CS4215 Programming Language Implementation

Lab task for Week 10
Language Processing with Camlp4

This lab is to be completed within the lab session. You are give 90 minutes to complete it. You must login to loris-88.ddns.comp.nus.edu.sg and complete this assignment in the designated account. The main site for a tutorial on Camlp4 is `http://caml.inria.fr/pub/docs/tutorial-camlp4/`

Camlp4 is a pre-processor for OCaml that can be used to support syntax extension and also to build new languages. Its name "p4" stands for *Pre-Processor-Pretty Printer*. It provides the following syntactic features:

1. A system of extensible *grammars*.
2. A meta-programming feature, known as *quotations*.
3. A revised syntax for OCaml
4. A pretty printing system for OCaml
5. Other features, such as extensible functions and function streams.

In camlp4 system, syntax extension is done through grammar rules which allows a rather general way to extend languages. Camlp4 can thus be used as a stand-alone parser for a new language, or it can be used as a pre-processor to extend the OCaml language system. While the OCaml parser is based on bottom-up LALR parsing, the camlp4 parser is based on top-down recursive descent parsing. These differences are noticeable when dealing with erroneous scenarios. In this closed lab, we will first look at how to construct stand-alone language parser using camlp4, and will later show how to build language extension to OCaml.

Camlp4 extension is achieved primarily via streams and parsers. A stream is a value of the abstract type `'a Stream.t`, while each parser is a function of the type `'a Stream t -> 'b`. Streams are lazy list-like values, where only the first element is available, while the second element is only visible after the first one has been removed from the stream. A stream may be expressed using `[<'3;'1;'4;'5>]`. Parsers are functions that examine streams. A parser may be constructed using `parser [< 'x >] -> x`. It may return three possible values (i) successful outcome (ii) exception `Stream.Failure` which indicates that no first element matches the pattern (iii) exception `Stream.Error` which

indicates that first element match input stream, but not the rest. Examples of these scenarios are shown below:

```
# let p = parser [< '3; '1; '4 >] -> "hey";;
# p [< '3; '1; '4 >];;
string : "hey"
# p [< '3; '1; '4; '1; '5; '9 >];;
string : "hey"
# p [< '1; '1; '4 >];;
Exception: Stream.Failure
# p [< '3; '2; '4 >];;
Exception: Stream.Error ""
```

As a top-down look-ahead parser, such a parser must not use (i) left recursion (ii) no two patterns must start with the same element. Parsers are realization of gammar rules, and may actually be be built from the latter. This could help provide a higher-level description mechanism. Camlp4 thus provided a grammar type which gives a more general way to build parsers, through values of type `Gram.Entry.t`. This is created using function `Gram.Entry.mk` which takes a lexer as its parameter. As a default, one may use the OCaml lexer via `Plexer.gmake()`.

1 Parser for a Simple Language

Let us look at how we may define a grammar form for a simple arithmetic calculator language in `ex1_calc.ml`, as follows:

```
let expression = Gram.Entry.mk "expression"
EXTEND Gram
GLOBAL: expression ;
expression:
 [ [ x = SELF; "+"; y = SELF -> x + y
   | x = SELF; "-"; y = SELF -> x - y
   | x = SELF; "*"; y = SELF -> x * y
   | x = SELF; "/"; y = SELF -> x / y
   | x = INT -> int_of_string x ] ];
END
```

This grammar form is quite general and can be used to generate a parser for a simple calculator language, as follows:

```
let main =
  let _ = print_string "# " in
  let r = Gram.parse_string expression (Loc.mk "<string>")
    (read_line()) in
  print_string ("Result is "^(string_of_int r)^"\n")
```

Take note that the parser generated from this grammar form has function type of form `string Stream.t -> int`, where its input is a stream of string tokens. We can compile this parser with the help of camlp4, as follows:

```
ocamlc.opt -annot -I +camlp4 camlp4lib.cma -pp camlp4of.opt \
   -c -g ex1_calc.ml
```

We may also run this parser interactively, as follows:

```
camlp4 -parser ex1_calc.cmo
# 3 + 8 *2
Result is 22
```

The current parser is rather simple since it uses the same precedence for all operators. Moreover, it uses left-associativity in its parsing of infix operators.
TASK : Extend the calculator language to support negation (`~`) and exponentiation (`^`). Use a multi-level grammar that is supported by Camlp4 of the form below. Use precedence rule that is similar to the C language.

```
expression:
  [ "add"  LEFTA
    [ .. ]
  | "mul"  LEFTA
    [ .. ]
  | "simple" NONA
    [ .. ]
  ];
END
```

Note that each entry lower in the grammar list is expected to have a higher precedence since it would bind more tightly. Moreover, one may apply either left or right associativity using `LEFTA` and `RIGHTA`, respectively.

2 Code Generation via OCaml

As Camlp4 is a pre-processor for OCaml, we can generate a corresponding OCaml code from our little language. This would provide a quick way for code generation that leverages on the infrastructure of OCaml compilation system. This OCaml code could subsequently be compiled and even executed. Let us look at how this may be done using a simple vector language. As a simple example, we may have a simple expression `[1,2]*3`. We can generate the following OCaml code.

```
# [1,2]*3
let res = List.map (fun a -> a *. 3.) [ 1.; 2. ]
in print_float res;;
```

This code is not type-correct, as we are assuming that `print_float` is a generic printer that would also work for both float and list of float. Nevertheless, one way to achieve this is by building the corresponding AST of OCaml. However, this is often rather cumbersome. For example, if we wish to generate the OCaml expression `let a = b in c`, we may have to build the OCaml AST directly using:

```
MLast.ExLet
 (loc, false, [MLast.PaLid (loc, "a"), MLast.ExLid (loc, "b")],
    MLast.ExLid (loc, "c"))
```

However, this is rather tedious. Fortunately, Camlp4 has a quotation system that allow us to construct AST with the help of syntactic sugar. In this case, we can actually built an AST, as follows:

```
<:expr< let a = b in c >>
```

We specify a grammar category `expr` for which we are to built its corresponding AST. In the case of our vector example, we could use the following quotation code to generate AST to represent an input `[1,2]*3`

```
<:expr< List.map ( fun a -> a *. $ast_v2$ ) $ast_v1$ >>
```

where `$ast_v2$` is the scalar `3` from the second argument, while `$ast_v1$` is the vector value `[1,2]` from the first argument. The notation `<:expr<...>>` is a *quotation* to convert a string into its corresponding AST representation using grammar rule for entry `expr`. The notation `$..$` is an *anti-quotation* that can be used inside a quotation to convert an AST to its string representation.
TASK : Complete `ex2_code_gen.ml` so that a simple OCaml code may be generated. You do not need to worry about type-safety. You may do so by adding code to two places with exception `failwith "to be implemented"` (with guidance given) by suitable OCaml quotation codes.

3 Generating Type-Safe OCaml Program

Copy your solution for `ex2_code_gen.ml` to `ex3_code_gen.ml`. Rewrite your implementation in `ex3_code_gen.ml` so that it generates type-safe OCaml code. We have provided a shell script `ex3e.sh` to execute and then compile the generated program. This code will also take methods that you pre-declare at `m1header.txt`. Two test runs are shown. The first one has a type error,

```
./ex3e.sh
> 1 * [2,3]
> File "m1.ml", line 2, characters 64-67:
> Error: This expression has type float list
>       but an expression was expected of type float

./ex3e.sh
> 2 * 4
```