CSCE 314: Programming Languages

Dr. Dylan Shell

Parsing and grammars

Arithmetic Expressions

Consider a simple form of expressions built up from single digits using the operations of addition + and multiplication *, together with parentheses.

We also assume that:

* and + associate to the right.

* has higher priority than +.

Example: Parsing a token

```
space :: Parser ()
space = many (sat isSpace) >>
        return ()

token :: Parser a -> Parser a
token p = space >>
          p >>= \v ->
          space >>
          return v

identifier :: Parser String
identifier = token ident

ident :: Parser String
ident = sat isLower >>= \x ->
        many (sat isAlphaNum) >>= \xs ->
        return (x:xs)
```

Formally, the syntax of such expressions is defined by the following context free grammar:

$$
\begin{aligned}
\mathit{expr} &\to \mathit{term}\ \text{'+'}\ \mathit{expr} \mid \mathit{term} \\
\mathit{term} &\to \mathit{factor}\ \text{'*'}\ \mathit{term} \mid \mathit{factor} \\
\mathit{factor} &\to \mathit{digit} \mid \text{'('}\ \mathit{expr}\ \text{')'} \\
\mathit{digit} &\to \text{'0'} \mid \text{'1'} \mid \ldots \mid \text{'9'}
\end{aligned}
$$

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

$$expr \longrightarrow term\ (\text{'+'}\ expr \mid \varepsilon)$$

$$term \longrightarrow factor\ (\text{'*'}\ term \mid \varepsilon)$$

Note: The symbol ε denotes the empty string.

It is now easy to translate the grammar into a parser that evaluates expressions, by simply rewriting the grammar rules using the parsing primitives.

That is, we have:

```
expr :: Parser Int
expr  = do t ← term
           do char '+'
              e ← expr
              return (t + e)
            +++ return t
```

expr ⟶ *term* ('+' *expr* | ε)

term ⟶ *factor* ('*' *term* | ε)

```
term :: Parser Int
term  = do f ← factor
           do char '*'
              t ← term
              return (f * t)
            +++ return f
```

$$expr \longrightarrow term\ ('+'\ expr \mid \varepsilon)$$

$$term \longrightarrow factor\ ('*'\ term \mid \varepsilon)$$

```
factor :: Parser Int
factor  = do d ← digit
             return (digitToInt d)
           +++ do char '('
                  e ← expr
                  char ')'
                  return e
```

Finally, if we define

```
eval    :: String → Int
eval xs = fst (head (parse expr xs))
```

then we try out some examples:

```
> eval "2*3+4"
10

> eval "2*(3+4)"
14

> eval "2+5-"
7

> eval "+5-"
*** Exception: Prelude.head: empty list
```

Here’s a variation:

$$
\begin{aligned}
&\mathit{expr} &&\rightarrow \mathit{term}\ \text{'+'}\ \mathit{expr} \mid \mathit{term} \\
&\mathit{term} &&\rightarrow \mathit{factor}\ \text{'*'}\ \mathit{term} \mid \mathit{factor} \\
&\mathit{factor} &&\rightarrow \mathit{digit} \mid \text{'('}\ \mathit{expr}\ \text{')'} \mid \text{'['}\ \mathit{expr}\ \text{']'} \\
&\mathit{digit} &&\rightarrow \text{'0'} \mid \text{'1'} \mid \ldots \mid \text{'9'}
\end{aligned}
$$

```
factor :: Parser Int
factor  = do d ← digit
             return (digitToInt d)
           +++ do char '('
                  e ← expr
                  char ')'
                  return e
           +++ do char '['
                  e ← expr
                  char ']'
                  return e
```