Answers to Quiz 6

Distribution of Scores

Range: Scores
5-6: 6 6
7-8: 8
9-10: 9 10 10
11-12: 11
13-14: 13 13 13 14 14 14
15-16: 15 15 15 15 16 16 16 16 16
17-18: 17 17 17 17 18 18 18 18 18
19-20: 19 20
Average ≈ 14.32 Median = 15

Question 1 (4 points)
(a) (2 points) Suppose that \( \text{precedent}(\odot) < \text{precedent}(+) \) and that both \( \odot \) and + are left-associative. Give a fully parenthesized version of:

\[
9 + 8 \odot 7 \odot 3 + 1 + 5 \odot 4 \odot 3 + 10
\]

Answer for 1(a)

\[
= (9 + 8) \odot 7 \odot (3 + 1) + 5 \odot 4 \odot (3 + 10)
\]

\[
= (((((9 + 8) \odot 7) \odot ((3 + 1) + 5)) \odot 4) \odot (3 + 10))
\]

(b) (2 points) Suppose that \( \text{precedent}(\odot) > \text{precedent}(+) \) and that both \( \odot \) and + are right-associative. Give a fully parenthesized version of:

\[
9 + 8 \odot 7 \odot 3 + 1 + 5 \odot 4 \odot 3 + 10
\]

Answer for 1(b)

\[
= 9 + (8 \odot 7 \odot 3) + 1 + (5 \odot 4 \odot 3) + 10
\]

\[
= (9 + ((8 \odot (7 \odot 3)) + (1 + (5 \odot (4 \odot 3)) + 10)))
\]

Problem 2 (4 points). Use Hutton’s library to write a parser for the language \( \{ a^n b^n \# : n \geq 0 \} \).

An answer for 2

\[
abn = \text{do as <- many (char 'a')}
\]
\[
bs <- \text{many (char 'b')}
\]
\[
\text{char '#'}
\]
\[
\text{if (length as == length bs)}
\]
\[
\text{then return ()}
\]
\[
\text{else failure}
\]

An alternative answer for 2

\[
abn' = \text{do} \{ \text{ab}
\]
\[
; \text{char '#'}
\]
\[
; \text{return ()}
\}
\]
\[
ab = \text{do} \{ \text{char 'a'}
\]
\[
; \text{ab}
\]
\[
; \text{char 'b'}
\]
\[
; \text{return ()}
\}
\]

\[
+++
\]
\[
\text{return ()}
\]
**Problem 3 (4 points).** Here is a grammar for prefix expressions

\[
E ::= + TE | T \\
T ::= * FT | F \\
F ::= a | \cdots | z
\]

Draw a parse tree for the string: \( + * a b + c d \)

**Answer for 3**

![Parse tree for the string: \( + * a b + c d \)](image)

---

**Problem 4 (4 points).** **Background.** Many programming languages support hexadecimal constants, which are expressed as follows: the sequence 0x (i.e., a zero then a lowercase x), followed by a sequence of one or more hex digits. For our purposes, a hex digit is either a digit between 0 and 9 or an uppercase letter between A and F. Examples of hexadecimal constants include 0x1A, 0x235, and 0xFFFF00.

**Your Problem.** Use Hutton’s Parsing library to implement a parser

\[
\text{hexConst} ::= \text{do} \text{ string "0x"} \\
\text{digits} \leftarrow \text{many1 (sat isHexDigit)} \\
\text{return "0x"++digits}
\]

**An answer for 4**

```haskell
hexConst = do string "0x" \\
            digits <- many1 (sat isHexDigit) \\
            return ("0x"++digits)
```

---

**Problem 5 (4 points).** A \( \text{NatVect} \) has the grammar

\[
\text{NatVect} ::= <\text{Nat};\text{Nat}^*> \\
\text{Nat} ::= (0 | 1 | \cdots | 9)^+
\]

**Examples:** <99> and <12;9;44;3>. **Note:** There is no whitespace in a \( \text{NatVect} \) string.

**Your Problem:** Use Hutton’s Parsing library to implement a parser

\[
\text{natVect} ::= \text{Parser [Int]}
\]

that parses a \( \text{NatVect} \) and returns the list of numbers in the list. Please use Hutton’s \text{nat} parser for parsing the \text{Nat}'s.

