SC-T-501-FMAL Programming languages, Assignment 2
Spring 2022
Due 28 Feb 2022 at 23:59

Write your solutions in a single F# file named `Assignment2.fs`. Follow the template provided. The file must contain your names, and must contain the line `module Assignment2`. Do not change the names of the functions, their type signatures, the order of the arguments, or any of the `type` declarations. Top-level helper functions are OK, you can call them what you want. F# must process your file without errors, for example, doing

```
#load "Assignment2.fs";;
open Assignment2;;
```

inside the interpreter should succeed (warnings are OK). The file `Assignment2Tests.fs` contains some test cases that you can use on your code.

The type `expr` in `Assignment2.fs` represents the abstract syntax of a simple language with

- numbers (`Num`) and operations on them (`Plus` and `Times`);
- pairs (`Pair`), which can be nested;
- variables (`Var`), which can contain numbers or pairs;
- pattern-matching non-recursive let bindings (`Let`), which can be used to split pairs into their components.

The patterns allowed in let bindings are underscore (`PUnderscore`, which matches everything), variables (`PVar`) or pairs (`PPair`). For example, the abstract syntax

```
Let
  (PVar "x", Pair (Num 1, Num 2),
   Let (PPair (PVar "y", PUnderscore), Var "x", Plus (Var "y", Num 5)))
```

corresponds to the F# expression

```
let x = 1, 2 in let y, _ = x in y + 5
```

The only tuples allowed are pairs, so for example there is no direct equivalent of the F# expression

```
let x = 1, 2, 3 in let y, z, w = x in y * z + w
```

involving triples. But, using nested pairs, the F# expression

```
let x = (1, 2), 3 in let (y, z), w = x in y * z + w
```

can be rendered in the abstract syntax as

```
Let
  (PVar "x", Pair (Pair (Num 1, Num 2), Num 3),
   Let
     (PPair (PPair (PVar "y", PVar "z"), PVar "w"), Var "x",
      Plus (Times (Var "y", Var "z"), Var "w")))
```

In the concrete syntax, parentheses are needed around a pair, except when this is necessary for disambiguation (like in F#). For example, the following two expressions are the same when converted to abstract syntax:

```
let x = (1, 2), 3 in let (y, z), w = x in y * z + w
let x = ((1, 2), 3) in let ((y, z), w) = x in (y * z) + w
```

(You can use the function `prettyprint` to see what the abstract syntax would look like in F#.)

1. In F#, no variable can appear more than once in a single pattern. For example, the expression

   ```
   let x, (_, x) = (1, (2, 3)) in 4
   ```

   is not allowed, but

   ```
   let x, y = (1, (2, 3)) in let (_, x) = y in 4
   ```

   is OK. Write a function `checkAllPatterns : expr -> bool` that takes an expression, and returns `true` if this rule is followed, `false` if it is not. You can use `checkPattern : pattern -> bool` to do this.

2. `Assignment2.fs` contains an implementation of a lexer for this language, which is broken in that it does not handle commas and underscores. Modify the function `tokenize` so that they are treated too.

3. `Assignment2.fs` also contains a broken implementation of a parser, which will fail to parse a let binding in which the pattern is not just a variable. Change the implementations of `parsePattern` and `parseSimplePattern` so that the other forms of pattern (pairs and the underscore) are parsed correctly. Commas are non-associative, so parsing of `let x, y, z = w in 2` should fail, but `let (x, y), z = w in 2` and `let x, (y, z) = w in 2` are OK.

4. `Assignment2.fs` contains a partial implementation of an evaluator `eval` for this language, with the `Let` case unimplemented.

   (i) Write a function `patternMatch : pattern -> value -> envir -> envir` that matches the given value against the pattern, and adds the relevant bindings to the environment. Use `failwith` for cases in which pattern matching fails (for example if the value is `VNum 0` and the pattern is `PPair (PUnderscore, PUnderscore)`). You can assume that no variable appears more than once in the pattern (the check in Problem 1 would succeed). No binding should be added to the environment if the pattern is `PUnderscore`.

