Haskell Practice Driving Test
Operator Precedence Parsing

The maximum mark is 30

1 Parsing

A "parser" is a program that takes as input a sequence of *tokens* and which analyses them with respect to a *grammar* in order to determine the tokens' collective grammatical structure. In this exercise you are going to write a tokeniser and a parser for simple arithmetic expressions, written in a Haskell-like syntax. The input will be a string representing a well-formed Haskell-like expression and the output will be an *abstract syntax tree*, or AST, representing the expression.

You will be processing a very simple expression language comprising only *non-negative* integer constants, e.g. 7, 187, 54 etc., variables, e.g. `x`, `myVar`, `catch22`, operator applications, e.g. `x+y`, `z^2*5`, and bracketed (sub)expressions, e.g. `(a*(b-1))^(c+1)`.

Internally, these expressions will be represented by an AST with the following Haskell type:

```
type Operator = Char

data Expr = Num Int | Var String | Op Operator | App Expr Expr Expr
```

You may assume throughout that the first argument of an `App` constructor will always be of the form `Op o` for some operator `o`. The other two arguments represent the two arguments of the operator, which are both well-formed in this sense. Moreover, expressions of the form `Op o` can be assumed *only* to occur as the first argument to an `App`. You may assume that all input expressions will be well-formed in a similar sense. For example, expressions such as `*` and `x+(*^7)` are not well-formed and need not be catered for. Do *not* litter your code with superfluous error checks that validate well-formedness.

1.1 Ambiguity

The main difficulty with expression parsing is that there are potentially many ways to map a source expression into an AST depending on where we implicitly place brackets. For example, the expression `1+3*x^2` could be represented by:

```
App (Op '+') (Num 1) (App (Op '*') (Num 3) (App (Op '^') (Var "x") (Num 2)))
  the representation of 1+(3*(x^2))
App (Op '*') (App (Op '+') (Num 1) (Num 3)) (App (Op '^') (Var "x") (Num 2))
  the representation of (1+3)*(x^2)
App (Op '+') (Num 1) (App (Op '^') (App (Op '*') (Num 3) (Var "x")) (Num 2))
  the representation of 1+((3*x)^2)
```

Similarly, the expression `a^b^c` could be represented by:

```
App (Op '^') (Var "a") (App (Op '^') (Var "b") (Var "c"))
App (Op '^') (App (Op '^') (Var "a") (Var "b")) (Var "c")
```

Adding brackets explicitly in the input can resolve any potential ambiguity, e.g. `1+(3*x)^2` if we really mean the third option above. In general, however, the parser needs to know the operator precedences and associativities in order to produce an unambiguous mapping from input to AST. For this reason it is called an *operator precedence parser*. The language you will be parsing has just five operators, each of whose precedence and associativity is consistent with Haskell:

Operator	Precedence	Associativity
+	6	Left
-	6	Left
*	7	Left
/	7	Left
∧	8	Right

For reasons that will become apparent, it is also convenient to think of left and right parentheses as being operators (see later). Another special "sentinel" operator `$` will also be useful, as will be explained below. The complete operator table can be expressed in Haskell as follows:

```
data Associativity = L | N | R
                   deriving (Eq,Ord,Show)

ops :: [Operator]
ops = "+-*/^()$"

type Precedence = Int

opTable :: [(Operator, (Precedence, Associativity))]
opTable  = [('$',(0,N)), ('(',(1,N)), ('+',(6,L)), ('-',(6,L)),
            ('*',(7,L)), ('/',(7,L)), ('^',(8,R)), (')',(1,N))]
```

Note that `L` and `R` represent "left" and "right" associative, respectively, and `N` "non-associative" or "not applicable". For example, `*` is left associative with precedence 7, so that `2*3*5` means `(2*3)*5` and ^ is right associative with precedence 8, so that `2^3^5` means `2^(3^5)`.

