Teorija programskih jezikov 2017–18

Homework 1: Implementing Imp

Version of March 14, 2018

This goal of this homework exercise is to implement implement a lexer, parser and evaluator for the simple imperative language Imp from Note 3. To achieve this, you have to complete three files, for which templates are provided for you. These files are: `Imp_Lex.x`, `Imp_Parse.y` and `Imp_Evaluator.hs`.

You should complete these three files and submit them by the deadline of **10.00** on **Wednesday 28th March 2018**. Philipp will send round precise submission instructions in due course.

1 Lexing and parsing [12 marks]

Recall the abstract syntax of Imp from Note 3:

$$\begin{aligned}
\mathit{AExp} &::= l \mid n \mid \mathit{AExp}\ \texttt{+}\ \mathit{AExp} \mid \mathit{AExp}\ \texttt{-}\ \mathit{AExp} \mid \mathit{AExp}\ \texttt{*}\ \mathit{AExp} \\
\mathit{BExp} &::= b \mid \mathit{AExp}\ \texttt{==}\ \mathit{AExp} \mid \mathit{AExp}\ \texttt{<}\ \mathit{AExp} \\
&\mid \mathit{BExp}\ \texttt{\&\&}\ \mathit{BExp} \mid \mathit{BExp}\ \texttt{||}\ \mathit{BExp} \mid \texttt{!}\mathit{BExp} \\
\mathit{Com} &::= l \texttt{:=} \mathit{AExp} \mid \texttt{if}\ \mathit{BExp}\ \texttt{then}\ \mathit{Com}\ \texttt{else}\ \mathit{Com} \\
&\mid \mathit{Com}\ \texttt{;}\ \mathit{Com} \mid \texttt{skip} \mid \texttt{while}\ \mathit{BExp}\ \texttt{do}\ \mathit{Com}
\end{aligned}$$

In implementing a concrete syntax, you need to make several decisions. For example: What is the lexical specification for location names (identifiers)? What are the appropriate choices for precedence and associativity of operations? Similarly, how are the potential parsing ambiguities involving combinations of sequential composition (`;`) with the `while` and `if` constructions to be resolved?

To help you make such decisions, two Imp programs, `factorial.imp` and `primes.imp`, are included in the homework folder. Your lexer and parser should correctly handle these programs. In addition, you should obey the following rules.

- Location names (identifiers) should always begin with a capital letter (from `A` to `Z`) in the English alphabet.

- Round brackets `(   )` should be supported as a scoping facility in arithmetic and boolean expressions.

- Curly brackets { } should be supported as a scoping facility in commands.

The specific tasks you need to do are the following.

1. In `Imp_Lex.x` you need to implement the lexical specification of the language, using Haskell's Alex lexer. This means defining patterns for all the required lexical classes, and also actions for translating patterns to lexical tokens. In order to do the latter, you will need to implement a Haskell datatype `Token` for the lexer to use as tokens. The Alex lexer will translate your specification into a function `imp_lex :: String -> [Token]`, which lexes the text of an Imp program into the corresponding list of tokens.

2. In `Imp_Parse.y` you need to define the full grammar for Imp. This means specifying the terminal symbols, specifying the rules (productions) of the grammar, and specifying all required precedence and associativity conventions. The resulting grammar must be unambiguous. That is, the Happy parser generator must not complain about ambiguities when applied to `Imp_Parse.y` (for example, it must not mention any "shift/reduce conflicts").

   In addition, each rule of the grammar needs to be given an associated action that serves to construct the abstract syntax tree resulting from a correct parsing. The homework folder contains a Haskell module `Imp_AbsSyntax` that specifies three Haskell datatypes, `AExp`, `BExp` and `Com`, which provide notation for abstract syntax trees for arithmetic expressions, boolean expressions and commands respectively. Your actions in `Imp_Parse.y` *must* construct abstract syntax trees as elements of these predefined datatypes.

For example, the program text below (note that the curly brackets are necessary!),

```
Y := 1 ; while 0 < X do { Y := Y * X ; X := X - 1 }
```

which is in the file `factorial.imp`, should get converted, by the lexer and parser, into the following element of datatype `Com` representing its abstract syntax tree.

```
Seq(Assign ("Y",  Num 1),
    While (LessThan (Num 0, Loc "X"),
          Seq (Assign ("Y", Times (Loc "Y", Loc "X")),
               Assign ("X", Minus (Loc "X", Num 1)) ) ) )
```

The successful application of Alex to `Imp_Lex.x` and Happy to `Imp_Parse.y` will generate Haskell modules `Imp_Lex` and `Imp_Parse` respectively. These are required by the module `Imp_Interpreter` discussed in Section 3 below.

