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

It is incomplete as not all the code for dealing with expressions is given, and none of the code for dealing with commands is given. That's coursework!

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, we find found (especially if we read the notes from Hutton) 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>

<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
```

Stuff missing here!!

```
> factor                        :: Parser Exp
> factor                        =  do symbol "read"
>                                     return Read
>                                  +++
>                                  do symbol "false"
>                                     return FF
```

More stuff missing here!!!

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"
```