CIS 552: Advanced Programming


```
{-# OPTIONS -Wall -fwarn-tabs -fno-warn-type-defaults  #-}
```

HW #4 Parsing

```
-- Advanced Programming, HW 4
-- by <YOUR NAME HERE> <pennkey>, <YOUR PARTNER'S NAME> <pennkey>
```

This homework provides more practice with monads, via the parsing monad that we developed in class. Feel free to use generic operations on Monads when completing this assignment.

```
import Control.Monad
```

Before starting this assignment:

0. Find a partner.

1. Download the test files, which are sample files for the WHILE language that you worked with on the last assignment. test.imp, fact.imp, abs.imp, times.imp. Take a look at these files to get an idea of the concrete syntax of the WHILE language.

2. Read Ch. 5 of Read World Haskell. In particular, that chapter goes through the design of a library called, "Prettify". The library is a simplified version of the HughesPJ library. In Problem 0, we will use the HughesPJ library for pretty-printing WHILE programs. Note that most of the contents of the library (such as char, text, etc.) we will import qualified so that it will not conflict with the other parts of this assignment. The documentation for this library summarizes its contents.

```
import Text.PrettyPrint.HughesPJ (Doc, (<+>),($$),(<>))
import qualified Text.PrettyPrint.HughesPJ as PP
```

3. Download the Parsing library that we developed in lecture Parser.hs ParserCombinators.hs

```
import Parser
import ParserCombinators
```

4. Download Main.hs, the non-literate version of this file to complete your assignment.

5. Remember that as you complete the assignment, you should be adding test cases. Parts of this assignment are intentionally vague, so the test cases that you provide specify your interpretation of the problem.

```
import Test.HUnit
```

6. After you complete the assignment, submit all three files, Main.hs, Parser.hs, and ParserCombinators.hs to the course website.

7. Recall the abstract syntax of the WHILE language that you worked with in Homework 3.

The WHILE language

```
type Variable = String

data Value =
    IntVal Int
  | BoolVal Bool
  deriving (Show, Eq)

data Expression =
    Var Variable
  | Val Value
  | Op  Bop Expression Expression
  deriving (Show, Eq)

data Bop =
    Plus
  | Minus
```

```
  | Times
  | Divide
  | Gt
  | Ge
  | Lt
  | Le
  deriving (Show, Eq)

data Statement =
    Assign Variable Expression
  | If Expression Statement Statement
  | While Expression Statement
  | Sequence Statement Statement
  | Skip
  deriving (Show, Eq)

main :: IO ()
main = do _ <- runTestTT (TestList [ t0,
                                     t11, t12, t13,
                                     t2 ])
          return ()
```

A Pretty Printer for WHILE

```
-- Problem 0
--------------------------------------------
```

The derived Show instances for the datatypes above are pretty hard to read, especially when programs get long.

A pretty printer is a function that converts a value into a document (or "Doc") a special type that can be used for rendering text. Once the Doc has been created, the rendering function can set parameters, such as the line length, etc. and figure out how to display the text nicely.

```
class PP a where
  pp :: a -> Doc
```

For example, the Bop instance of this class converts each binary operator into a document.

```
instance PP Bop where
  pp Plus   = PP.char '+'
```

```
  pp Minus  = PP.char '-'
  pp Times  = PP.char '*'
  pp Divide = PP.char '/'
  pp Gt     = PP.char '>'
  pp Ge     = PP.text ">="
  pp Lt     = PP.char '<'
  pp Le     = PP.text "<="
```

Your job is to fill in the definitions of pp for Values, Expressions and Statements.

```
instance PP Value where
  pp _ = error "TBD"

instance PP Expression where
  pp _ = error "TBD"

instance PP Statement where
  pp _ = error "TBD"

display :: PP a => a -> String
display = show . pp
```

Your code for your pretty printer should pass the following simple tests for displaying values, expressions and statements.

```
-- Simple tests

oneV,twoV,threeV :: Expression
oneV   = Val (IntVal 1)
twoV   = Val (IntVal 2)
threeV = Val (IntVal 3)

t0 :: Test
t0 = TestList [display oneV ~?= "1",
      display (BoolVal True) ~?= "true",
      display (Var "X") ~?= "X",
      display (Op Plus oneV twoV) ~?= "1 + 2",
      display (Op Plus oneV (Op Plus twoV threeV)) ~?= "1 + (2 + 3)",
      display (Op Plus (Op Plus oneV twoV) threeV) ~?= "1 + 2 + 3",
      display (Assign "X" threeV) ~?= "X := 3",
      display Skip ~?= "skip"  ]

--- Your own test cases
```

However, there are many ways that a pretty printer can display text, and a good

pretty printer may adjust its output based on the line length. For example, for short compound statements, it is reasonable for the pretty printer to try to fit everything one one line:

```
t0b :: Test
t0b  = display (If (Val (BoolVal True)) Skip Skip) ~?=
       "if true then skip else skip endif"
```

Alternatively, you may wish to write your pretty printer so that it always inserts line breaks.

```
t0b' :: Test
t0b' = display (If (Val (BoolVal True)) Skip Skip) ~?=
      "if true then\n  skip\nelse  skip\nendif"
```

A Parser for WHILE

```
-- Problem 1
--------------------------------------------
```

The dual problem to pretty printing is parsing. For this part of the assignment, you will practice with the monadic parser combinators that we discussed in class.