2 Evaluation [8 marks]

This part of the assignment can be done independently of the previous part. That is, the evaluation functions described below can be programmed without first completing the lexer and parser. However, it will be more awkward to test them, since the lexer and parser are required by the interpreter described in Section 3 below.

The file `Imp_Evaluator.hs` contains type declarations for three functions

```
evalAExp :: State -> AExp -> Integer
evalBExp :: State -> BExp -> Bool
evalCom :: State -> Com -> State
```

Your task is to implement the three functions. Here, `AExp`, `BExp` and `Com` are the datatypes for abstract syntax trees, which are defined in the module `Imp_AbsSyntax`, as mentioned above. The type *State* is implemented in the module `Imp_State`. It is an *abstract datatype*, which means that manipulation of state can only be performed via the *interface* that is *exported* from the `Imp_State` module. What this means, in practice, is that all manipulation of state has to be performed using the value and functions below.

```
emp :: State
valof :: State -> String -> Integer
update :: State -> String -> Integer -> State
```

Here: `emp` defines the empty state (in which no locations have values assigned to them); `valof s l` finds the value stored in location `l` in state `s`; and `update s l n` constructs the new state `s`[`l` $\mapsto$ `n`] defined in the lecture notes. (See Section 3 below for an example using these functions.)

In order to implement the functions `evalAExp`, `evalBExp`, `evalCom` in Haskell, your definitions should follow very closely the structural operational semantics of Imp from lecture note 3.

3 Executing Imp code

A Haskell module `Imp_Interpreter` is provided for you. It can be used only after you have successfully completed all three of the modules `Imp_Lex`, `Imp_Parse` and

`Imp_Interpreter`. The module implements a very simple interpreter for Imp. The dialogue below illustrates how it can be used.

```
$ ghci
GHCi, version *.*.*: http://www.haskell.org/ghc/
Prelude> :load "Imp_Interpreter"
*Imp_Interpreter> let s5 = update emp "X" 5
*Imp_Interpreter> let s100 = update s5 "X" 100
*Imp_Interpreter> s5
State [("X",5)]
*Imp_Interpreter> s100
State [("X",100)]
*Imp_Interpreter> runImp "factorial.imp" s5
State [("X",0),("Y",120)]
*Imp_Interpreter>  runImp "primes.imp" s5
State [("Count",5),("Y",5),("Z",5),("FoundFactors",0),("XthPrime",11),("X",5)]
*Imp_Interpreter> runImp "primes.imp" s100
State [("Count",100),("Y",25),("Z",25),("FoundFactors",0),("XthPrime",541),("X",100)]
```

The implemented interpreter is very crude. The program in the loaded file is lexed and parsed, again and again, every time it is run. Also, states have to be defined from first principles, as above. Feel free to implement your own more advanced interpreter if you wish; but doing so not part of the homework exercise.

4 Overview of logistics

The folder for `Homework1` (which is on the course webpage in both zipped and unzipped formats) contains the following eight files.

`Imp_AbsSyntax.hs`	`Imp_Lex.x`	`factorial.imp`
`Imp_Evaluator.hs`	`Imp_Parse.y`	`primes.imp`
`Imp_Interpreter.hs`	`Imp_State.hs`	

You are asked to complete the three files `Imp_Lex.x`, `Imp_Parse.y` and `Imp_Evaluator.hs` and submit *these three completed files only*, before the deadline.

In the files you submit, your code should be appropriately commented. Your submitted files should contain only code that is relevant to the homework exercise. You should not alter the files `Imp_AbsSyntax.hs` and `Imp_State.hs`. You may, if you wish, alter the file `Imp_Interpreter.hs`; for example, if you wish to improve the functionality of the interpreter. But doing this is not part of the assessment, and the file should not be submitted.

The homework exercise will be given a mark out of a maximum of 20. This mark can contribute 20% towards your course mark. It will be counted towards your course mark if doing so is of benefit to you.