# Assignment 1 — Lambda calculus — due Wednesday 4 September

Total number of points available on this project is 150. Full credit is equivalent to 100 points.

## A library of λ-terms

$\mathbf{I} \triangleq \lambda x.x$ $\quad \mathbf{K} \triangleq \lambda xy.x$ $\quad \mathbf{S} \triangleq \lambda fgx.(fx)(gx)$ $\quad \mathbf{B} \triangleq \lambda fgx.f(gx)$ $\quad \mathbf{C} \triangleq \lambda fgx.fxg$

$\omega \triangleq \lambda x.xx$ $\quad \Omega \triangleq \omega\omega$ $\quad \mathbf{Y} \triangleq \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$

$\mathbf{true} \triangleq \lambda xy.x$ $\quad \mathbf{false} \triangleq \lambda xy.y$ $\quad \mathbf{not} \triangleq \lambda t.t\ \mathbf{false}\ \mathbf{true}$ $\quad \mathbf{cond} \triangleq \lambda ee_1e_2.ee_1e_2$

$\mathbf{pair} \triangleq \lambda e_1e_2f.fe_1e_2$ $\quad \mathbf{fst} \triangleq \lambda p.p\ \mathbf{true}$ $\quad \mathbf{snd} \triangleq \lambda p.p\ \mathbf{false}$

$\mathbf{0} \triangleq \lambda fx.x$ $\quad \mathbf{1} \triangleq \lambda fx.fx$ $\quad \mathbf{2} \triangleq \lambda fx.f(fx)$ $\quad \mathbf{succ} \triangleq \lambda nfx.nf(fx)$ $\quad \mathbf{add} \triangleq \lambda mnfx.mf(nfx)$
$\mathbf{iszero} \triangleq \lambda n.n(\lambda x.\mathbf{false})\mathbf{true}$ $\quad \mathbf{prefn} \triangleq \lambda fp.\mathbf{pair}\ \mathbf{false}(\mathbf{cond}(\mathbf{fst}\ p)(\mathbf{snd}\ p)(f(\mathbf{snd} p)))$
$\mathbf{pred} \triangleq \lambda nfx.\mathbf{snd}(n(\mathbf{prefn}\ f)(\mathbf{pair}\ \mathbf{true}\ x))$

$\mathbf{cons} \triangleq \lambda hts.sht$ $\quad \mathbf{hd} \triangleq \lambda L.L\ \mathbf{true}$ $\quad \mathbf{tl} \triangleq \lambda L.L\ \mathbf{false}$ $\quad \mathbf{nil} \triangleq \lambda x.\mathbf{true}$ $\quad \mathbf{isempty} \triangleq \lambda L.L(\lambda ht.\mathbf{false})$

## Normal forms of some λ-terms

$\mathbf{SKK} \twoheadrightarrow \lambda x.x$ $\quad \mathbf{K}(\mathbf{SII}) \twoheadrightarrow \lambda ab.bb$ $\quad \mathbf{S}(\mathbf{S}(\mathbf{KS})(\mathbf{KI}))(\mathbf{KI}) \twoheadrightarrow \lambda ab.bb$
$\mathbf{SSSSSSS} \twoheadrightarrow \lambda ab.(ab(ab(ab\lambda c.ac(bc))))$

## 1.1 A substitution-based λ-calculus interpreter

We've learned that β-reduction of terms of the λ-calculus by hand is tedious and error-prone.

The goal of this project is to develop an interpreter for the λ-calculus that will automate the reductions. This program will follow literally the rules for β and η conversion, and the rules for substitution. The internal representation of λ-terms is essentially the same as the textual representation, though the data type makes the bracketting structure apparent, and pattern-matching easier. We call this kind of system "string rewriting".

We must first specify the internal representation for λ-expressions. We shall use the following type:

> **datatype** *Expr* = *Var* **of** *string* | *Abstraction* **of** *string* ∗ *Expr* | *Application* **of** *Expr* ∗ *Expr*

I am using ML notation throughout; if you use ML as your implementation language, you should strictly adhere to the specifications given here. If you are using a different implementation language, make the appropriate mapping.

