
(2,'S'==>[]*[],2)
(3,'S'==>[]*[],3)
(1,'S'==>['(','S',')','S']*[],3)
(4,'S'==>[]*[],4)
(5,'S'==>[]*[],5)
```

```
(3,'S'==>['(','S',')','S']*[],5)
(1,'S'==>['(','S',')','S']*[],5)
(6,'S'==>[]*[],6)
(0,'S'==>['(','S',')','S']*[],6)
(0,'S''==>['S']*[],6)

DP>  parse grammar3 "(()())" Tree
'S'
     '('
     'S'
          '('
          'S'
          ')'
          'S'
               '('
               'S'
               ')'
               'S'
     ')'
     'S'
```

6 Function for Stand-alone Use

```
module Main

where

import System
import DP
```

If the program is executed as a script (or as a compiled binary), the following function gets executed.

```
main :: IO()
main = do args <- getArgs
          if length args /= 2
             then putStrLn "Usage: dp <grammar> <string>."
             else let [name,tokens] = args in
               do grammar  <- getGrammar name :: IO (Grammar Char Char)
                  parse grammar tokens Tree
                  return ()
```

References

[1] Earley, J. An efficient context-free parsing algorithm. *Communications of the ACM 13* (1970), 94–102.

[2] Jones, S. P., Hughes, J., et al. Report on the programming language Haskell 98. Available from the Haskell homepage: `http://www.haskell.org`, 1999.

[3] Knuth, D. *Literate Programming.* CSLI Lecture Notes, no. 27. CSLI, Stanford, 1992.

[4] Shieber, S., Schabes, Y., and Pereira, F. Principles and implementation of deductive parsing. *Journal of Logic Programming 24* (1995), 3–36.