TINY parser, built up from the expression parser example from section 8.8 of Programming in Haskell, Graham Hutton, Cambridge University Press, 2007.

Parser for TINY

Expressions

Grammar

```
<expr> ::= 0 | 1 | true | false | read | <ide> | not <expr> |
           <expr> = <expr> | <expr> + <expr> | (<expr>)
```

Note that I've added parentheses to the productions to allow grouping, so disambiguating, for example:

x + y * z

can now be written:

(x + y) * z

or:

x + (y * z)

so the syntax can now make clear the intended (different) meanings.

Of course, the internal form does not need to change since it already, because of its tree structure, allows these two forms to be appropriately represented properly.

In fact, you will have found (especially if you read the notes form Hutton that I gave out) that the grammar above is not directly translatable into code which works. The main problem comes when you try to translate a production like:

<expr> ::= <expr> = <expr>

since this says: "to parse an expression of the form e1 = e2 first parse an expression (e1). To parse an expression, first parse an expression, and to parse an expression first parse an expression......" and you never get onto "...and then parse a =..."---which means in computational terms we get an "infinite" recursion, stack space being used and finally running out.

So, stratification is needed---and it's a stratified grammar for expressions that Hutton actually gives and implements, and which I essentially repeat below.

The grammar implemented is:

```
<expr> ::= <expr> + <expr> | <expr> = <expr> | <term>

<term> ::= not <expr>

<factor> ::= read | false | true | 0 | 1 |  <ide> | (<expr>)
```

```
> import Parsing
>
> type Ide  =  String
>
> data Exp  =  Zero | One | TT | FF | Read | I Ide | Not Exp | Equal Exp Exp | Plus Exp Exp
>              deriving Show
>
```

Here we have not binding tighter than = or +.

```
>
> expr                          :: Parser Exp
> expr                          =  do e1 <- term
>                                     do symbol "+"
>                                        e2 <- expr
>                                        return (Plus e1 e2)
>                                  +++
>                                  do e1 <- term
>                                     do symbol "="
>                                        e2 <- expr
>                                        return (Equal e1 e2)
>                                  +++
>                                  term
>
> term                          :: Parser Exp
> term                          =  do symbol "not"
>                                     e <- expr
>                                     return (Not e)
>                                  +++
>                                  factor
>
> factor                        :: Parser Exp
> factor                        =  do symbol "read"
>                                     return Read
>                                  +++
>                                  do symbol "false"
>                                     return FF
>                                  +++
>                                  do symbol "true"
>                                     return TT
>                                  +++
>                                  do symbol "0"
>                                     return Zero
>                                  +++
>                                  do symbol "1"
>                                     return One
>                                  +++
>                                  do i <- identifier
>                                     return (I i)
>                                  +++
>                                  do symbol "("
>                                     e <- expr
>                                     symbol ")"
>                                     return e
```

Here is a little function to try the expression parser with:

```
> eeval                          :: String -> Exp
> eeval xs                       =  case (parse expr xs) of
>                                     [(n,[])]  -> n
>                                     [(_,out)] -> error ("unused input " ++ out)
>                                     []        -> error "invalid input"
```

Commands

We started with the TINY grammar for commands:

```
<cmd>  ::= <ide> := <exp> | output <exp> |
           if <exp> then <cmd> else <cmd> fi |
           while <exp> do <cmd> od |(<cmd>)
```

Note that I've added parentheses to the productions, like in the expressions case, to allow grouping, so disambiguating, for example:

```
while n = 0 do n := n + 1; output n
```

can be read, by the unextended grammar, as either:

```
while n = 0 do (n := n + 1; output n)
```

or:

```
(while n = 0 do n := n + 1); output n
```

Also, as before, we have had to stratify so that productions like:

```
<cmd> ::= <cmd> ; <cmd>
```

do not lead to "infinite" recursion of the parser. So, the grammar actually implemented is:

```
<cmd> ::= <comp> ; <cmd> | <comp>

<comp>  ::= <ide> := <exp> | output <exp> |
           if <exp> then <cmd> else <cmd> fi |
           while <exp> do <cmd> |(<cmd>)
```

```
> data Cmd = Assign Ide Exp | Output Exp |
>            IfThenElse Exp Cmd Cmd |
>            WhileDo Exp Cmd |
>            Seq Cmd Cmd
>            deriving Show

> cmd                            :: Parser Cmd
> cmd                            =  do c1 <- comp
>                                      do symbol ";"
>                                         c2 <- cmd
>                                         return (Seq c1 c2)
>                                    +++
>                                    comp
>
> comp                            :: Parser Cmd
> comp                            =  do i <- identifier
>                                       do symbol ":="
>                                          e <- expr
>                                          return (Assign i e)
>                                    +++
>                                    do symbol "output"
>                                       e <- expr
>                                       return (Output e)
>                                    +++
>                                    do symbol "if"
>                                       e <- expr
>                                       do symbol "then"
>                                          c1 <- cmd
>                                          do symbol "else"
>                                             c2 <- cmd
>                                             return (IfThenElse e c1 c2)
>                                    +++
>                                    do symbol "while"
>                                       e <- expr
>                                       do symbol "do"
>                                          c <- cmd
>                                          return (WhileDo e c)
>
>                                    +++
>                                    do symbol "("
>                                       c <- cmd
>                                       symbol ")"
>                                       return c
>
```

Here is a little function to try the command parser with:

```
> ceval                           :: String -> Cmd
> ceval xs                        =  case (parse cmd xs) of
>                                       [(n,[])]  -> n
>                                       [(_,out)] -> error ("unused input " ++ out)
>                                       []        -> error "invalid input"
```

Our test program

```
> gordon = "sum := 0; x := read; (while not (x = true) do (sum := sum + x; x := read));output sum"
>
```