Major Assignment 1

Date of Submission - ?? Weightage - 30%

This is the first of the three parts of a major assignment which will cover the implementation of a Haskell-like language, which we shall call μ-Haskell (pronounced micro-Haskell). The three parts are:

1. Writing a parser to convert programs in μ-Haskell to an internal representation in Haskell.
2. Translating the internal representation to G-code.
3. Interpreting G-code.

This is the grammar of μ-Haskell:

$$\begin{array}{rcl}
\mathit{program} & \rightarrow & \{\mathit{fundef}\}^* \ \mathit{exp} \\
\mathit{fundef} & \rightarrow & \mathit{fname} \ \{\mathit{var}\}^* = \mathit{exp} \\
\mathit{exp} & \rightarrow & \mathit{con} \mid \mathit{var} \mid \mathit{exp} + \mathit{exp} \mid \mathit{exp} - \mathit{exp} \mid \mathit{fname} \ \{\mathit{exp}\}^* \mid \\
 & & \mathit{exp} == \mathit{exp} \mid \mathit{if} \ \mathit{exp} \ \mathit{then} \ \mathit{exp} \ \mathit{else} \ \mathit{exp} \mid \mathit{exp} : \mathit{exp} \mid \\
 & & \mathit{car} \ \mathit{exp} \mid \mathit{cdr} \ \mathit{exp} \mid \mathit{null} \ \mathit{exp} \mid [\mathit{exp} \ \{, \mathit{exp}\}^*] \mid (\mathit{exp}) \\
\mathit{con} & \rightarrow & \mathit{intlit} \mid \mathit{boollit} \mid \mathit{nil}
\end{array}$$

$\{x\}^*$ above stands for 0 or more repetitions of x. Thus $[\mathit{exp} \ \{, \mathit{exp}\}^*]$ stands for lists of length ≥ 1. *nil* represents empty list.

Your job is to take a program written in μ-Haskell, parse it using the functional parsing techniques that you have studied and represent it in terms of the following Haskell data types. Assume that the input program does not contain type errors.

```
type Fname = String
type Var = String

data Program = Prog [Fundef] Exp deriving Show
data Fundef  = Fun String [String] Exp deriving Show
data Exp     = I Int |  V Var | B Bool | Nil |
               Fname String | -- both user defined and builtin
               App Exp Exp deriving Show
```

Here is a small program written in μ-Haskell:

```
append xs ys = if (null xs) then ys else (car xs:(append (cdr xs) ys))
reverse xs = if (null xs) then xs else (append (reverse (cdr xs)) (car xs))
reverse [4,5]
```

And here is the same program represented in Haskell:

```
Fun "append" ["xs","ys"] (App (App (App (Fname "If")
                               (App (Fname "null") (V "xs")))
                               (V "ys"))
                               (App (App (Fname "cons")
                                    (App (Fname "car") (V "xs")))
                                    (App (App (Fname "append")
                                          (App (Fname "cdr") (V "xs")))
                                          (V "ys"))))

Fun "reverse" ["xs"] (App (App (App (Fname "If")
                           (App (Fname "null") (V "xs")))
                           (V "xs"))
                           (App (App (Fname "append")
                                (App (Fname "reverse")
                                     (App (Fname "cdr") (V "xs"))))
                                (App (Fname "car") (V "xs"))))

App (Fname "reverse")
     (App (App (Fname "cons") (I 4))
          (App (App (Fname "cons") (I 5))
            Nil))
```

Example 2:
Code in μ-Haskell :

```
 4 + (5 - 6) + (fib 8)
```

Representation in Haskell :

```
App (App (Fname "+")
     (App (App (Fname "+")
           (I 4))
           (App (App (Fname "-") (I 5)) (I 6))))
     (App (Fname "fib") (I 8))
```

Remarks:

- Function applications are represented in curried form (using `App`).
- You will find that an identifier can be parsed as a variable name as well as a 0-ary function call. Deciding which one to be used, cannot be done at the parsing level. We will let you know how to handle this case. For now, any correct parse will do.
- You can assume that every function definition will begin from a new line. The main expression will also begin from a new line.
- Define the final wrapper function called `parse` which will take the program in μ-Haskell as a string and return the parsed program. You can use the following code for reading the input program file (say `pfile`):

```
main = do
  input <- readFile "pfile"
  let
    program = parse input
  print program
```