interpreter tweaks
mhe authored 1 hour ago

afad15cf

Parser.md 6.46 KB

Parser

Recall our concrete syntax in BNF notation:

```
Program ::= Identifier := Expr;                 -- assignment
          | { [Program] }                       -- block
          | while (Expr) Program                -- whileStatement
          | If (Expr) Program                   -- ifStatement
          | If (Expr) Program else Program

Expr  ::= Expr1 | Expr1 OrOp   Expr             -- lowest precedence
Expr1 ::= Expr2 | Expr2 AndOp  Expr1
Expr2 ::= Expr3 | Expr3 EqOp   Expr2
Expr3 ::= Expr4 | Expr4 CompOp Expr3
Expr4 ::= Expr5 | Expr5 AddOp  Expr4
Expr5 ::= Expr6 | Expr6 MulOp  Expr5
Expr6 ::= Expr7 | NotOp Expr6
Expr7 ::= Constant | Identifier | (Expr)        -- highest precedence

OrOp   ::=  ||
AndOp  ::=  &&
EqOp   ::=  ==
CompOp ::=  <=  |  <  |  >=  |  >
AddOp  ::=  +   |  -
MulOp  ::=  *   |  /  | %
NotOp  ::=  !
```

The following is a direct translation of BNF to monadic parsing combinators:

```haskell
module Parser where

import Data.Char
import Control.Monad

import AbstractSyntax
import Parsing
```

The function `program` parses a program according to the above BNF definition. The result is a `Program` tree in the `Parser` monad:

```haskell
program :: Parser Program
```

Similarly, the various `expr` functions below parse expressions, with an expression tree in the `Parser` monad:

```haskell
expr, expr1, expr2, expr3, expr4, expr5, expr6, expr7 :: Parser Expr
```

And the following parse and return `Expr` constructors:

```haskell
orOp, andOp, eqOp, compOp, addOp, mulOp, notOp :: Parser ([Expr] -> Expr)
```

Based on the BNF production rule for programs, we define:

```haskell
program =
      assignment
  <|> block
  <|> whileStatement
  <|> ifStatement
```

where the functions `assignment`, `block`, `whileStatement` and `ifStatement` are defined below. The production rule for assignments is `Identifier := Expr;`, which we translate as follows to monadic parsing:

```
assignment =
  do
    i <- identif
    symbol ":="
    e <- expr
    symbol ";"
    return (i := e)
```

This works as follows:

- We parse an identifier with `identif` (defined below), which is bound to the variable `i`.
- We then parse the string `":="`.
- We then parse an expression with `expr`, which is bound to the variable `e`.
- We then parse the string `";"`.
- We finally return the program tree `i := e`.

The production rule for blocks is `{ [Program] }` (a list of programs enclosed in curly braces):

```
block =
  do
    symbol "{"
    ps <- many program
    symbol "}"
    return (Block ps)
```

The function `many`, applied to the function `program`, parses a list of programs. It is predefined in the type class `Alternative` (see the monadic parsing lecture notes, the textbook or hoogle). This works because the type constructor `Parser` is in the `Alternative` type class. The production rule for while-statements is `while (Expr) Program`:

```
whileStatement =
  do
    symbol "while"
    symbol "("
    e <- expr
    symbol ")"
    p <- program
    return (While e p)
```

The production rule for if-statements is

```
     If (Expr) Program
   | If (Expr) Program else Program
```

This becomes, in factorized form:

```
ifStatement =
  do
    symbol "if"
    symbol "("
    e <- expr
    symbol ")"
    p1 <- program
    ((do
        symbol "else"
        p2 <- program
        return (IfElse e p1 p2))
      <|>
       (return (If e p1)))
```

That is:

- Parse `"if"` and then `"("` and then an expression `e` and then `")"` and then a program `p1`.
- Then parse one of
  - `"else"` and then a program `p2` (then return the tree `IfElse e p1 p2`), or
  - nothing (then return the tree `If e p1`).

The following function is used in order to implement parsing of BNF definitions of the form

```
expr := expr' | expr' op expr
```

in factorized form:

```
binExpr :: Parser e -> Parser ([e] -> e) -> Parser e -> Parser e
binExpr expr' op expr =
  do
    e' <- expr'
    ((do
        o <- op
        e <- expr
        return (o [e',e]))
      <|>
        return e')
```

Then the definitions of expressions become:

```
expr  = binExpr expr1 orOp   expr
expr1 = binExpr expr2 andOp  expr1
expr2 = binExpr expr3 eqOp   expr2
expr3 = binExpr expr4 compOp expr3
expr4 = binExpr expr5 addOp  expr4
expr5 = binExpr expr6 mulOp  expr5

expr6 = expr7
     <|>
        do
          op <- notOp
          e <- expr6
          return (op [e])

expr7 = constant
    <|> do
          i <- identif
          return (Var i)
    <|> do
          symbol "("
          e <- expr
          symbol ")"
          return e
```

The above use the following helper functions:

```
parseOp :: String -> OpName -> Parser ([Expr] -> Expr)
parseOp s op = do
                 symbol s
                 return (Op op)

orOp   = parseOp "||" Or

andOp  = parseOp "&&" And

eqOp   = parseOp "==" Eq

compOp = parseOp "<=" Leq
     <|> parseOp "<"  Less
     <|> parseOp ">=" Geq
     <|> parseOp ">"  Greater

addOp  = parseOp "+"  Add
     <|> parseOp "-"  Sub

mulOp  = parseOp "*"  Mul
     <|> parseOp "/"  Div
     <|> parseOp "%"  Mod
```

```
notOp  = parseOp "!"  Not
```

This is to parse numerical constants:

```
constant :: Parser Expr
constant = do
             n <- integer
             return (Constant(toInteger n))
```

Notice that `integer` (defined in the textbook code Parsing.hs) parses an `Int` rather than an `Integer`, and this is why we need the type cast `toInteger`.

This is to parse identifiers, which are defined like in the textbook but excluding our keywords:

```
keywords = ["if", "else", "while"]

identif :: Parser String
identif =
  do
   cs <- token identifier
   guard (not (elem cs keywords))
   return cs
```

Parsing a program can cause an error:

```
parseProgram :: String -> Program
parseProgram xs = case parse program xs of
                   [(p , [])] -> p
                   [(_ , s)]  -> error ("syntax: unparsed string " ++ s)
                   _          -> error "syntax: failed to parse program"
```

There is no error when we get a singleton list (unambiguous parsing) consisting of a pair `(p,[])` where `p` is a program syntax tree and where `[]` indicates that there was nothing left unparsed after the program.

Next: Interpreter