22c:111 (CS:3820) Programming Language Concepts
Fall 2014

Homework Assignment 3

Due: Monday, Nov 10 at 11.59pm

1 Roman Numerals (16 points)

Roman numerals, as used today, are based on seven symbols[1]:

Symbol	Value	Symbol	Value
I	1	C	100
V	5	D	500
X	10	M	1,000
L	50		

Numbers are formed by combining symbols and adding the values. So *II* is two ones, i.e. $1 + 1 = 2$, and *XIII* is a ten and three ones, i.e. $10 + 1 + 1 + 1 = 13$. There is no zero in this system, so 207, for example, is *CCVII*, using the symbols for two hundreds, a five and two ones. 1066 is *MLXVI*, one thousand, fifty and ten, a five and a one.

Symbols are placed from left to right in order of value, starting with the largest. However, in a few specific cases, to avoid four characters being repeated in succession (such as *IIII* or *XXXX*) these can be reduced using subtractive notation as follows:

- The numeral *I* can be placed before *V* and *X* to make 4 units (*IV*) and 9 units (*IX*), respectively.
- *X* can be placed before *L* and *C* to make 40 (*XL*) and 90 (*XC*), respectively.
- *C* can be placed before *D* and *M* to make 400 (*CD*) and 900 (*CM*) according to the same pattern

An example using the above rules would be 1904: this is composed of 1 (one thousand), 9 (nine hundreds), 0 (zero tens), and 4 (four units). To write the Roman numeral, each of the non-zero digits should be treated separately. Thus 1,000 = *M*, 900 = *CM*, and 4 = *IV*. Therefore, 1904 is *MCMIV*.

1. Write a context-free grammar for the language of Roman numerals as described above. Give your grammar as the usual four tuple $\mathsf{G} = (\mathsf{N}, \mathsf{T}, \mathsf{P}, \mathsf{S})$ of non-terminals N, terminals T, the set of production rules P, and the start symbol S.

   Notice that you can break any Roman numeral into four separate words that are concatenated in order: the thousands, the hundreds, the tens and the units. Each of those words can be produced with similar rules, where the symbol sets of the words are pairwise disjoint.

   (4 points)

[1]This text is adapted from `http://en.wikipedia.org/wiki/Roman_numerals`

2. Write a lexical analyzer and a syntax analyzer using `ocamllex` and `menhir` that together parse a string that supposedly is a Roman numeral. If the input is a Roman numeral, the syntax analyzer must return its integer value, and otherwise raise an exception.

   In your lexical analyzer let each valid character of a Roman numeral become a separate token, and fail with the `SyntaxError` exception if you encounter a character that is not in a valid Roman numeral. A new line, whitespace and the end of the input all signal the end of the numeral, and you should return the `EOF` token in this case.

   Your syntax analyzer should implement your grammar from the previous question. In syntax-directed translation, there is an action associated to each production rule, which is executed on a reduction from the associated production rule. In functional programming actions of the same head symbol evaluate to a values of the same type.

   Instead of building an abstract syntax tree over the input, which is the most common use of syntax-directed translation, let us evaluate the numeral directly. That is, each of the actions should return the integer value of the numeral seen so far. Let the starting symbol of the parser be `roman`. If there are shift/reduce or reduce/reduce conflicts in your parser, use appropriate mechanisms to resolve them such that `menhir` is able to produce an unambiguous parser.

   Type your code into the template files `romanLexer.mll` and `romanParser.mly` that are provided along with this assignment. You will find two additional files with this assignment, `romanChecker.ml` and `Makefile`. Do not modify or submit these, they are meant to give you a preview on how your submission is assessed. We will use different test cases for the actual grading. Open a terminal window in Linux or Mac OS X, or use the Cygwin terminal in Windows, and type `make test-1` to run some test cases on your files in the same directory. You can run the command `make test-1-v` that will not only give you a summary but also show you which test cases failed if any.

   Before you submit make sure that your code compiles and run `make test-1` at least once. You will get zero points otherwise. If you want to get partial credit for this exercise by submitting only a parser or only a lexer, you have to leave the code in the templates as it is. (12 points)

2 OCaml Type Expressions (32 points)

The following is simplified fragment of OCaml type expressions in EBNF.

```
   ⟨typexpr⟩ ::= '⟨ident⟩
               | int
               | bool
               | (⟨typexpr⟩)
               | ⟨typexpr⟩->⟨typexpr⟩
               | ⟨typexpr⟩[*⟨typexpr⟩]
               | ⟨typeconstr⟩
               | ⟨typexpr⟩⟨typeconstr⟩
               | (⟨typexpr⟩{,⟨typexpr⟩})⟨typeconstr⟩
⟨typeconstr⟩ ::= ⟨ident⟩
     ⟨ident⟩ ::= (a...z){a...z | 0...9}
```

Note that terminal symbols are printed in blue, and an expression surrounded by { and } stands for zero or more repetition of the expression, an expression in { and }+ stands for one or more repetitions, and an expression in [and] stands for zero or one repetitions of the expression.

Examples for valid type expressions in this grammar are:

- `int`
- `bool`
- `'a`
- `int -> bool`
- `int -> 'a list -> bool`
- `int * bool`
- `('a, 'b) trie`

The grammar is ambiguous. The following are some type expressions that have more than one derivation. Also listed is a type expression that the ambiguous expression should be equivalent to.

ambiguous	unambiguous
`int -> int -> int`	`int -> (int -> int)`
`int -> bool list`	`int -> (bool list)`
`'a * 'b trie`	`'a * ('b trie)`
`int -> int * bool`	`int -> (int * bool)`

Write a lexical analyzer and a syntax analyzer for this grammar that parses strings into the following abstract syntax tree. Use the equivalences above to resolve any shift/reduce or reduce/reduce conflicts in the generated parser.

```
type otype =
  | Int
  | Bool
  | TVar of string
  | Arrow of otype * otype
  | Pair of otype * otype
  | Type of string * otype list
```

Type your code into the template files `ocamlLexer.mll` and `ocamlParser.mly`. The supporting file `ocamlType.ml` contains the abstract syntax tree. You will find two additional files with this assignment, `ocamlChecker.ml` and `Makefile`. Do not modify or submit the latter three files, they are meant to give you a preview on how your submission is assessed. We will use different test cases for the actual grading. Open a terminal window in Linux or Mac OS X, or use the Cygwin terminal in Windows, and type `make test-2` to run some test cases on your files in the same directory. You can run the command `make test-2-v` that will not only give you a summary but also show you which test cases failed if any.

Before you submit make sure that your code compiles and run `make test-1` at least once. You will get zero points otherwise. If you want to get partial credit for this exercise by submitting only a parser or only a lexer, you have to leave the code in the templates as it is. (32 points)