CITS3211
FUNCTIONAL PROGRAMMING

12. Parsing

Summary: This lecture discusses the programming of parsers in Haskell, using a special library based on a monad. (It also serves as an example of how FP allows other languages to be embedded.)

These notes based on a lecture by Graham Hutton.

Where Are They Used?

Many real life programs use some form of parser to pre-process their inputs.

<table>
<thead>
<tr>
<th>Hugs</th>
<th>parses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unix</td>
<td>Haskell programs</td>
</tr>
<tr>
<td>Explorer</td>
<td>Shell scripts</td>
</tr>
<tr>
<td></td>
<td>HTML documents</td>
</tr>
</tbody>
</table>

Functional Parsers

In a functional language such as Haskell, parsers can naturally be viewed as functions.

type Parser = String -> Tree

data Tree = Num Int
            | Plus Tree Tree
            | Times Tree Tree

A parser is a function that takes a string and returns some form of tree.
However, a parser might not require all of its input string, so we also return any unused input:

\[
\text{type Parser} = \text{String} \to (\text{Tree, String})
\]

A string might be parsable in many ways, including none, so we generalize to a list of results:

\[
\text{type Parser} = \text{String} \to [(\text{Tree, String})]
\]

Finally, a parser might not always produce a tree, so we generalize to a value of any type:

\[
\text{type Parser } a = \text{String} \to [(a, \text{String})]
\]

Note:

- For simplicity, we will focus on parsers that either fail and return the empty list of results, or succeed and return a singleton list.

## Primitive Parsers

- The parser `item` fails if the input is empty, and consumes the first character otherwise:

\[
\begin{align*}
\text{item :: Parser Char} \\
\text{item } [] &= [] \\
\text{item } (x:xs) &= [(x, xs)]
\end{align*}
\]

- The parser `mzero` always fails:

\[
\begin{align*}
\text{mzero :: Parser } a \\
\text{mzero } \text{inp} &= []
\end{align*}
\]

- The parser `return v` always succeeds, returning the value `v` without consuming any input:

\[
\begin{align*}
\text{return :: } a \to \text{Parser } a \\
\text{return } v \text{ inp} &= [(v, \text{inp})]
\end{align*}
\]
The parser \( p +++ q \) behaves as the parser \( p \) if it succeeds, and as the parser \( q \) otherwise:

\[
(++) :: \text{Parser} \ a \rightarrow \text{Parser} \ a \rightarrow \text{Parser} \\
(++) \ p \ q = \text{\textbackslash inp} \rightarrow \text{\textbackslash take 1 (p \ inp \ ++ \ q \ inp)}
\]

The function \texttt{papply} applies a parser to a string:

\[
papply :: \text{Parser} \ a \rightarrow \text{String} \rightarrow ([a,\text{String}]) \\
papply \ p \ inp = p \ inp
\]

\[
> \ \text{papply \ mzero} \ "hello" \\
\quad []
\]

\[
> \ \text{papply (return 1)} \ "hello" \\
\quad [(1,"hello")]
\]

\[
> \ \text{papply (item ++ return 'a')} \ "hello" \\
\quad [('h','ello')]
\]

\[
> \ \text{papply (mzero ++ return 'a')} \ "hello" \\
\quad [('a','hello')]
\]

\section*{Examples}

The behavior of the five parsing primitives can be illustrated with some simple examples:

\[
\% \text{hugs ParseLib} \\
> \ \text{papply \ item} \ "" \\
\quad []
\]

\[
> \ \text{papply \ item} \ "hello" \\
\quad [("h","ello")]
\]

\section*{Note:}

\begin{itemize}
  \item The library file \texttt{ParseLib.hs} comes with the Hugs system, but is not a standard Haskell library.
  \item IO and Parser are both monads, a mathematical structure that has proved useful for modeling many situations involving sequences.
  \item In Haskell, a monad is a type constructor that has the following functions for constructing sequences (which must satisfy some laws).
\end{itemize}

\[
\text{return} :: a \rightarrow \text{Parser} \ a \\
\text{>>=} :: \text{Parser} \ a \rightarrow (a \rightarrow \text{Parser} \ b) \rightarrow \text{Parser} \ b
\]
Sequencing Parsers

A sequence of parsers can be combined as a single composite parser using the keyword do.

