Computer Science 131, Fall 2001
Assignment 3: Lambda Calculus Implementation

Out: Thursday, September 27

Due: Friday, October 5, 5pm

Process

Read the entire assignment carefully before beginning!

For this assignment, rather than running SML/NJ via the command `sml`, you must instead run `/cs/cs131/bin/sml-cs131-a3` . This copy of the compiler already has built-in all the definitions you will need to do this assignment.

Your SML code should go in a file named `assign3.sml` that includes your answers. Make sure this file starts with a comment including at least your name and an estimate of the time required to complete your solution. Your functions should have the names and types specified, but you are free to define other helper functions whenever this would be useful.

You are also required to create files named `fact3` and `fib4`. Submit all three files using `cs131submit` as usual.

1 Introduction

1.1 Abstract Syntax

For this assignment you will implement code working with λ-calculus expressions. These expressions will be represented in SML as values of type `absyn`, which are predefined in the `sml-cs131-a3` binary as follows:

```
datatype absyn = Var of string
               | App of absyn * absyn
               | Lam of string * absyn
```

This abstract syntax includes variables (represented as strings), applications, and single-argument functions. The binary also contains two helper functions for displaying abstract syntax.

```
val print_raw : absyn -> unit
val print_nice : absyn -> unit
```

The function `print_raw` takes an argument of type `absyn` and prints it in a representation that could be understood by SML. The function `print_nice` takes an argument of type `absyn` and prints it out in a form that should be fairly familar; it looks just like the expressions you have seen in class except that the symbol λ is approximated by a backslash `\`.

As an example, the term $\lambda x.\lambda y.yx$ would be represented in this abstract syntax as

```
Lam("x",Lam("y", App(Var "y", Var "x")))
```

and this exactly what would be printed if this value were passed to to `print_raw`. In contrast, the function `print_nice` would display the more compact form `\x.(\y.(y x))`

1.2 Concrete Syntax

The `sml-cs131-a3` binary also includes two separate parsers that can take a file and interpret the contents as a λ-calculus expression. These two functions are:

```
M.parse : string -> absyn
S.parse : string -> absyn
```

Both functions take a string argument containing a filename, and interpret the contents of that file as a λ-calculus expression. The only difference between the two is the concrete syntax they expect.

1.2.1 First Concrete Syntax

The function `M.parse` expects code to be in an ML-like syntax. This can be described with the following grammar:

⟨exp⟩ ::=	⟨id⟩	*identifier*
|	⟨exp⟩ ⟨exp⟩	*application*
|	`fn` ⟨id⟩ `=>` ⟨exp⟩	*function*
|	`let` ⟨defns⟩ `in` ⟨exp⟩ `end`	*let-expression*
|	`(` ⟨exp⟩ `)`	*parenthesized expression*
⟨defn⟩ ::=	`val` ⟨id⟩ `=` ⟨exp⟩	*single definition*
⟨defns⟩ ::=	ϵ | ⟨defn⟩ ⟨defns⟩	*sequence of definitions*

This grammar uses ⟨id⟩ to represent an `identifier` or variable which for this assignment can be any sequence of letters and numbers starting with a letter; unlike SML, underscores and primes are *not* allowed. Because this grammar is ambiguous, we add the usual conventions that application is left-associative and function bodies are interpreted to be as large as possible. White space is ignored (except to distinguish the application `x y` from the single identifier `xy`). SML-style comments, `(* ... *)` are permitted as well.

So, if we have an input file `input` containing

```
(* This is an example in the ML-like syntax
 *)
fn x => fn y => y x
```

then the function call `M.parse "input"` will return the abstract syntax shown in Section 1.1.

If you've been reading carefully, you may be surprised to see `let` in the concrete syntax — the abstract syntax didn't include any special provisions for defining variables, so how can such programs be parsed?

Frequently, as here, `let`-expressions can be viewed as convenient syntax for function applications. In particular, the source code

`let val` x `=` e_1 `in` e_2 `end`

(where x is an arbitrary identifier and e_1 and e_2 are arbitrary source expressions) is completely equivalent to the function application

`(fn` x `=>` e_2`)` e_1

This works because in either case we will associate x with e_1 and use this while evaluating e_2. *Stop here and make sure you understand this.*

Thus the `M.parse` function turns `let`-expressions in the source code into the corresponding function applications in the abstract syntax. For example, the input

```
let val id = fn x => x
in id id end
```

is parsed into the abstract syntax

```
App(Lam("id", App(Var "id", Var "id")),
    Lam("x", Var "x"))
```

1.2.2 Second Concrete Syntax

The other parsing function `S.parse` expects a completely different concrete syntax, one based on the language Scheme (which in turn was derived from the λ-calculus and the language LISP.) This syntax is based on a representation of the abstract syntax using lists; lists are written using what are called `S-expressions`. These lists are bounded by parentheses, and list elements are separated by whitespace (rather than commas). This stripped-down version of scheme doesn't include lists as run-time values, but the syntax remains:

⟨sexp⟩	::=	⟨id⟩	*identifier*
	|	`(` ⟨exp⟩ ⟨exp⟩ `)`	*application*
	|	`(lambda (` ⟨id⟩ `)` ⟨sexp⟩ `)`	*function*
	|	`(let (` ⟨sdefns⟩ `)` ⟨sexp⟩ `)`	*let-expression*
⟨sdefn⟩	::=	`(` ⟨id⟩ ⟨sexp⟩ `)`	*single definition*
⟨sdefns⟩	::=	ϵ | ⟨sdefn⟩ ⟨sdefns⟩	*sequence of definitions*

The definition of identifiers is unchanged. Comments in Scheme are written with a semicolon, and extend to the end of the line; there are no multi-line comments.