Tasks:

1. (5 pts.)

   Write a function *lexpPrint* to convert an *Expr* into a character string in the fully-bracketted syntax for $\lambda$-expressions. Use the character # for $\lambda$.

2. (5 pts.)

   Write a function *lexpPrettyPrint* to convert an *Expr* into a character string in the loose syntax for $\lambda$-expressions. Remove as many parentheses as possible.

3. (10 pts.)

   Write a function *lexpParse* to parse a character string containing the text of a $\lambda$-expression (which may be in the loose syntax!) and convert it into an *Expr*. If you are pressed for time, restrict input to strict syntax.

4. (10 pts.)

   Implement an environment mapping identifiers to $\lambda$-expressions, with type $\mathit{string} \to \mathit{Expr}$. There should be a mechanism to build new environments out of old ones by introducing new definitions for an identifier.

5. (10 pts.)

   Implement a top-level environment and appropriate input/output handling, so that it is possible interactively to add new definitions of named $\lambda$-expressions, and ask for expressions to be reduced and printed. It should be possible to print all intermediate steps of the reduction as well.

6. (40 pts.)

   Implement a generic reduction framework, in which several notions of conversion can be specified, and different reduction orders can be specified. For instance, one should be able to specify that either $\beta$ and $\eta$ conversions, or both, should be tried, and that applicative-order or normal-order reduction should be used.

   Implement $\beta$ and $\eta$ conversions, as well as a conversion that we'll call $\zeta$ that performs the "macro-expansion" of a name defined in the top-level environment.

7. (20 pts.)

   Test your program by reducing these expressions: **SKK**; **K**(**SII**); **S**(**S**(**KS**)(**KI**))(**KI**); **SSSSSSS**.

   Document your design and your program well. Submit program listing and transcript of a sample session including at least the reductions above.

8. (10 pts. extra credit)

   Do what is necessary so that strictness annotations can be added to the $\lambda$-bindings: even if normal-order reduction is otherwise used, $\beta$-conversion of a redex $(\underline{\lambda} v.B)E$ will cause $E$ to be reduced first, before substitution into the body $B$.

9. (10 pts. extra credit)

   Write a function *lexpInterpretivePrettyPrint* to convert an *Expr* into a character string in the loose syntax for $\lambda$-expressions, but in a form that is easier for humans to read, as follows. Recognize subexpressions that are exactly as the Church numerals and the truth values and print them as 0, 1, etc. Also recognize `cons` and `nil` so that lists can be printed as `(cons 2 (cons 1 nil))`, for instance.

10. (10 pts. extra credit)

    Implement $\delta$-conversion rules for arithmetic, and input/output syntax for numbers. Extend the parser to recognize decimal integers and the common arithmetic operators +, -, etc. The operators do not need to be infix. You may rely on the underlying SML type *int*, or a similar machine-precision integer type in your implementation language.

11. (10 pts. extra credit)

    Implement $\delta$-conversion rules for a primitive list type with constructors **cons** and **nil** that can be used to build lists of $\lambda$-terms. Extend the parser to recognize **cons** and **nil**, and extend the pretty-printer to print them.

12. (10 pts. extra credit)

    Extend the parser to recognize and the pretty-printer to print infix arithmetic operators, the infix **cons** (written **::**), and the list notation **[ , ]**.

## How to turn in

Turn in your code by running *~darko/handin your-file* on a regular UNM CS machine or an OAL machine. You should use whatever filename is appropriate in place of your-file. You can put multiple files on the command line, or even directories. Directories will have their entire contents handed in, so please be sure to clean out any cruft.

Remember to submit extensive tests of your programs!

Homework must be accompanied by the following statement: *"I pledge my honor that in the preparation of this assignment I have complied with the University of New Mexico Board of Regents' Policy Manual, including Section 4.8, Academic Dishonesty."* The manual is available at `http://www.unm.edu/~brpm/index.html`.