IB Compiler Construction - Supervision 1

Nandor Licker <nl364@cl.cam.ac.uk>

Due noon before the day of supervision

1. Consider the following example from some awful programming language which shall remain unnamed:

```
function test(a, b, c) {
  return a + b / c.length - 123.0;
}
function some_call(c, d) {
  z = c + c;
  return test(3, z, "\"hello" + d);
}
some_call(5, "world\"");
```

   (a) Enumerate all the tokens, grouping them by their type.

   (b) Draw the automatons which accept the following tokens: numbers, identifiers and strings.

   (c) Provide regular expressions which match numbers, identifiers and strings.

   (d) Constructs the NFAs of the previous regular expressions and reduce them to DFAs.

   (e) Draw the automaton of the entire lexer, with states to accept all tokens of the language.

   (f) Implement the lexer in `ocamllex`.

2. In some languages, floating point numbers can be defined in multiple ways. The following are examples of valid numeric constants in C++11:

   - `0b101`
   - `0x123`
   - `0233`
   - `1234`
   - `-123.5`
   - `123.5f`
   - `1.3e-20`
   - `2.3e+10`
   - `1e1f`

   (a) Provide a regular expression for the following: binary, decimal, hexadecimal, float and double.

   (b) Draw an automaton with accepting states for the following tokens, capable of parsing numbers similar to the ones shown before, with 3 accepting states: `INT`, `FLOAT`, `DOUBLE`.

   (c) Can you design a regex to only accept 32-bit signed integers? Sketch a solution. How about 32-bit floats? Explain why this might not be practical. How would you handle this in `ocamllex`?

   (d) At what stage in the compiler should an error be emitted when a numeric constant is too large? Discuss the following languages: C/C++, Java, OCaml and JavaScript.

3. This question concerns parsing and representations.

   (a) Define the Abstract Syntax Tree (AST) of the awful language mentioned before in OCaml, using records and variants. What are the options of representing function bodies and statements? Make sure the AST can represent first-class functions and closures.

   (b) Draw the parse tree and the AST of the test function:

   ```
   function test(a, b, c) {
     return a + b / c.length - 123.0;
   }
   ```

   (c) Provide the EBNF grammar for a parser.

   (d) Sketch the code of a recursive descent parser for this grammar.

4. The awful language mentioned before also supports regular expressions with the following syntax:

   ```
   function is_aaa(str) {
     return /aaa/.match(str);
   }
   ```

   (a) Write a function which contains the string `/aaa/`, but does not contain any regexes.

   (b) Can you tokenize the stream by simply adding a rule to the previous lexer? Explain why not.

   (c) Show how the recursive descent parser would interact with a modified lexer to parse this language.

5. Provide the small-step semantics of the language, with eager evaluation, passing arguments from left-to-right. Discuss any other assumptions. Show how the following example reduces to a single value:

   ```
   function f(x) {
     return function(z) {
       return x + z;
     };
   };
   f(2)(3)
   ```