INF-122 Compulsory assignment 1, fall 2017

- On 122-page, clicking on "Oppgåver" and then "Obligatorisk oppgave 1" opens the page of the assignment. Choosing (on this page, again) "Obligatorisk oppgave 1" gives you the text of the assignment. The solution can be uploaded using the "Lever" button.

  Read the entire exercise set before you start working out your solutions.

- It is allowed to discuss the approach and possible solutions, but everybody has to write an independent solution on his/her own. Suspicious repetitions of code pieces in distinct answers will qualify as an immediate fail. (What counts as a suspicious repetition is decided by the person evaluating the solutions.)

- Questions about the assignment can be asked at the workshops: 28-29.09, 2.10, 5-6.10 or 9.10. The next lecture will be on Monday, 9.10.

- Submit your solutions not later than **Monday, 9th October 2017, at 12:00**. Solutions submitted after that will not be considered.

- Solution must be submitted as a plain-text, interpretable file with the name **Oblig1.hs** (both the name Oblig1 and the suffix .hs are essential). The file must start with
  – a comment containing your name, followed by
  – `module Oblig1 where`, which is the opening part of the code and
  – after which you write your program in the usual way.

- There are no second chances and your solution must be accepted at the first trial.

1 A simple calculator language

Consider the following grammar for a simple, prefix calculator language:

```
num ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
int ::= num | num int
expr ::= int | - expr | + expr expr | * expr expr
```

Any number of whitespaces can be used between any two tokens, but you decide if some tokens can be written without whitespace. E.g., `* 5 5` must be correct, while `*5 5` may – but need not – be. The expression `* 55` is not legal, missing another `expr`. Zeros are allowed in front of numbers, e.g., `0012` is a legal representation of 12.

You will parse and evaluate expressions in a language extending slightly this one. We begin with this simplified sublanguage to ease presentation and, presumably, your programming task. The operators `-`, `+` and `*` have their expected meaning so, for instance:

```
+ 5 5
```

should evaluate to `10`, while

```
* + 1 2 * 3 + 7 - 2,
```

representing `((1+2) * 3) * (3 * (7-2))`, should evaluate to `45`.

You will also implement another evaluation pattern, with odd `int` interpreted as boolean True, even ones as False, + as boolean 'or', * as 'and', - as negation. Then, `+ 5 5`, corresponding to

'True or True', should evaluate to `True`, `* 5 4`, corresponding to 'True and False' should evaluate to False, and `* + 1 2 * 3 + 7 - 2` to True.

Important:

- Don't change the grammar, even if you don't like it.
- Names and types of data constructors and functions which are specified in the text below **must be exactly the same** in your solution.
- We will test your solution using HUnit (`hackage.haskell.org/package/HUnit`) and you are encouraged (but not required) to do it also yourself.

2 Preliminaries

Your task is stated in Section 3 and you can start programming directly what is required there. But following this Section will lead you towards a solution through a number of manageable steps.

2.1 Preliminary parser

It is not allowed to use the `Parsing` module (Hutton, chapter 13).

Write a function `parse :: String -> Ast` which takes a program (a string) as argument and produces an abstract syntax tree (AST) for the program if it is valid, which you can assume it is (with one possible exception, described in part 3). The datatype for ASTs must have the following format and names of constructors (fill in the missing parts) :

```
data Ast = Nr Int | Sum Ast Ast | Mul ...  | Min ...  deriving (Eq, Show)
```

For instance:

```
> parse "+ 4 5"                > parse "+ 4 * 8 - 5"
Sum (Nr 4) (Nr 5)               Sum (Nr 4) (Mul (Nr 8) (Min (Nr 5)))
```

Hints

- You can test if a character is an uppercase letter (representing a variable in Section 3), or a digit, by using the function `isUpper :: Char -> Bool`, or `isDigit :: Char -> Bool`, from the module `Data.Char`.
- You can read in an integer (with leading zeros) using the function `read :: String -> Int` from the Haskell prelude. To find the integer in a prefix of a string `str`, you can do something like `read (takeWhile isDigit str) :: Int`.
- To check if your character list starts with some string (e.g. `let`), you can use patterns defining the function, e.g., something like: `parseExpr ('l':'e':'t':s) =` ...
- To do the actual parsing, you can write a function `parseExpr :: String -> (Ast, String)` taking a string as input and returning an AST fragment together with whatever portion of the string remains to be parsed. To parse subexpressions, just call `parseExpr` recursively.
- Your `parse` function will probably be just a wrapper around `parseExpr`.

