Haskell Interim Test

Imperial College London

Department of Computing

Expression Parsing

Friday 8th November 2024

14:00 to 16:00

TWO HOURS

- The maximum mark is 20.
- Please make your swipe card visible on your desk.
- After the planning time log in using your username as **both** your username and password.
- Work in the file named `Parser.hs` inside the `parser/src` subfolder of your Home folder. **Do not move any files**.
- You are **not permitted** to edit the folder structure, or edit either the `hit.cabal` or `cabal.project` files without having been directed by the Examiners; any changes will be reverted.
- You may add additional tests, which will be reflected in the final script (*like in the PPTs*), but these will not be assessed. However, any changes made to the code that cause the **original given** tests to not compile will incur a compilation penalty.

1 Expression parsing

An expression parser takes as input the textual representation of an expression (a `String`) and generates as output the internal representation of that expression, here as a Haskell data type. In this exercise expressions are made from *non-negative* integer constants, e.g. 7, 187, 54 etc., variables, which are alpha-numeric strings beginning with a letter, e.g. `x`, `MAXINT`, `thx1138`, and binary operator applications, e.g. `x+y`, `z^2*5`.

A parser typically works by first turning a given input string into a list of *tokens* (`[Token]`), with each token representing a non-negative integer, variable or operator symbol. In this test, tokens are then turned into an expression tree (`Expr`), and the resulting tree's structure reflects the *precedence* and *associativity* of the various operators. The following types are included in the module `Types.hs`:

```
type Operator = Char

data Token = TNum Int | TVar String | TOp Operator deriving (Eq, Show)

data Expr = ENum Int | EVar String | EApp Operator Expr Expr
            deriving (Eq, Show)

type Precedence = Int
data Associativity = L | N | R deriving (Eq, Show)
```

Hopefully, you can see that `Token` and `Expr` types are data representations of the components described above.

The `Precedence` type, which is just an `Int`, represents the "binding-power" of an operator: when deciding how to bracket the expression `x ○ y ● z` for some arbitrary operators ○ and ●, we must compare their precedence:

- If `precedence ○ < precedence ●` then the expression must be parsed as though it were bracketed `x ○ (y ● z)`.
- If `precedence ○ > precedence ●` then it should be parsed as `(x ○ y) ● z`.
- If `precedence ○ == precedence ●` then the assumed bracketing depends on the operators' *associativity*. If they have different associativity then the expression is ill-formed. If they have the same associativity then:
  - if `associativity ○ = L`, the assumed bracketing should be `(x ○ y) ● z`
  - if `associativity ○ = R`, the assumed bracketing should be `x ○ (y ● z)`
  - if `associativity ○ = N`, the operator is not associative and the expression is ill-formed. `N` can also represent "not-applicable"‘, as we'll see later.

As an example, assuming Haskell's built-in operator precedences the input string `1+3*x^2` should be parsed as though it were bracketed `1+(3*(x^2))`, which means that the representation we ultimately require is:

```
EApp '+' (ENum 1) (EApp '*' (ENum 3) (EApp '^' (EVar "x") (ENum 2)))
```

Brackets (parentheses) can also appear in the input strings, in which case the parser should process them so as to yield the correct expression tree. For example `2*(3+7)` should be parsed as:

```
EApp '*' (ENum 2) (EApp '+' (ENum 3) (ENum 7))
```

There are five arithmetic operators, each of whose precedence and associativity is consistent with the equivalent Haskell operator:

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. Another special "sentinel" operator $ will also be useful, as described below. Note that the sentinel operator ($) is only used to simplify the definition of the parser – it will never appear in an input expression. The properties of each operator are captured in an operator table, which is provided in the skeleton file:

```
ops :: [Operator]
ops = ['+', '-', '*', '/', '^', '(', ')', '$']

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

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)`. An expression like `2*3^7` will bracket as `2*(3^7)`.

2 Tokenisation

The *tokeniser* converts a given input string, e.g. `"x+(y-z)^2"`, into a list of `Tokens`, each of which is a separately recognisable component of the input. Because left and right parentheses are treated as operators the above example will be tokenised into