**An answer for 5**

```haskell
natVect = do char '<' \\
            n <- nat \\
            ns <- many semiNat \\
            char '>' \\
            return (n:ns)
```

```haskell
semiNat = do char ';' \\
            m <- nat \\
            return m
```
A few Haskell library functions

\(\langle\langle\rangle\rangle\) :: (Eq a) => [a] -> [a] 
(\()\) :: a -> [a] -> [a] 
(++) :: [a] -> [a] -> [a] 
concat :: [[a]] -> [a] 
concatMap :: (a -> [b]) -> [a] -> [b] 
elem :: (Eq a) => a -> [a] -> Bool 
delete :: (Eq a) => a -> [a] -> [a] 
drop, take :: Int -> [a] -> [a] 
drop, take :: Int -> [a] -> [a] 
filter :: (a -> Bool) -> [a] -> [a] 
foldl, foldr :: (a -> b -> b) -> b -> [a] -> b 
head, last :: [a] -> a 
init, tail :: [a] -> [a] 
map :: (a -> b) -> [a] -> [b] 
nub :: (Eq a) => [a] -> [a] 
union :: (Eq a) => [a] -> [a] 

Hutton's Parsing Combinators

module Parsing where

import Char
import Monad

infixr 5 +++

-- The monad of parsers ------------------------------
newtype Parser a = P (String -> [(a,String)])

instance Monad Parser where
    return v = P (
        \inp -> [(v,inp)])
    p >>= f = P \inp -> case parse p inp of 
        [] -> [] 
        [(v,out)] -> parse (f v) out

instance MonadPlus Parser where
    mzero = P (
        \inp -> [])
    p 'mplus' q = P \inp -> case parse p inp of 
        [] -> parse q inp 
        [(v,out)] -> [(v,out)]

-- Basic parsers ------------------------------------

failure :: Parser a
failure = mzero

item :: Parser Char
item = P \inp -> case inp of 
    [] -> [] 
    (x:xs) -> [(x,xs)]

parse :: Parser a -> String -> [(a,String)]
parse (P p) inp = p inp

-- Choice ------------------------------------------
(+++) :: Parser a -> Parser a -> Parser a
p +++ q = p 'mplus' q

-- Derived primitives -----------------------------
sat :: (Char -> Bool) -> Parser Char
sat p = do x <- item
    if p x then return x
    else failure
digit, lower, upper, letter, alphanum :: Parser Char
    digit = sat isDigit
    lower = sat isLower
    upper = sat isUpper
    letter = sat isAlpha
    alphanum = sat isAlphaNum

case :: Char -> Parser Char
case c = sat (== c)

string :: String -> Parser String
string [] = return []
string (x:xs) = do char x
    string xs
    return (x:xs)

many :: Parser a -> Parser [a]
many p = many1 p 'mplus' return []

many1 :: Parser a -> Parser [a]
many1 p = do v <- p
    vs <- many p
    return (v:vs)

ident :: Parser String
ident = do x <- lower
    xs <- many alphanum
    return (x:xs)
nat :: Parser Int
nat = do x <- many1 digit
    return (read x)

int :: Parser Int
int = do char '-'
    n <- nat
    return (neg n)

++ nat

space :: Parser ()
space = do many (sat isSpace)
    return ()

-- Ignoring spacing -------------------------------
token :: Parser a -> Parser a
token p = do space
    v <- p
    space
    return v

identifier :: Parser String
identifier = token ident

natural :: Parser Int
natural = token nat

integer :: Parser Int
integer = token int

symbol :: String -> Parser String
symbol xs = token (string xs)

-- Extras ------------------------------------------
isHexDigit :: Char -> Bool
isHexDigit c = elem c "0123456789abcdef"

ident :: Parser String
ident = do x <- lower
    xs <- many alphanum
    return (x:xs)
nat :: Parser Int
nat = do x <- many1 digit
    return (read x)

int :: Parser Int
int = do char '-'
    n <- nat
    return (neg n)

++ nat

space :: Parser ()
space = do many (sat isSpace)
    return ()

-- Ignoring spacing -------------------------------
token :: Parser a -> Parser a
token p = do space
    v <- p
    space
    return v

identifier :: Parser String
identifier = token ident

natural :: Parser Int
natural = token nat

integer :: Parser Int
integer = token int

symbol :: String -> Parser String
symbol xs = token (string xs)