CS 3EA3: Example Haskell Code for Recursive Descent Parsing

Wolfram Kahl

September 17, 2009

The module *Data.Char* provides us with the functions *isDigit*, *isLetter*, and $ord :: Char \to Int$.

module *ExprParse* **where**
import *Data.Char*

An extremely simple expression datatype (abstract syntax):

```haskell
data Expr
  = Var String
  | Num Integer
  | Add Expr Expr
  | Mul Expr Expr
```

For the concrete syntax, we encode the different precedence levels into separate nonterminals to achieve a context-free grammar that allows recursive descent parsing:

```
Expr    ::= Term + Expr     |   Term
Term    ::= Factor * Term   |   Factor
Factor  ::= Number          |   Identifier   |    ( Expr )
```

The precedence levels can easil be accommodated by the unparsing function[1]; whether parentheses are added is decided depending on the precedence argument, which stands for the precedence of the *surrounding* constructor. Adding parentheses uses an auxiliary function *paren*:

```haskell
paren :: String -> String
paren s = '(' : s ++ ")"
```

```haskell
showPrecExpr :: Int -> Expr -> String
showPrecExpr p (Var s) = s
showPrecExpr p (Num k) = show k
showPrecExpr p (Add e1 e2) = (if p > 4 then paren else id)
  (showPrecExpr 4 e1 ++ " + " ++ showPrecExpr 4 e2)
```

[1] Idiomatic Haskell would define the more efficient $showsPrecExpr :: Int \to Expr \to String \to String$.

$$showPrecExpr\ p\ (Mul\ e1\ e2) = (\textbf{if}\ p > 5\ \textbf{then}\ paren\ \textbf{else}\ id)$$
$$\quad (showPrecExpr\ 5\ e1 ++ \texttt{" * "} ++ showPrecExpr\ 5\ e2)$$

For installing this as standard *show* function for *Expr* values, we choose lowest precedence by default:

$$\textbf{instance}\ Show\ Expr\ \textbf{where}\ show = showPrecExpr\ 0$$

For recursive descent parsing see also `http://en.wikipedia.org/wiki/Recursive_descent`; for an extremely "low-tech" Haskell version we use a parser type that takes a *String* as argument, and returns a pair consisting of a parsed *Expr*, and the not-yet-parsed rest of the *String*:

$$parseExpr, parseTerm, parseFactor :: String \rightarrow (Expr, String)$$

This is a very simple choice; it forces us to use Haskell run-time errors for reacting to parse errors.

The grammar rules can now straight-forwardly be translated into Haskell, always continuing to work on the rest strings:

$$\begin{array}{l}
parseExpr\ cs = \textbf{let}\ (t, cs') = parseTerm\ cs\ \textbf{in case}\ cs'\ \textbf{of} \\
\quad \texttt{'+'} : cs'' \rightarrow \textbf{let}\ (e, cs''') = parseExpr\ cs''\ \textbf{in}\ (Add\ t\ e, cs''') \\
\quad _ \rightarrow (t, cs')
\end{array}$$

$$\begin{array}{l}
parseTerm\ cs = \textbf{let}\ (f, cs') = parseFactor\ cs\ \textbf{in case}\ cs'\ \textbf{of} \\
\quad \texttt{'*'} : cs'' \rightarrow \textbf{let}\ (t, cs''') = parseTerm\ cs''\ \textbf{in}\ (Mul\ f\ t, cs''') \\
\quad _ \rightarrow (f, cs')
\end{array}$$

$$\begin{array}{l}
parseFactor\ (\texttt{'('} : cs) = \textbf{let}\ (e, cs') = parseExpr\ cs\ \textbf{in case}\ cs'\ \textbf{of} \\
\quad \texttt{')'} : cs'' \rightarrow (e, cs'') \\
\quad _ \rightarrow error\ \texttt{"closing parenthesis expected"} \\
parseFactor\ (c : cs) \\
\quad \mid isLetter\ c\ \ = \textbf{let}\ (ident, cs') = parseIdentRest\ [c]\ cs \\
\qquad\qquad\qquad\quad \textbf{in}\ (Var\ ident, cs') \\
\quad \mid isDigit\ c\ \ = \textbf{let}\ (num, cs') = parseNumberRest\ (numFromDigit\ c)\ cs \\
\qquad\qquad\qquad\quad \textbf{in}\ (Num\ num, cs') \\
parseFactor\ cs = error\ (\texttt{"factor expected; "} ++ show\ cs ++ \texttt{" found"})
\end{array}$$

Since we allow multi-letter dentifiers and multi-digit numbers, we use auxiliary parser to complete them after we recognised the first character:

$$\begin{array}{l}
parseIdentRest :: String \rightarrow String \rightarrow (String, String) \\
parseIdentRest\ s\ (c : cs) \mid isLetter\ c = parseIdentRest\ (s ++ [c])\ cs \\
parseIdentRest\ s\ cs = (s, cs)
\end{array}$$

$$\begin{array}{l}
parseNumberRest :: Integer \rightarrow String \rightarrow (Integer, String) \\
parseNumberRest\ n\ (c : cs) \\
\quad \mid isDigit\ c = parseNumberRest\ (10 * n + numFromDigit\ c)\ cs \\
parseNumberRest\ n\ cs = (n, cs)
\end{array}$$

$$\begin{array}{l}
numFromDigit :: Char \rightarrow Integer \\
numFromDigit\ c = toInteger\ (ord\ c - ord\ \texttt{'0'})
\end{array}$$