2 Tokenisation

In this exercise, the job of the *tokeniser* is to map the input string (e.g. `"x+(y-z)^2"`) into a list of *tokens*, each of which is a separately recognisable component of the input. To keep it simple we are going to assume that there are just three types of token: numbers, variables and operators. Both left and right parentheses will be treated as operators. Moreover, to avoid having a separate `Token` data type, we're going to use the constructors `Num`, `Var` and `Op` of the `Expr` data type to represent the three token types. Thus, the above example would be tokenised into

```
[Var "x",Op '+',Op '(',Var "y",Op '-',Var "z",Op ')',Op '^',Num 2]
```

Note that the sentinel operator (`$`) is only used to simplify the definition of the parser – it will never appear in an input expression.

3 Dijkstra's "Shunting Yard" Algorithm

A neat way to parse tokenised expressions is to use the "Shunting Yard" algorithm, developed by the famous Dutch computer scientist Edsger Dijkstra. The idea is to read the stream of tokens one by one and to assemble the required AST representation with the help of two stacks. The *argument stack* contains the argument expressions associated with incomplete operator applications. The *operator stack* contains the operators whose argument expressions have not yet been determined.

The algorithm is best explained with an example. To begin with we'll assume that the input expressions contain no parentheses. The very last part of this exercise invites you to add them in, **but you should not attempt to do so until you have the basic algorithm working.**

Consider a simple example: `4+x^2-8*y`, which tokenises to:

```
[Num 4,Op '+',Var "x",Op '^',Num 2,Op '-',Num 8,Op '*',Var "y"]
```

We read each token in turn. Numbers and variables always represent the argument to some operator and so are pushed onto the expression stack. We thus begin by pushing the expression `Num 4`:

Note: Do not read anything into the arrow that "points to" `Num 4` – this serves only to simplify the layout of the diagrams.

When an operator is encountered there are precisely two possible courses of action:

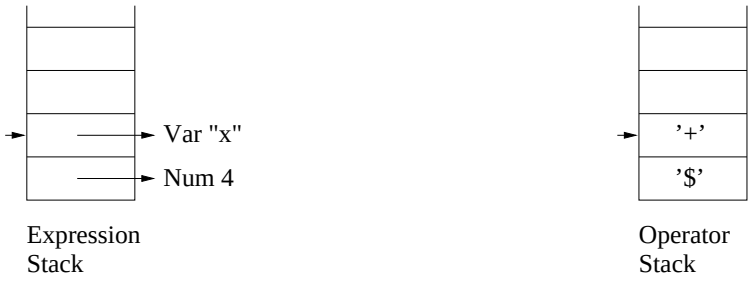
1. If the operator forms part of the argument of some earlier operator application (i.e. to the left of the current token), then the operator is pushed onto the operator stack. The process repeats from the next token in the input.

2. Otherwise, the operator at the top of the operator stack (if there is one) will have both of its argument expressions sitting on top of the argument stack. In this case all three items are respectively popped from their stacks and the expression representing the application of that operator to its two arguments (constructor `App`) is then formed and pushed back onto the argument stack. The process repeats, using the *same* input token, but with the now-modified stacks.

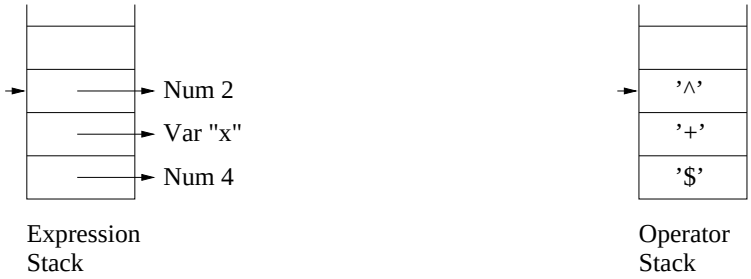
Exactly which course of action we take depends on a "comparison" (see below) between the operator on the input token stream and the operator at the top of the operator stack: `True` ⇒ action 1 and `False` ⇒ action 2.

