Prof. Dr. Peter Thiemann
Winter Term 2024/25

Hannes Saffrich
saffrich@informatik.uni-freiburg.de

Functional Programming

https://proglang.informatik.uni-freiburg.de/teaching/functional-programming/2024/

Exercise Sheet 10

Exercise 1 (An Interpreter for the Lambda Calculus)

Write an interpreter for the lambda calculus, which reads lambda expressions from the terminal and tries to reduce them to β-η-normal form, i.e. tries to repeatedly apply β- and η-reduction anywhere in the expression until this is no longer possible. It should print the expression after each reduction step, e.g.

```
lambda> ((\x. x) (\y. y))

((\x. x) (\y. y))
(\y. y)

lambda> (((\x. (\y. y)) (\a. a)) (\b. b))

(((\x. (\y. y)) (\a. a)) (\b. b))
((\y. y) (\b. b))
(\b. b)

lambda> (\x. f x)

(\x. f x)
f

lambda> \m/

ERROR: Failed to parse input. Try again!

lambda> ((\x. x x) (\x. x x))

((\x. x x) (\x. x x))
((\x. x x) (\x. x x))
((\x. x x) (\x. x x))
[...]
```

We recommend to proceed as followed:

1. Use the parser combinators from exercise sheet 7 to write the parser. A simple way to get an unambiguous grammar is to require parentheses around both lambda abstractions and applications, e.g.

   ```
   var ::= [a-zA-Z]+
   exp ::= var
         | '(' ws '\' ws var ws '.' ws exp ws ')'
         | '(' ws exp ws1 exp ws ')'
   ```

   where `ws` stands for zero or more whitespaces, and `ws1` stands for one of more whitespaces.

2. Implement a function `free`, which takes an expression and returns a list of its free variables.
3. Implement a function `fresh`, which takes a list of variables `xs`, and returns a new variable, which does not occur in `xs`.
4. Implement capture-avoiding substitution as shown on the slides. You will need to make use of `free` and `fresh`.
5. Implement a function `step`, which tries to apply a single β- or η-reduction to the expression. If the expression itself has the shape suitable for β- or η-reduction, then apply it directly, otherwise try to apply it in the subexpressions. The implementation closely follows the slides.
6. Implement a function `steps`, which tries to apply the `step` function repeatedly to an expression, until no further reduction is possible. The function should return a list of all expressions that occur during reduction, i.e. the first expression is the input expression, and the last expression is the expression at which no further reduction is possible. For non-terminating expressions, this list is infinite.
7. Implement a function `pretty`, which converts an expression back into a string.
8. Finally, implement a REPL (Read-Eval-Print-Loop) as an `IO ()` action, which repeatedly reads a line from the terminal, then tries to parse the line to an expression, then reduces the expression to a normal form while `pretty`-printing each expression that occurs during reduction (as shown in the example). If an expression has no normal form, e.g. the non-terminating expression `((\x. x x) (\x. x x))`, then the interpreter also should not terminate.