For example:

```haskell
parseTwo :: Parser (Char, Char)
parseTwo = do x ← item
            y ← item
            return (x, y)
```

If any parser in a sequence of parsers fails, then the sequence as a whole fails. For example:

```haskell
> papply parseTwo "h"
[]
```

The do notation is not specific to IO and Parser, but can be used with any monad.

```haskell
>>= :: Parser a → (a → Parser b) → Parser b
p >>= pf = p2
where p2 inp = concat [pf ret rest |
            (ret, rest) ← p inp]
```

Other Library Parsers

- Parsing a character that satisfies a condition:

```haskell
sat :: (Char → Bool) → Parser Char
sat p = do c ← item
            if p c then
                return c
            else
                mzero
```

- Parsing a digit and specific characters:

```haskell
digit :: Parser Char
digit = sat isDigit

char :: Char → Parser Char
char c = sat (c ==)
```

- Applying a parser zero or more times:

```haskell
many :: Parser a → Parser [a]
many p = many1 p +++ return []
```
Applying a parser one or more times:

```haskell
many1 :: Parser a -> Parser [a]
many1 p = do x ← p
           xs ← many p
           return (x:xs)
```

Parsing a specific string of characters:

```haskell
string :: String -> Parser String
string [] = return []
string (c:cs) = do char c
                 string cs
                 return (c:cs)
```

Example

We can now define a parser that consumes a list of one or more digits from a string:

```haskell
digits :: Parser String
digits = do char '
           d ← digit
           ds ← many (do char ','
                        digit)
           char ']
           return (d:ds)
```

Arithmetic Expressions

Consider a simple form of expressions built up from single digits using the arithmetic operators + and *, and using parentheses, such that:

- The operators associate to the right;
- * has higher priority than +.

For example:

```haskell
> papply digits "[1,2,3,4]"
["1234","""]

> papply digits "[1,2,3,4"
[]
```

Note:

- More sophisticated parsing libraries can indicate and/or recover from errors in the input string.
Formally, the syntax of such expressions is defined by the following context free grammar:

\[
\begin{align*}
expr & \rightarrow \text{term } + \text{ expr} \\
\quad & \mid \text{ term} \\
\text{term} & \rightarrow \text{ factor } \ast \text{ term} \\
\quad & \mid \text{ factor} \\
\text{factor} & \rightarrow \text{ digit} \mid (\text{ expr } )
\end{align*}
\]

However, for reasons of efficiency, it is important to factorise the rules for expr and term:

\[
\begin{align*}
expr & \rightarrow \text{ term } (\text{ ' + ' expr} \\
\quad & \mid \varepsilon ) \\
\text{term} & \rightarrow \text{ factor } (\text{ ' * ' term} \\
\quad & \mid \varepsilon ) \\
\text{factor} & \rightarrow \text{ digit } \mid (\text{ ' expr } )'
\end{align*}
\]

Finally, we turn the grammar into a functional parser that generates the tree for an expression:

\[
\begin{align*}
\text{expr} & :: \text{ Parser Tree} \\
\text{expr} & = \text{ do } t \leftarrow \text{ term} \\
\quad & \text{ do char ' + ' } \\
\quad & \quad e \leftarrow \text{ expr} \\
\quad & \quad \text{ return (Plus t e) } \\
\quad & \text{ return } t \\
\text{term} & :: \text{ Parser Tree} \\
\text{term} & = \text{ do } f \leftarrow \text{ factor} \\
\quad & \text{ do char ' * ' } \\
\quad & \quad t \leftarrow \text{ term} \\
\quad & \quad \text{ return (Times f t) } \\
\quad & \text{ return } f \\
\text{factor} & :: \text{ Parser Tree} \\
\text{factor} & = \text{ do } d \leftarrow \text{ digit} \\
\quad & \text{ return (Num (digitToInt d))} \\
\quad & \text{ do char '(' } \\
\quad & \quad e \leftarrow \text{ expr} \\
\quad & \quad \text{ char ')'} \\
\quad & \text{ return e}
\end{align*}
\]
Then we can test the parser under hugs:

```haskell
> papply expr "1+2*3"
[Plus (Num 1) (Times (Num 2) (Num 3))
 ,"")]

> papply expr "(1+2)*3"
[Times (Plus (Num 1) (Num 2)) (Num 3)
 ,"]}]}