Returning to the example, we clearly want to push the `+` operator on the operator stack as we have only its first argument on the argument stack. However, there is no operator on the operator stack to compare it with! We could treat empty stacks separately but a common trick is to use a special *sentinel* operator (`$`) which is such that when it is compared with any input operator the comparison always yields `True`. The sentinel always sits at the base of the operator stack.

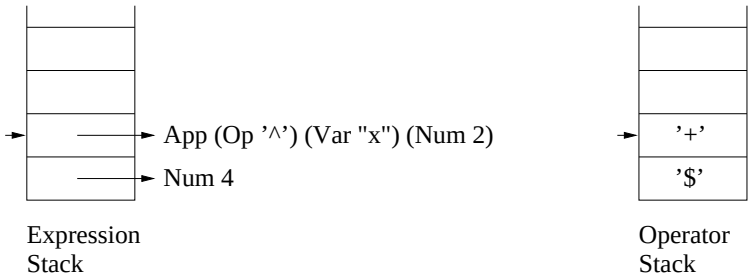
In the example, we thus push the `+` on the operator stack and, repeating the process for the next token, the `Var "x"` on the argument stack:



The next token is `Op '^'`. Because ^ has a higher precedence than `+` it would be wrong to form the subexpression representing `4+x`, because the `x^2` term should form the second argument of the application of `+`. We therefore push the ^ operator on the operator stack and continue. After pushing the next token, `Num 2`, on the argument stack we arrive at:



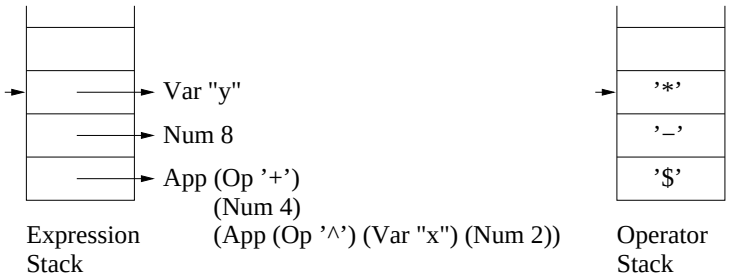
with the next input token being `Op -`. Now, the `-` operator has lower precedence than ^, so we know that the two expressions on the input stack form the arguments to ^. We thus form the now-completed application of ^ and push this onto the argument stack:



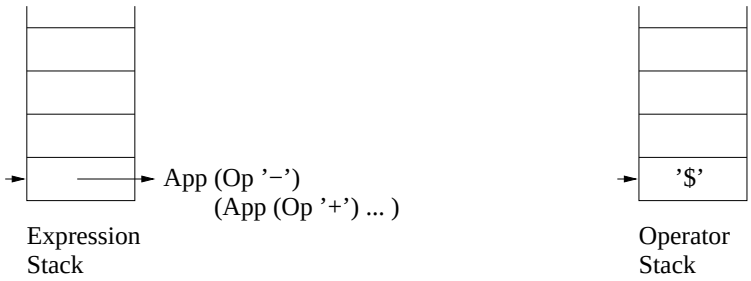
Note the order of the arguments.

We now continue, but from the *same* input token `-`, as it has not yet been pushed onto the operator stack. Now when we compare `-` with the top of the operator stack (`+`) we find that both operators have the same precedence. In this case the decision as to which course of action to take depends on the associativity of the operator at the top of the stack. If it is *right* associative, we push the input operator onto the operator stack (action 1) because we are in the process of forming the rightmost argument of the operator that is already on the stack. If it is left associative then we have completed another operator application as above, so we perform action 2; this is what happens in the example, as `+` is left associative.

