Assignment 2: The λ-Calculus; Basic Type Theory

CS 442/642

Due January 27^{th}, 5:30pm

Solve the programming portion of this assignment using DrRacket. Set the language to R5RS.

1 A λ-Calculus Interpreter

Using the pure functional subset of Scheme, write an evaluator that reduces pure λ-calculus expressions to β-normal form. The pure functional subset of Scheme excludes all functions that have side-effects, notably those whose names end with an exclamation mark, '`!`'. Submissions that use such functions will receive a mark of 0. λ-expressions will be represented using Scheme data structures as follows:

e ::= <symbol>	(any symbol except `fun` is a variable)
e ::= (`fun` <symbol> e)	(function abstraction)
e ::= (e e)	(function application)

For example, the λ-expression $\lambda x.\lambda z.x(\lambda y.yy)zz$ would be represented as

```
(fun x (fun z (((x (fun y (y y))) z) z)))
```

Note that (for now) λ-expressions are fully parenthesized.

Part A: Download the file `http://www.student.cs.uwaterloo.ca/~cs442/subst.scm`. This file contains a number of utility functions for constructing and deconstructing λ-expressions, and can serve as an example of a reasonable Scheme coding style. It also contains a substitution function for λ-expressions. However, the substitution function implements dynamic binding. You are to modify the substitution function so that it implements *static* binding. Your implementation of the corrected substitution function must exactly match the corrected definition of substitution outlined in class. (Solely for the purpose of generating new variable names, you are allowed a single use of `set!` here, without penalty. You are permitted to "reserve" a class of variable names for internal use; document your choice.)

Part B: Using your solution for Part A as a starting point, write a function `reduce` that takes a λ-term and returns the result of reducing the term to β-normal form. Your interpreter (i.e., the function `reduce`) is to follow the Normal-Order (i.e., lazy) reduction strategy. For example,

```
(reduce '((fun x x) ((fun y y) z))
```

should return

```
'z
```

Part C: Now enhance your interpreter so that it makes no assumptions about how the λ-expression is parenthesized—it may have fewer parentheses than prescribed by the grammar above, or it may have more parentheses. Do this by writing a Scheme function `parse-lambda` that takes an arbitrarily-parenthesized λ-expression and returns the equivalent λ-expression that is parenthesized according to the grammar in the preamble to this question. For example,

`(parse-lambda '(fun x fun z x (fun y y y) z z))`

should return

`'(fun x (fun z (((x (fun y (y y))) z) z)))`

and

`(parse-lambda '(fun x ((x (((x)))))))`

should return

`'(fun x (x x))`

Your completed interpreter is then essentially the function

`(define (interpret E) (reduce (parse-lambda E)))`

Submit your solutions to Parts A, B, and C as a single source file, but clearly indicate the portion that solves each part (and preferably order your program so that part B follows part A, and part C follows part B) so that everything is easy to find. Be sure to name your functions as prescribed above, so that the TA will not have trouble running your program. Include the definition of `interpret`, as given above, in your submission. You are to hand in a listing of the source code of your program on paper, as well as electronically, via `submit`.

Part D (Demonstration): To help convince the marker that your interpreter works, create a λ-calculus implementation of the function `fib`, which computes the n-th Fibonacci number, where `(fib 0)` is `0` and `(fib 1)` is `1`, such that your interpreter will run it. Then test your `fib` function on a few sample numbers, using the λ-calculus representation of natural numbers from class. Submit a transcript of your interpreter's responses to your inputs (print out your interaction with your interpreter).

For extra fun (not to be submitted): Create a λ-calculus implementation of the function `foldl`, such that your interpreter will run it. Then use `foldl` to implement `reverse` and test your reversal function on a few sample lists, all implemented in the λ-calculus. Keep in mind that there are no multi-argument functions in the λ-calculus, so your implementation will have to be curried.

Hints:

- The crux of the problem is to find the leftmost β-redex and to reduce it; then repeat.
- Consider writing a trace facility that prints intermediate results.

2 Eager Evaluation

After you have finished your evaluator, answer the following questions:

1. Assume you are given an evaluator that takes input in the same format as the evaluator you wrote, but reduces it using the Applicative-Order Evaluation (AOE) strategy. Assume also that numbers, and the usual operations on them, have already been defined for you, as they were defined in class. Provide an implmentation of the function `fact`, which takes a single argument and produces its factorial, that would work with the new evaluator.
2. Assume that the evaluator you wrote in Question 1 of this assignment works flawlessly. (We hope this is a valid assumption!) If you were to run your evaluator on the new version of `fact` you just wrote, applied to a numeric argument, would it work? Why or why not? Provide a full explanation. You should not need to actually try it.

3. Are there any circumstances under which your evaluator (assuming again that it works flawlessly) and the AOE evaluator supplied to you, given the same input, would both terminate, and yet return different results (α-equivalent expressions with different bound variable names don't count)? If so, give an example. If not, explain why not.

3 Simple Types

The Church numerals may be given the type $\tau_C = (\text{int} \to \text{int}) \to \text{int} \to \text{int}$.

Part A For each of the first three Church numerals presented below, annotate the abstractions with types, and give a type derivation for each, such that the conclusion assigns each numeral the type τ_C:

- $\lambda f.\lambda x.x$
- $\lambda f.\lambda x.fx$
- $\lambda f.\lambda x.f(fx)$

Part B The Church numeral addition function, $\lambda m.\lambda n.\lambda f.\lambda x.mf(nfx)$, takes two Church numerals and returns the Church numeral representing their sum. Given the type presented above for Church numerals, determine the correct type annotations for the addition function, and give a type derivation for it.

Submission

Questions 2 and 3, on eager evaluation and simple types, are written questions, and should be submitted to Crowdmark only. For Question 1, the λ-calculus interpreter, you should submit **both** a copy of your source to Crowdmark, **and** an electronic copy of your source via `submit`. In addition, for Question 1D, you are to submit, to Crowdmark only, a transcript of your demonstration. It must not exceed two pages in length.

The distribution of marks for this assignment is expected to be approximately 50% for Question 1, 20% for Question 2, and 30% for Question 3.