2.2 Preliminary evaluation

You shall program two evaluation functions `evi::String -> Int` and `evb::String -> Bool`. The implementation should, however, contain only one main function doing the "real job" on the `Ast`, with both these functions just calling appropriately the main one. But the requirements below are stated only in terms of the user functions `evi` and `evb`.

Evaluation of arithmetic expressions is as one should expect. For example:

```
> evi "+ 5 5"                      > evi "+ 4 * 8 - 5"
10                                 -36
```

For boolean evaluation, we interpret the `int` constructor of `exp` as False, when its argument is even and as True when it is odd, e.g.:

```
> evb "8"                          > evb "7"
False                              True
```

Then + represents boolean 'or', * 'and' and - boolean negation. E.g.:

```
> evb "+ 5 3"                      > evb "+ 4 * 8 - 5"
True                               False.
```

Note that you should convert to Bool only the `int` constructors; on the left only `5` and `3`, not the result of their arithmetic evaluation, i.e., not $5 + 3 = 8$ which, being even, would represent False.

2.3 Adding conditional expressions

Add to the grammar the `if`-expressions, namely:

```
expr ::= ...  (as before) ...  | if expr expr expr
```

and extend your parser and evluator to handle such expressions. To `Ast`, add the constructor

```
data Ast = ...  (as before) ...  | If Ast Ast Ast.
```

Evaluation of an `if`-expression amounts to evaluating its first argument and, when it is `0`, returning the result of evaluating the second argument, while if it is not `0`, returning the result of evaluating its third argument. E.g., evaluation of the (AST for the) expression

```
 * if + 1 2 * 3 4 7 - 2,
```

representing `(if 1+2=0 then 3*4 else 7) * -2`, should return `-14`.

```
> evi "* if + 1 2 7 * if 2 5 3 4 + 2 - 3"
-12.
```

For boolean evaluation, the meaning of the the condition is as expected. If the first argument of `if`-expression evaluates to `True`, the result of evaluating the second one is returned and, otherwise, the result of the third one. E.g.,

```
> evb "+ if + 1 2 * 3 4 7 - 2"        > evb "* if + 1 2 7 * if 2 5 3 4 + 2 - 3"
True                                  False.
```

3 The assignment is to implement

– parsing for the grammar below (extending the language from Sections 1 and 2.3),
– evaluation (`evi` and `evb`) reflecting its semantics as described earlier and below.

NOTE! If you programmed solutions to problems 2.1, 2.2 and 2.3, producing now the final solution may require adding some arguments, needed to handle the variable bindings.

The complete grammar is as follows:

```
num ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
int ::= num | num int
var ::= A | B | C | ...  | Z
expr ::= var | int | - expr | + expr expr | * expr expr | if expr expr expr
       | let var = expr in expr
```

Note that variables are single uppercase letters. (Whitespace must be still admissible between any two tokens, e.g., `let A = 3 in let B = 4 in + A B` must be correct, even if you make also `let A=3in letB=4in+AB` correct.) To `Ast`, add the constructors

```
data Ast = ...  (as before) ...  | Let String Ast Ast | Var String.
```

The `let`-expression manipulates contents of the memory which may contain up to 26 cells (one for each upper case letter). The expression

```
let A = expr1 in expr2
```

evaluates expression `expr1`, stores the obtained value in the memory cell `A`, and then returns the result of evaluation of `expr2`, in the environment where `A` can be accessed, evaluating to the value stored in this cell. The grammar allows nesting of `let`, e.g.:

```
let A = 2 in let B = 3 in let A = 4 in * A B
```

should evaluate to `12`, because the outermost binding `A=2` becomes overshadowed by the innermost one `A=4`. The following expression

```
let A = 2 in * let B = 3 in let A = 4 in * A B A,
```

representing `let A=2 in (let B=3;A=4 in A*B) * A`, should evaluate to `24`, since the last `A` is bound by the outermost `A=2`.