The above process is repeated until there are no more input tokens. At this point the stacks look like this:



To finish the job we repeatedly apply the operator at the top of the operator stack to the top two argument expressions on the argument stack, popping the stacks accordingly as per action 2. This continues until there is only *one* item left on the argument stack; this is the final answer:



The resulting AST is:

```
App (Op '-')
    (App (Op '+') (Num 4) (App (Op '^') (Var "x") (Num 2)))
    (App (Op '*') (Num 8) (Var "y"))
```

4 What to do

You're going to develop some housekeeping functions, a tokeniser and a parser. As usual, uncomment each function definition as you go along.

All the types and definitions above are available in the skeleton file `Exam.hs` provided. Also included are two stack type synonyms and a "show" function that produces a fully bracketed string representation of an AST:

```
type ExprStack = [Expr]

type OpStack = [Operator]

showExpr :: Expr -> String
showExpr (Num n)
  = show n
showExpr (Var s)
  = s
showExpr (Op c)
  = [c]
showExpr (App op e e')
  = "(" ++ showExpr e ++ showExpr op ++ showExpr e' ++ ")"
```

You might find the show function useful when checking your output, particularly as `Expr` objects themselves are hard to read. For example, with the above example:

```
Main> showExpr (buildExpr "4+x^2-8*y")
"((4+(x^2))-(8*y))"
```

which verifies that the bracketing implicit in the AST has been performed correctly.

1. Define functions `precedence :: Operator -> Precedence` that returns the precedence of a given operator and `associativity :: Operator -> Associativity` that returns the associativity of a given operator, both using the `opTable` provided. For example:

   ```
   Main> precedence '*'
   7
   Main> associativity '^'
   R
   ```

   [2 Marks]

2. Define three functions: `higherPrecedence :: Operator -> Operator -> Bool` that returns `True` iff the first operator provided has higher precedence than the second; `eqPrecedence :: Operator -> Operator -> Bool` which returns `True` iff the first operator provided has the same precedence as the second; `isRightAssociative :: Operator -> Bool` that returns `True` iff the given operator is right associative. For example,

```
Main> higherPrecedence '*' '-'
True
Main> eqPrecedence '*' '/'
True
Main> isRightAssociative '+'
False
```

[3 Marks]

3. Define a function `supersedes :: Operator -> Operator -> Bool` that returns `True` iff the first operator "supersedes" the second. `a` supersedes `b` iff `a` has higher precedence than `b` or if `a` has the same precedence as `b` and `a` is right associative. For example:

```
Main> supersedes '+' '-'
False
Main> supersedes '*' '^'
False
Main> supersedes '^' '^'
True
```

[1 Mark]

4. In order to tokenise the input string, you need to be able to build numbers (here, non-negative integers) from their string representation. Thus, define a function `stringToInt :: String -> Int`. For example,

```
Main> stringToInt "0"
0
Main> stringToInt "96118"
96118
```

Hint: Recall that the character representation of a digit can be converted to its integer equivalent via `ord c - ord '0'`, where `c` is the character. Assume each element of the list is a digit character.

[4 Marks]

5. Now define a function `tokenise :: String -> [Token]` that generates a list of tokens from the string representing a well-formed expression. To do this, traverse the string from left to right, generating the tokens as you go along. At each step, look at the first character:

   - If it is whitespace, e.g. `' '`, `'\t'`, `'\n'` etc., ignore it. Use the built-in `isSpace` function to check this.
   - If it is an operator symbol, `o` say (the complete list of operator symbols is defined in `ops`), form the token `Op o` from it.
   - If it is neither of the above then the token is either a number (`Num`) or a variable (`Var`) – you can tell by looking at the first character. Whichever it is, you need to split the string at the next *non alphanumeric* character; equivalently, at the next whitespace or operator character. Hint: you might find the built-in `break` function useful when doing this.

- If the token is a variable, form the expression `Var cs` where `cs` is the input to the left of the split (break) point.
- If the token is a number, similarly form the expression `Num n` where `n` is obtained from `stringToInt`.

