Haskell Interim Test

Imperial College London

Department of Computing

Expression Parsing

Monday 14 December 2020
09:00 to 12:00
THREE HOURS
(including 10 minutes planning time)

- The maximum total is **20 marks**.
- Credit will be awarded throughout for conciseness, clarity, *useful* commenting, and the appropriate use of the various language features.
- **Important:** TWO MARKS will be deducted from solutions that do not compile in the test environment, which will be the same as the lab machines. Comment out any code that does not compile before you submit.
- The examples and test cases here are not guaranteed to exercise all aspects of your code. You are therefore advised to define your own tests to complement the ones provided.
- The extracted files should be **directly inside** your gitlab repository. You can extract the archives by adjusting the following command: `7z x filename.7z`
- Push the final version of your code to `gitlab` before the deadline, and then go to `LabTS`, find the final/correct commit and submit it to `CATe`.

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 as a Haskell data type. To do this it first turns the string into a list of tokens (`[Token]`), each of which is either a *non-negative* integer, a variable or an operator symbol. The parser then turns the list of tokens into an expression tree (`Expr`) whose structure reflects the precedence and associativity of the various operators.

Expressions are made from *non-negative* integer constants, e.g. 7, 187, 54 etc., variables, e.g. `x`, `y1`, `catch22` and operator applications, e.g. `x+y`, `z^2*5`; there are separate constructors for each in the internal representation. The following types are included in the module `Types.hs` provided:

```
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,Ord,Show)
```

Note that `L` and `R` represent "left" and "right" associative, respectively, and `N` "non-associative" or "not applicable".

As an example, 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 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 must 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))
```

The language you will be parsing has five operators, each of whose precedence and associativity is consistent with the equivalent Haskell operators:

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 (see below). 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)`.

2 Tokenisation

The job of a *tokeniser* is to map an input string, e.g. `"x+(y-z)^2"`, into a list of `Token`s, each of which is a separately recognisable component of the input. Both left and right parentheses will be treated as operators, so the above example would be tokenised into

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

Number and variable tokens always represent the arguments to some operator and so are pushed onto the expression stack in the form of `Expr`s. When an operator is encountered 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 assembled its argument expressions, so the operator symbol (type `Operator`) is pushed onto the operator stack. The same rule applies if it has the *same* precedence as the operator on top

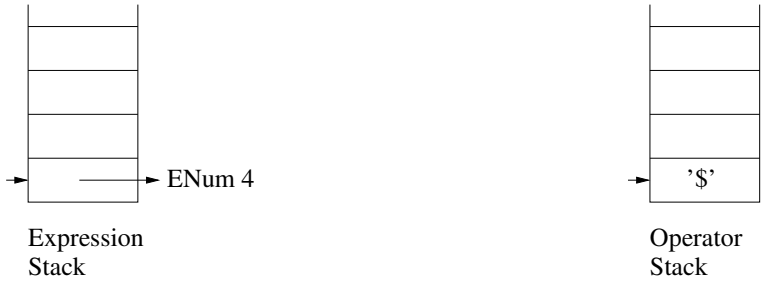
of the operator stack and is right associative. The process repeats from the next token in the input.

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 symbol 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:

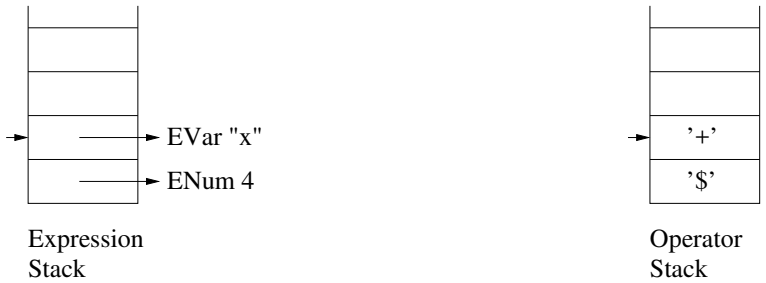
```
[TNum 4,TOp '+',TVar "x",TOp '^',TNum 2,TOp '-',TNum 8,
 TOp '*',TVar "y"]
```

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:

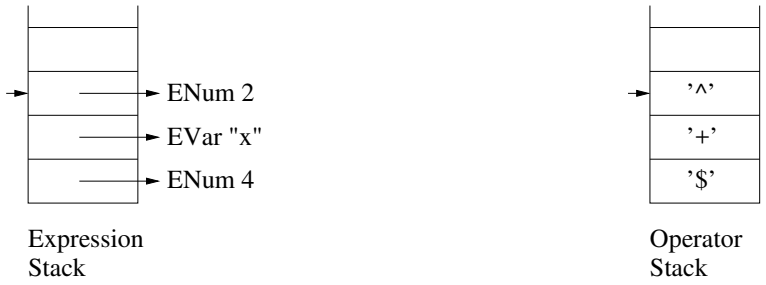


The next token is `TOp '+'`. We clearly want to push the `+` operator on the operator stack as we have only its first argument on the expression stack. However, there is no operator on the operator stack, so we can't perform test 1. above. The trick is therefore to initialise the operator stack with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

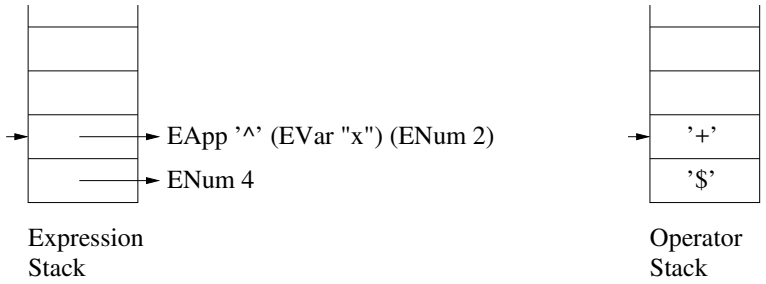
The next token is `TVar "x"` and, similarly to the above, this results in the expression `EVar "x"` being pushed onto the expression stack:



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,



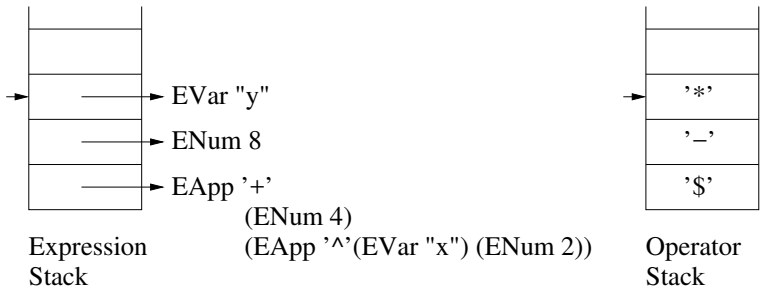
The next input token is `TOp -`. This has lower precedence than `^`, so we know that the two expressions on the input stack form the arguments to `^`. We therefore remove (pop) the operator from the operator stack and the top two items on the expression stack and form an application (`EApp`) which is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



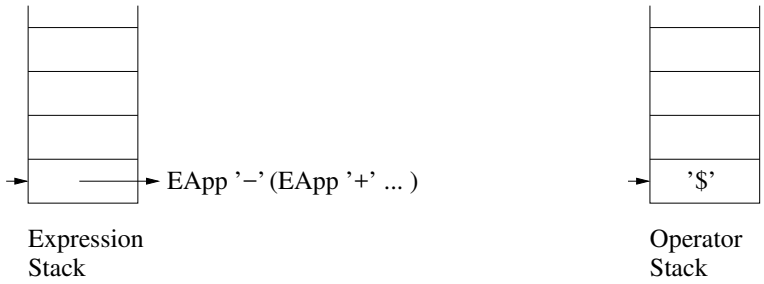
Note the order of the arguments.

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

All the types and definitions above are available in the skeleton file provided. Stacks are implemented as lists, so pushing is done using `:` and popping is done by pattern matching. The following are also included in `Types.hs`:

```
type ExprStack = [Expr]

type OpStack = [Operator]
```

A function `stringToInt` is provided for converting a `String` of digits to its corresponding `Int`. The function `showExpr` generates a printable representation of an `Expr` as a `String`, e.g. For example (`expParser` is described later),

```
Parser> stringToInt "96118"
96118
```

```
*Parser> showExpr e1
"(4+(3*7))"
*Parser> showExpr 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
   ```

   [1 Mark]

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 (the complete list of operator symbols is defined in `ops`), form the token `TOp op` from it.
   - 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`. In both cases you now need to split the input string at the next *non alphanumeric* character; equivalently, at the next whitespace or operator character; you might consider using the built-in `break` function to do this. This will give 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 extracted 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 "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 ')']
```

[**8 Marks**]

4. Using `tokenise`, define a 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 Marks**]

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:

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

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

5. Define the function `parse :: [Token] -> ExprStack -> OpStack -> Expr` by encoding the rules outlined in Section **??** above. For example:

```
*Parser> s1
"1+7*9"
*Parser> expParser s1
EApp '+' (ENum 1) (EApp '*' (ENum 7) (ENum 9))
*Parser> s2
```

```
"4+x^2-8*y"
*Parser> showExpr $ expParser s2
"((4+(x^2))-(8*y))"
*Parser> expParser s2 == e2
True
```

[6 Marks]

6. **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 `parse` function to handle bracketed expressions.

   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. How should you handle the `TOp ')'` token? For example,

```
*Parser> expParser "(((6)))"
ENum 6
*Parser> s4
"(4+x)^(2-8)*y"
*Parser> expParser s4
EApp '*' (EApp '^' (EApp '+' (ENum 4) (EVar "x")) (EApp '-' (ENum 2) (ENum 8)))
(EVar "y")
*Parser> expParser s4 == e4
True
```

   [1 Mark]