Boolean evaluation happens in the same way, with the variables storing elements of type Bool instead of Int. Thus for instance

```
> evb "* let A = 2 in + * A 2 A 3"          > evb "+ let A = 2 in + * A 2 A 3"
False                                       True.
```

Reporting undeclared variables

Variables are only bound for the expression directly following the `in` keyword. If, for example, a `let`-expression occurs in the first argument to the `*`-operator, that binding should not be considered while evaluating the second argument to this operator. Thus, for instance, the expression

```
* let A = 2 in + * A 2 A 3,
```

representing `(let A=2 in (A*2)+A) * 3`, should evaluate to `18` (or `False`), since all occurrences of `A` are in the scope of its binding. But in the expression

```
+ let A = 2 in + A 2 * A 3,
```

that is, `(let A=2 in A+2) + (A * 3)`, the occurrence of `A` in the second argument, `* A 3`, of the initial `+` is outside the scope of `A`'s binding. This expression is therefore not a legal program.

Your parser (i.e., the `parse` or `parseExpr` function, and not `evi`, `evb` or some function supporting evaluation) must handle such cases when unbound variables appear in the AST, providing an appropriate error message. This can be done, for instance, by postprocessing the AST constructed by the parser, with a function that checks that all variables are correctly bound and raises an error when they are not. Alternatively, such checking can be performed while parsing the expression, but then the parser needs additional argument collecting the declared variables. It is desirable to obtain best possible feedback. As the minimum, you must provide a list of undeclared variables, but the error message may also try to specify, for instance, where they occurr. Explain briefly your solution to this problem in a comment.

Hints

- The state of bound variables can be represented by a list of variables and values paired in tuples, which is passed as an extra argument to all functions using memory (or checking declared variables). Such a list of bound variables is often called an environment in functional programming circles, or a symbol table in compiler literature.
- Preferably, program one HO function for traversing the AST recursively, passing along the environment on each call. The `let`-expression adds to the list it sends to its subtree; a `var`-expression constitutes a use of a variable, so check if it is bound by searching the environment.

  The `evi` and `evb` functions are then wrappers around this function (doing the main job), which specify all function parameters and instantiate the initial environment to `[]`.
- To evaluate `let X = expr1 in expr2`, first evaluate `expr1`, getting a value `v`, then evaluate `expr2` in the environment extended with the extra binding `(X,v)`.
- **The whole program should not have more than around 50 lines.**

  (This is only a hint – not any requirement.)

Sample programs

```
> parse "* 2 + 3 4"
Mul (Nr 2) (Sum (Nr 3) (Nr 4))

> parse "let X = + 1 2 in * X + 2 - X")
Let "X" (Sum (Nr 1) (Nr 2)) (Mul (Var "X") (Sum (Nr 2) (Min (Var "X"))))

> evi "let X = + 1 2 in * X + 2 - X"
-3
> evb "let X = + 1 2 in * X + 2 - X"
False

> evi "let X = 1 in let X = 2 in X"
2
> evb "let X = 1 in let X = 2 in X"
False

> evi "let X = + 1 2 in if X + 1 X * 2 X"
6

> evb "let X = + 1 2 in if X + 1 X * 2 X"
True

> evi "* let X = 3 in + X 1 2"
8
> evb "* let X = 3 in + X 1 2"
False

> evi "* let X = 3 in + X 1 X"
```

An appropriate `error` message about the undeclared (last) `X`.