This part of the assignment uses the definition of the "Parser" type from Parser.hs. This module is augmented by generic combinators from ParserCombinators.hs. You should read over these modules before continuing.

Parsing Constants

First, we will write parsers for the `Value` type

```
valueP :: Parser Value
valueP = intP <|> boolP
```

To do so, fill in the implementation of

```
intP :: Parser Value
intP = error "TBD"
```

Next, define a parser that will accept a particular string `s` as a given value `x`

```
constP :: String -> a -> Parser a
constP _ _ = error "TBD"
```

and use the above to define a parser for boolean values where `"true"` and `"false"` should be parsed appropriately.

```
boolP :: Parser Value
boolP = error "TBD"
```

Continue to use the above to parse the binary operators

```
opP :: Parser Bop
opP = error "TBD"
```

Parsing Expressions

Next, the following is a parser for variables, where each variable is one-or-more uppercase letters.

```
varP :: Parser Variable
varP = many1 upper
```

Now define a parser combinator which takes a parser, runs it, then skips over any whitespace characters occurring afterwards

```
wsP :: Parser a -> Parser a
wsP p = error "TBD"
```

Use the above to write a parser for `Expression` values

```
exprP :: Parser Expression
exprP = error "TBD"
```

Of course, you will need to write some test cases for this parser. In particular, be sure to make sure that your parser succeeds even when there is white space *within* and *after* an expression.

```
t11 :: Test
t11 = TestList ["s1" ~: succeed (parse exprP "1 "),
                "s2" ~: succeed (parse exprP "1  + 2") ] where
  succeed (Left _)  = assert False
  succeed (Right _) = assert True
```

Parsing Statements

Next, use the expression parsers to build a statement parser.

```
statementP :: Parser Statement
statementP = error "TBD"
```

Again, don't forget to add your own test cases.

```
t12 :: Test
t12 = TestList ["s1" ~: p "fact.imp",
                "s2" ~: p "test.imp",
                "s3" ~: p "abs.imp" ,
                "s4" ~: p "times.imp" ] where
  p s = do { y <- parseFromFile statementP s ; succeed y }
  succeed (Left _)  = assert False
  succeed (Right _) = assert True
```

However, not only should you be able to parse all of the test cases, but your parser and pretty printer should satisfy the following round trip property: pretty printing a WHILE program followed by parsing should be the same as the original. (The other direction does not necessarily hold because of whitespace.)

```
testRT :: String -> Assertion
testRT filename = do
   x <- parseFromFile statementP filename
   case x of
     Right ast -> case parse statementP (display ast) of
       Right ast' -> assert (ast == ast')
       Left _ -> assert False
     Left _ -> assert False

t13 :: Test
t13 = TestList ["s1" ~: testRT "fact.imp",
                "s2" ~: testRT "test.imp",
                "s3" ~: testRT "abs.imp" ,
                "s4" ~: testRT "times.imp" ]
```

Lexing then Parsing

```
-- Problem 2
-----------------------------------------
```

The [illegible] problem 1 [illegible] due to the fact that it did

The parser from problem 1 was somewhat complicated by the fact that it did lexing and parsing simultaneously. As a result, the parser had to worry about skipping over whitespace.

Instead, most parsing is usually done in two stages: first lexing, which converts the input into a list of tokens, and then the actual parsing of that list of tokens.

We can use our parsing library to write a little lexer for the WHILE language. The tokens that we will use include variables, values, binary operations, and keywords. I.e.

```
data Token =
     TokVar String       -- variables
   | TokVal Value        -- primitive values
   | TokBop Bop          -- binary operators
   | Keyword String    -- keywords
      deriving (Eq, Show)

keywords :: [ Parser Token ]
keywords = map (\x -> constP x (Keyword x))
             [ "(", ")", ":=", ";", "if", "then", "else",
             "endif", "while", "do", "endwhile", "skip" ]

type Lexer = Parser [Token]

lexer :: Lexer
lexer = sepBy1
        (liftM TokVal valueP <|>
         liftM TokVar varP   <|>
         liftM TokBop opP    <|>
         choice keywords)
        (many space)

t2 :: Test
t2 = parse lexer "X := 3" ~?=
        Right [TokVar "X", Keyword ":=", TokVal (IntVal 3)]
```

Your job is to rewrite the expression and statement parsers above so that they parse a stream of tokens instead of a string. To do so, you will need to generalize the Parser and ParserCombinator libraries. Currently, this code only parses a list of characters (i.e. a String). However, you should modify it so that it supports parsing from any sort of list, including strings and lists of tokens.

Original version of problem 1 from UCSD.

- -

News :

Welcome to CIS 552!
See the home page for basic information about the course, the schedule for the lecture notes and assignments, the resources for links to the required software and online references, and the syllabus for detailed information about the course policies.

Links :

- Piazza
- Haskell.org
- GHC manual
- Library documentation
- Hackage