```
[TVar "x",TOp '+',TOp '(',TVar "y",TOp '-',TVar "z",TOp ')',
 TOp '^',TNum 2]
```

The job of the parser is then to distill a tree-structure from this list.

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 expression tree with the help of two stacks. The *expression stack* contains the argument expressions associated with incomplete operator applications. The *operator stack* contains the operators whose argument expressions have not yet been formed.

Number and variable tokens always represent the arguments to some operator and so are pushed onto the expression stack in the form of `Exprs`. When an operator is encountered in the input there are two cases to consider:

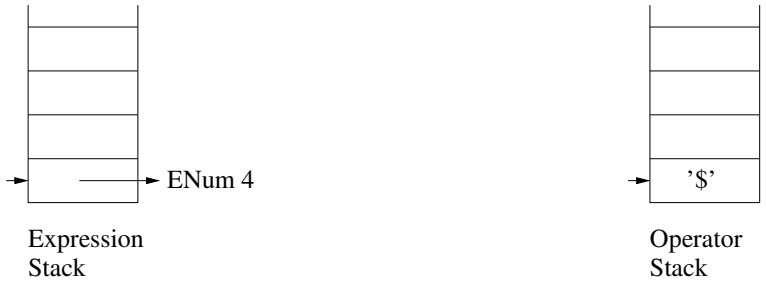
1. If the operator has higher precedence than the operator on top of the operator stack then we have not yet formed its argument expressions, so the operator is pushed onto the operator stack. The same rule applies if it has the *same* precedence as the operator on top of the operator stack and is right associative. The process repeats from the next token in the input.

2. Otherwise, the operator at the top of the operator stack has its two argument expressions sitting on top of the expression stack. In this case the operator and two arguments expressions are removed (popped) from their respective stacks and an application `EApp` is formed from the components. This is then pushed back onto the expression stack. The process repeats, using the *same* input token, but with the now-modified stacks.

To illustrate this we'll now walk through an example. You may wish to skip this on first reading and refer back to it if you get stuck. Consider an input string `4+x^2-8*y`, which tokenises to:

```
[TNum 4,TOp '+',TVar "x",TOp '^',TNum 2,TOp '-',TNum 8,
 TOp '*',TVar "y"]
```

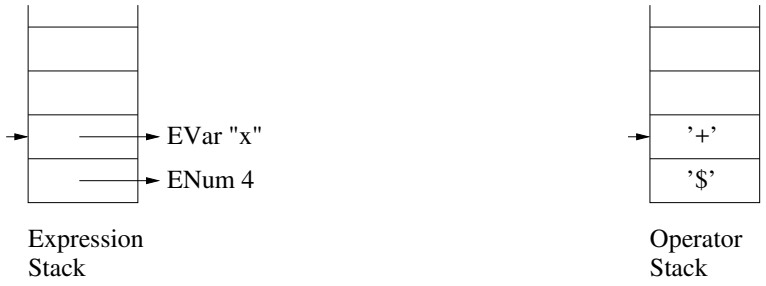
We now read each token in turn, beginning with `TNum 4`. We form an expression from the literal `4`, i.e. `ENum 4`, and this is pushed onto the expression stack:



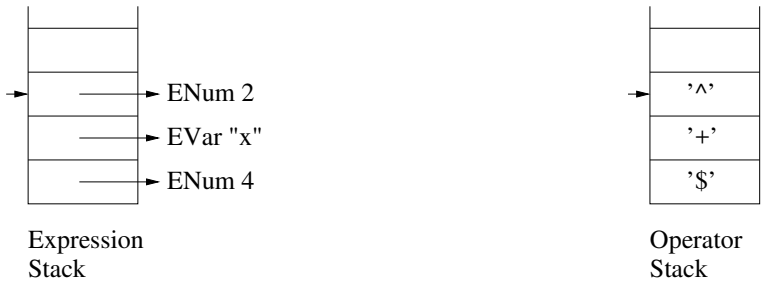
The next token is `TOp '+'`. We clearly want to push the `+` operator on the operator stack as we have only its first argument on the expression stack. However, there is no operator on the operator stack, so we can't perform test 1. above. The trick is therefore to initialise the operator stack with a special *sentinel* operator (`$`) which has

lower precedence than every other operator. Doing so means that test 1. succeeds in this case and `+` is pushed on to the operator stack.

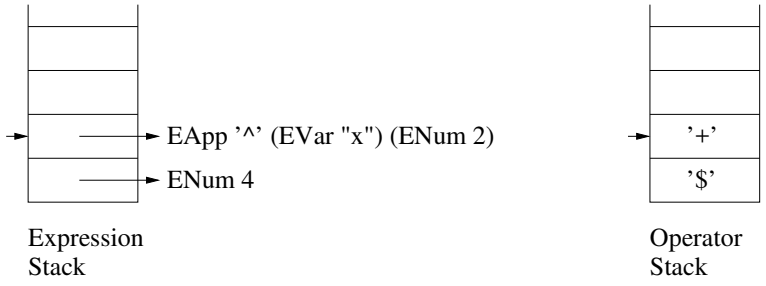
The next token is `TVar "x"` and, similarly to the above, this results in the expression `EVar "x"` being pushed onto the expression stack:



The next token is `TOp '^'`. Because ^ has a higher precedence than the `+` on top of the operator stack, we cannot form an expression from it, as its arguments have not yet been formed. We therefore push the ^ operator on the operator stack. The next token, `TNum 2`, is pushed onto the expression stack as the expression `ENum 2`: on top of the operator stack,



The next input token is `TOp -`. This has lower precedence than ^, so we know that the two expressions on the input stack form the arguments to ^. We therefore remove (pop) the operator from the operator stack and the top two items on the expression stack and form an application (`EApp`) which is pushed onto the expression stack:

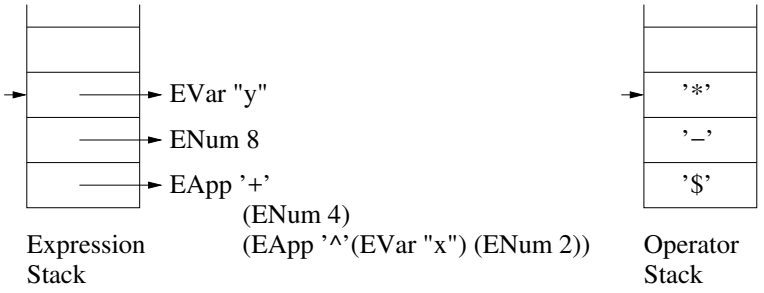


Note the order of the arguments.

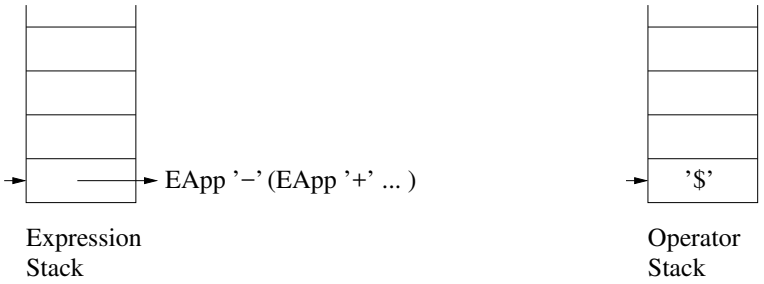
We now continue, but from the *same* input token -, as it has not yet been processed. When we compare - with the `+` on top of the operator stack we find that both

have the same precedence. Because `+` is left associative we know that we have completed another operator application, similar to the above, so we pop the operator and expression stacks and form another application,
`EApp '+' (ENum 4) (EApp '^' (Var "x") (ENum 2))`, which is pushed back onto the expression stack.

The above process is repeated until the token list is empty. At this point the stacks look like this:



To finish the job we repeatedly apply (`EApp`) the operator at the top of the operator stack to the top two elements of the expression stack, popping the stacks accordingly as described above. This continues until there is only *one* item left on the expression stack – this constitutes the termination condition:



The final result is the expression at the top of the expression stack:

```
EApp '-'
     (EApp '+' (ENum 4) (EApp '^' (EVar "x") (ENum 2)))
     (EApp '*' (ENum 8) (EVar "y"))
```

4 What to do

You're going to develop some housekeeping functions, a tokeniser, a utility function and a parser. To help with testing the module `Examples.hs` contains various example strings and expressions, which will be referred to in the following.

You should assume throughout that all expressions are syntactically well-formed, thus avoiding the need to check for errors. For example, badly-formed expressions such as `35 people`, `undefined$operator` etc. can be assumed not to occur.

Stacks are implemented as lists, so pushing can be done using `:` and popping by pattern matching. The following are also included in `Types.hs`:

