Your Grammar Is Bad

We can repair it... we have the technology

Troels Henriksen (athas@sigkill.dk)

September 18, 2014

Agenda

Last lecture, you saw how writing parsers can actually be fun. Today we tone it down a notch and make it slightly less fun, although still better than `yacc`. We will talk about:

- Left recursion
- Operator priority
- Tokenisation

Read this at home

> **Grammars and parsing with Haskell Using Parser Combinators**, *by Ken Friis Larsen and Peter Sestoft*

Available on Absalon.

From grammar to parser

- ```
  E ::= T + E
      | T - E
      | T
  T ::= int
  ```
- An example string

  `1 - 2 + 3`
- By the grammar, this has only the single parse

  `1 - (2 + 3)`
- How can we make sure we make the right decision in the rule for E?
  ```

Left factorisation

- ```
  E    ::= T Eopt
  Eopt ::= + T Eopt
         | - T Eopt
         |
  T    ::= int
  ```
- Now all choices between productions can be made by looking at the next symbol in the input.
  ```

From Grammar to Parser Skeleton

- Make a function (parser) for each non-terminal, replace choice | with the combinator `<|>`, use the `string` combinator for terminals, and use `do`-notation for sequences.
- ```
  I e = do t
           eopt
           return()
  eopt = (do string "-"
             t
             eopt
             return ()) <|>
         (do string "-"
             t
             eopt
             return ()) <|>
         return ()
  t = do integer
         return ()
  ```
  ```

Left-recursion

- ```
  E ::= E + T
      | E - T
      | T
  T ::= int
  ```
- Parses `1 + 2 - 3` as `(1 + 2) - 3`.
- We can left-factorise...

  ```
  E    ::= T Eopt
  Eopt ::= + E EOpt
         | - E EOpt
         |
  T    ::= int
  ```
- This now parses `1 + 2 - 3` as `1 + (2 - 3)`. We will need to rewrite the parse tree... sounds hard.
  ```

We have the technology!

- Enter:

```
chainl1 :: Parser a
        -> Parser (a -> a -> a)
        -> Parser a
```

- The parser `chainl1 prim op` parses `prim`s separated by `op`s.

```
data Exp = Con Int | Add Exp Exp | Sub Exp Exp
e = chainl1 t op
t = do x <- integer
       return (Con x)
op = (do string "+"
         return Add) <|>
     (do string "-"
         return Sub)
```

- Always does leftmost derivation.

Operator priority

- Operators with differing priority

```
E ::= E + T
    | E - T
    | E * T
    | E / T
    | T
T ::= int
```

- This we solve by adding multiple levels of nonterminals.

```
E0 ::= E0 + T
     | E0 - T
     | E1
E1 ::= E1 * T
     | E1 / T
     | T
T ::= int
```

Operator priority

Now we can use `chainl1` like before.

```
data Exp = Con Int | Add Exp Exp | Sub Exp Exp
                   | Mul Exp Exp | Div Exp Exp
e0 = chainl1 e1 op0
e1 = chainl1 t op1
t = do x <- integer
       return (Con x)
op0 = (do string "+"
          return Add) <|>
      (do string "-"
          return Sub)
op1 = (do string "/"
          return Div) <|>
      (do string "*"
          return Mul)
```

Whitespace

- In most parsers, we want to ignore whitespace between tokens.
- `spaces = many (satisfy isSpace)`
- Where to use `spaces`? Common solution: Litter it all over the place like some sacrifice to eldritch parser spirits.
- Better solution:

```
token :: Parser a -> Parser a
token p = do x <- p
             spaces
             return x
```

Maximum munch

- ```
  E0 ::= E0 + T
       | E0 - T
       | E1
  E1 ::= E1 * T
       | E1 / T
       | T
  T ::= int
      | if E0 then E0 else E0
      | var
  ```
- We want to ensure that the string `ifxthenyelsez` is parsed as a variable name.
- We want to ensure that `then` is not parsed as a variable name.
- Solution: `notFollowedBy`/`munch` and some ad-hoc hackery.
  ```

Summary

- Basic transformation of grammars to combinator-based parsers.
- Handle left-recursion with `chainl1`.
- Handle operator priority through grammar transformation.
- Have a principled approach to whitespace handling and munching.