SC-T-501-FMAL Programming languages, Assignment 2
Spring 2020
Due 29 Feb 2020 at 23:59

For this assignment, you will need to modify the file Assignment2.fs. This file contains definitions of the types used below and some helper functions you might find useful. It also contains some incomplete code that you need to complete and some code that is complete and type-correct, but does not exactly work as it should, your task then being to fix it. At the bottom of the file, there are also tests that you can use to check your answers.

1. Consider a language of expressions for records that contain integers, such as

   ```
   { x = 1; y = 2 }
   { x = 1; y = { y1 = 2; y2 = 3 } }
   { x = 1; y = { y1 = 2; y2 = x } }
   { x = 1; y = { y1 = 2; y2 = x }; a = {b = {c = 1}} }
   ```

   A record expression contains zero or more fields, separated by semicolons. Each field has a name, and a contents expression, which is either an integer constant, the name of another field, or a record expression.

   The code for the assignment contains a lexer for this language. Complete the definition of the function `tokenize : char list -> token list`.

2. The code for the assignment contains a parser `parse : token list -> expr` that correctly parses record expressions where all fields are integers, such as `{x = 1; y = 2; z = x}`, but fails on record expressions with nesting, such as `{x = 1; y = {z = 2; w = z}}`. Fix the parser to work correctly on all inputs.

   Hint: You need to change just one of the functions `parseExpr`, `parseFields` and `parseContents`.

3. Record expressions can be evaluated into record values by replacing field names in contents expressions = with their values. Fields can refer to other fields that appear to the left and in the same or an outer scope. For example:

   - `{x = 1; y = x}` should evaluate to `{x = 1; y = 1}`;
   - `{x = 1; y = {x = 2}; z = x}` should evaluate to `{x = 1; y = {x = 2}; z = 1}` (the middle `x` is in an inner scope, and therefore not visible in the definition of `z`);
   - `{x = {w = 1}; y = {z = x}}` should evaluate to `{x = {w = 1}; y = {z = {w = 1}}}`.

   (a) Evaluate the following:

   ```
   {x = {w = 1; z = 2}; y = x}
   {x = 20; y = {w = x}; z = {a = y}}
   {x = 10; y = {x = 11; z = 12}; z = x}
   ```

   You can do the evaluation manually or rely on your solution to the next subproblem.

   Write your answers in Assignment2.fs, in the designated section below `// Problem 3 (i)`.

   (b) The provided code defines an evaluator `eval : expr -> contentsValue envir -> value`. However, it is slightly broken (try it on the examples to see why). Fix it to work correctly. You may need to change `eval` or `evalContents` or both.

   Hint: The change needed is very small.

4. Write a type inference function `infer : expr -> contentsType envir -> typ` for the record language.

   Hint: Recall that type inference should be similar to evaluation.

5. The following F# function sums the elements of a list of integers, raising an exception if any of the elements are negative:

```
let sum xs =
    let rec sumCont xs =
        match xs with
        | [] -> k 0
        | x::xs ->
            if x < 0
            then failwith "Negative" // Change this line
            else sumCont (fun s -> k (s + x)) xs
    sumCont (fun s -> s) xs
```

By changing *only* the indicated line, make it so that the function sums the elements of the list up to and including the first negative element, ignoring the rest. (If none of the elements are negative the result should just be the sum as usual.)

Hint: think about how the argument `k` evolves in the following example:

```
sum [1; 2; 3; -1] ---> sumCont (fun s -> s) [1; 2; 3; -1]
                  ---> sumCont (fun s -> s + 1) [2; 3; -1]
                  ---> sumCont (fun s -> s + 2 + 1) [3; -1]
                  ---> sumCont (fun s -> s + 3 + 2 + 1) [-1]
```