Of course, you need to tokenise the remainder of the input string recursively, having extracted the first token. Note that if you get this right, parentheses (to be considered later) should be interpreted by the tokeniser as operators. This is exactly what you need for the last part of the exercise, but don't worry about them just yet. For example,

```
Main> tokenise "5-8*7"
[Num 5,Op '-',Num 8,Op '*',Num 7]
Main> tokenise "x1*x2-x3^2"
[Var "x1",Op '*',Var "x2",Op '-',Var "x3",Op '^',Num 2]
Main> tokenise "a  + b^  3 \t - \n  6"
[Var "a",Op '+',Var "b",Op '^',Num 3,Op '-',Num 6]
```

[5 Marks]

You're now going to build the parser, from which the following function (provided) can be used to combine the tokeniser and parser to map `String`s to `Expr`s:

```
buildExpr :: String -> Expr
buildExpr s
  = parse (tokenise s) ([], ['$'])
```

You may find this function useful in testing. Notice that the `parse` function accepts a list of tokens and a pair of stacks (respectively the argument stack and the operator stack, suitably primed with the sentinel operator): `parse :: [Token] -> (ExprStack, OpStack) -> Expr`.

Reminder: assume for now that there are no parentheses in the input expression.

6. To begin, define a function `buildOpApp :: (ExprStack, OpStack) -> (ExprStack, OpStack)` that builds an operator application from the topmost element of the operator stack and the top two elements of the argument stack, and which pushes the result on top of the argument stack. The function should return the two stacks, suitably modified. For example:

```
Main> buildOpApp ([Num 2, Var "x", Num 8], ['^', '*', '$'])
([App (Op '^') (Var "x") (Num 2),Num 8],"*$")
Main> buildOpApp ([App (Op '^') (Var "x") (Num 2),Num 8],"*$")
([App (Op '*') (Num 8) (App (Op '^') (Var "x") (Num 2))],"$")
```

Note that the operator stack type is synonymous with `String`, hence the double-quote shorthand. Note also that the top of a stack is physically the same as the head of the list used to represent it.

[2 Marks]

7. Now define `parse`. All you need to do is apply the rules outlined in Section 3 above. Recall:

   - You terminate when the token list is empty and there is one item left on the argument stack. Hint: do *not* use `length` to check this!
   - If the token list is empty when there is more than one item on the argument stack, build an operator application (`buildOpApp`).

- If the next token is an operator (`Op o`), implement either action 1 or action 2 above, by comparing (`supersedes`) `o` with the top of the operator stack.
- Otherwise, the next token is either a number or a variable, in which case it is should be pushed on the argument stack, before processing the remaining tokens.

For example,

```
Main> buildExpr "3+7*9"
App (Op '+') (Num 3) (App (Op '*') (Num 7) (Num 9))
Main> buildExpr "x^y^2"
App (Op '^') (Var "x") (App (Op '^') (Var "y") (Num 2))
Main> showExpr (buildExpr "x^y*z+y^y^2")
"(((x^y)*z)+(y^(y^2)))"
```

[10 Marks]

8. **You should only attempt this question if you have completed the above.** By amending your solution so far (save the working version in case you mess up!), extend your parser (`parse`) to handle bracketed expressions. Make sure also that the tokeniser already recognises brackets as operators, e.g.

```
Main> tokenise "8+(9*7)"
[Num 8,Op '+',Op '(',Num 9,Op '*',Num 7,Op ')']
```

Hints: If you treat `'('` as an operator and push it on the operator stack, you'll notice that its defined precedence (1) is such that all operator applications between it and the matching `')'` will be formed in their entirety – the `'('` forms a natural break from the preceding tokens, as required. Q: How should you handle the `Op ')'` token? For example,

```
Main> buildExpr "(((6)))"
Num 6
Main> buildExpr "x*(5-y)^3"
App (Op '*') (Var "x") (App (Op '^') (App (Op '-') (Num 5) (Var "y")) (Num 3))
Main> showExpr (buildExpr "x*(5-y)^3")
"(x*((5-y)^3))"
```

[3 Marks]