   (ii) Use `patternMatch` to implement the `Let` case of `eval`.

The type `nexpr` represents the abstract syntax of a language similar to the previous one, except that pattern matching is not allowed in let bindings, but `NFst` and `NSnd` can be used to get the first and second components of a pair (they correspond to `fst` and `snd` in F#).

5. (i) Write a function `nexprToExpr : nexpr -> expr` that converts between the two languages. `NFst` and `NSnd` should be implemented by pattern matching. For every expression `e : nexpr` and environment `env : envir`, the results of `neval e env` and `eval (nexprToExpr e) env` should be the same.

   (ii) We can also convert from `expr` to `nexpr`, by replacing each pattern-matching let binding with several ordinary let bindings that use `NFst` and `NSnd`. For example, we can convert

   ```
   let (x, (y, _)) = p in x * y
   ```

   to

   ```
   let toMatch = p in
     let x = fst toMatch in
       let y = fst (snd toMatch) in x * y
   ```

   The function `exprToNexpr : expr -> nexpr` is intended to do this. Complete the function `bindPattern` used by `exprToNexpr` (you only need to change the `PPair` case). For every expression `e : expr` (that does not have patterns with repeated variables) and environment `env : envir`, the results of `neval (exprToNexpr e) env` and `eval e env` should be the same.

6. `Assignment2.fs` also contains an implementation of a stack machine that supports both numbers and pairs. The stack is a list of integers, and each value (number or pair) has to be encoded as a list of integers on the stack.

   - A number i is encoded as the integer 1 followed by i.
   - A pair $(v1, v2)$ is encoded as the integer 2, followed by the encoding of $v2$, followed by the encoding of $v1$. (Be careful to ensure that the two components of a pair appear in the correct order.)

   For example, the value $(30, 31)$ is encoded as `[2;1;31;1;30]`, and the value $((30, 31), (32, 33))$ is encoded as `[2;2;1;33;1;32;2;1;31;1;30]`. The stack `[1;50;2;2;1;33;1;32;2;1;31;1;30]` contains the number 50 at the top, and the pair $((30, 31), (32, 33))$ below it.

   The stack machine supports some instructions for working with these encodings:

   - `RPair` takes two values from the top of the stack and pairs them. For example, if we execute `RPair` on the stack `[1;62;1;61;1;60]` we get the stack `[2;1;62;1;61;1;60]` (which contains $(61, 62)$ followed by 60).
   - `RUnpair` splits a pair into its two components (the opposite of `RPair`). Executing `RUnpair` on the stack `[2;1;62;1;61;1;60]` results in the stack `[1;62;1;61;1;60]`.
   - `RPop` pops a value from the top of the stack. If we do `RPop` on `[2;1;62;1;61;1;60]`, we get `[1;60]`. If we do `RPop` on `[1;62;1;61;1;60]`, we get `[1;61;1;60]`.
   - `RSwap` swaps the two values at the top of the stack. If we do `RSwap` on `[2;1;62;1;61;1;60]`, we get `[1;60;2;1;62;1;61]`. If we do `RSwap` on `[1;62;1;61;1;60]`, we get `[1;61;1;62;1;60]`.

   (You may find it helpful to uncomment the `printfn` line in `reval`, which will make `reval` print the stack before executing each instruction.)

   (i) The following is a stack of three values. What are these three values?

   ```
   [1;40;2;1;43;2;1;42;1;41;2;2;1;48;1;47;2;1;46;1;45]
   ```

   (ii) What are the encodings of the following values?

   - 701
   - $(701, 702)$
   - $(700, (701, 702))$
   - $((700, (701, 702)), 703)$

   (iii) The function `rcomp` compiles expressions (`nexpr`) to this stack machine. Complete the implementation, by writing functions

   ```
   rcompPair : rcode -> rcode -> rcode
   rcompFst : rcode -> rcode
   rcompSnd : rcode -> rcode
   ```

   You should *only* change these functions, do not change `rcomp` itself. The instructions `RPair`, `RUnpair`, `RPop` and `RSwap` will be useful.