```
type ExprStack = [Expr]

type OpStack = [Operator]
```

Note that the items on the operator stack are just operator symbols, e.g. `'+'`, rather than expressions, e.g. `Op '+'`.

A function `stringToInt` is provided in the skeleton to convert a `String` of digits to its corresponding `Int`[1]. A function `prettyExpr` is also provided which generates a printable representation of an `Expr` as a `String`; parentheses are used in order to reflect the structure of the tree. For example:

```
Parser> stringToInt "42"
42
*Parser> prettyExpr e1
"(4+(3*7))"
*Parser> prettyExpr e2
"((4+(x^2))-(8*y))"
```

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. A precondition is that the operator has a binding in the `opTable`. For example:

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

   [2 Marks]

2. 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:

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

   [2 Marks]

[1] `stringToInt` is synonymous with Haskell's `read` function.

3. Define a function `tokenise :: String -> [Token]` that generates a list of tokens from a string representing a well-formed expression. To do this, traverse the input 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, `op` say, then form the token `TOp op` from it. Recall that the valid operators are defined in `ops`.
   - If it is neither of the above then the token is either a number (`TNum`) or a variable (`TVar`) and you can work out which it is from the first character, e.g. using the built-in function `isDigit` or `isAlpha`. In both cases you now need to split the input string at the next *non alphanumeric* character[2]; you might consider using `break` or `span` function to do this – these will produce two strings, `(s, s')` say.
     - If the token is a variable, form the token `TVar s`
     - If the token is a number, form the token `TNum n` where `n` is obtained from `stringToInt s`.

   Of course, you need to tokenise the remainder of the input string recursively, having generated the first token.

   Note that parentheses (to be considered later) should be interpreted by the tokeniser as operators, but you don't need to worry about parsing them until the last part of the exercise. For example,

```
*Parser> tokenise "force^2"
[TVar "force",TOp '^',TNum 2]
*Parser> tokenise "5-8*7"
[TNum 5,TOp '-',TNum 8,TOp '*',TNum 7]
*Parser> tokenise "a  + b^  3 \t - \n  6"
[TVar "a",TOp '+',TVar "b",TOp '^',TNum 3,TOp '-',TNum 6]
*Parser> tokenise "5*(x-y)"
[TNum 5,TOp '*',TOp '(',TVar "x",TOp '-',TVar "y",TOp ')']
```

   [6 Marks]

4. Using `tokenise`, define a utility function `allVars :: String -> [String]` that will return the list of all variable names appearing in an expression string, in some order, without duplicates. For example,

```
*Parser>  allVars "7*x1+y2-b+x1"
["x1","y2","b"]
```

[2] Remember, we are assuming the input is well-formed, so you would never have something like `"1ab"` appearing. A digit will be followed by only digits, so an alpha-numeric check suffices.

[2 Marks]

5. You're now going to build the parsing function, `parse`. In order to test this the following function is provided for combining the tokeniser and parsing functions:

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

Assume for now that there are no parentheses in the input expression.

(a) Define the function `parse :: [Token] -> ExprStack -> OpStack -> Expr` by encoding the rules outlined in Section 3 above. For example:

```
*Parser> s1
"1+7*9"
*Parser> parseExpr s1
EApp '+' (ENum 1) (EApp '*' (ENum 7) (ENum 9))
*Parser> s2
"4+x^2-8*y"
*Parser> prettyExpr (parseExpr s2)
"((4+(x^2))-(8*y))"
*Parser> parseExpr s2 == e2
True
```

[5 Marks]

(b) By amending your solution so far (*save the working version in case you mess up!*), extend your program to handle bracketed expressions.

Hints: Modify `supersedes` so that `'('` supersedes any other operator *and vice versa.* This will ensure that open brackets always get pushed onto the operator stack and that all other operators will get pushed likewise when a `'('` is at the top of the operator stack. You then need to handle the `TOp ')'` token separately. How?! For example,

```
*Parser> parseExpr "(1)"
ENum 1
*Parser> s4
"(4+x)^(2-8)*y"
EApp '*' (EApp '^' (EApp '+' (ENum 4) (EVar "x")) (EApp '-' (ENum 2)
(ENum 8))) (EVar "y")
*Parser> prettyExpr (parseExpr s4)
"(((4+x)^(2-8))*y)"
*Parser> parseExpr s4 == e4
True
```

[3 Marks (making 8 in total for the parser)]