The examples of the previous section would be therefore be written as

```
; This is an example of
; the Scheme-like syntax
(lambda (x) (lambda (y) (y x)))
```

and

```
(let ((id (lambda (x) x)))
   (id id))
```

respectively.

1.3 Third and Fourth Concrete Syntaxes

Actually, the output of the functions `print_raw` and `print_nice` can be viewed as programs (character representations of abstract syntax) written using two still different concrete syntaxes. However, no parser for these forms has been supplied.

2 Exercises

2.1 Substitution (10%)

Substitution in the λ-calculus can be defined recursively as follows

$$\begin{array}{lll} x[x{\mapsto}e] & := e & \\ y[x{\mapsto}e] & := y & (\text{if } x \neq y) \\ (e_1 e_2)[x{\mapsto}e] & := (e_1[x{\mapsto}e])(e_2[x{\mapsto}e]) & \\ (\lambda y.e_1)[x{\mapsto}e] & := \lambda y.(e_1[x{\mapsto}e]) & (\text{if } x \neq y \text{ and } y \text{ is not free in } e) \end{array}$$

At first glance it looks like substitution might fail in some cases, such as

$$(\lambda x.x)[x{\mapsto}\lambda y.y]$$

or

$$(\lambda y.z)[z{\mapsto}y]$$

because the side-conditions fail for the definition of substituting into a function. However, we can always find an *equal* term to which the substitution can be applied, by renaming the bound variable. Thus

$$(\lambda x.x)[x{\mapsto}\lambda y.y] = (\lambda z.z)[x{\mapsto}\lambda y.y] = \lambda z.z$$

Similarly

$$(\lambda y.z)[z{\mapsto}y] = (\lambda w.z)[z{\mapsto}y] = \lambda w.y$$

We have to be careful though... it would be *incorrect* to rename the bound variable `y` to `z` in the second example and try to argue that

$$(\lambda y.z)[z{\mapsto}y] = (\lambda z.z)[z{\mapsto}y] = (\lambda w.w)[z{\mapsto}y] = \lambda w.w$$

We cannot safely transform $\lambda y.z$ into $\lambda z.z$ because the free variable z would be erroneously "shadowed" or "captured"; these are not α-equivalent.

When working with the λ-calculus on paper, a lot of this can be swept under the rug. We can generally just assume that we can choose bound variable names so that there are no clashes. Things are a little more complicated when we try to implement the λ-calculus as here; the computer will not automatically rename identifiers to prevent shadowing and variable capture.

There are many approaches that have been used to address this problem. We will add a notion of "fresh" variables — variables that we know have not been used *anywhere* yet, and which therefore differ from the variables in any particular term. If z is a fresh variable, then $\lambda y.e_1$ is guaranteed α-equivalent to $\lambda z.(e_1[y \mapsto z])$ and so and so by the definition of substitution above,

$$(\lambda y.e_1)[x \mapsto e] = (\lambda z.(e_1[y \mapsto z]))[x \mapsto e] = \lambda z.((e_1[y \mapsto z])[x \mapsto e]).$$

(Since z was chosen to be fresh, we are guaranteed that $z \neq x$ and that z is not free in e.) We can therefore revise the definition of substution as follows:

$$\begin{array}{lll} x[x \mapsto e] & := e & \\ y[x \mapsto e] & := y & (\text{if } x \neq y) \\ (e_1 e_2)[x \mapsto e] & := (e_1[x \mapsto e])(e_2[x \mapsto e]) & \\ (\lambda y.e_1)[x \mapsto e] & := \lambda z.((e_1[y \mapsto z])[x \mapsto e]) & (\text{where } z \text{ is fresh}) \end{array}$$

Now the case for substituting into functions will work in any case.

We can generate fresh variables by using a counter. The function

```
freshVar : unit -> string
```

will generate a "new" variable. These are simply the variables "z_0","z_1","z_2", etc. You should not ever create a variable with this sort of name when creating pieces of abstract syntax to test. (It is *impossible* to get variables with these names out of the parser, since these are not valid identifiers in the source language.) If desired, you can call the function `resetVar : unit -> unit` to start this sequence from zero.

Write a function

```
subst : absyn * string * absyn -> absyn
```

which takes the first argument and replaces all occurrences of the variable represented by the second argument by the expression given by the third argument, using capture-avoiding substitution as just described.

Extra Credit(10%) An unfortunate consequence of the way subsitution was re-defined is that it specifies that two substitutions must be applied in succession to the same function body. It is not too hard to show that in the worst case substitutions can take exponential time. This is not too hard to avoid; we must simply define substitution to take a list of variable/expression pairs, so that we can do all the substitutions in parallel. For extra credit, write a function

```
substs : absyn * (string * absyn) list -> absyn
```

that takes an expression and a list of substitutions for variables, and applies all of these subsitutions. Thus when we create a fresh variable, the renaming substitution (replacing y by z in the definition above) can simply be added to the list of substitutions we want to apply to the function body. Given this definition, the `subst` function can be defined simply to call this helper function with a list containing a single substitution.

3 One-step β-Reduction (40%)

In class you saw many examples of β-reduction applied to various terms; here's your chance to implement it! Because there may be many ways to reduce a single term, people have come up with various "strategies" which, given any term, specify what the next reduction step should be. Here are four (the names are generally historical).

- **Normal-Order Reduction**: reduce the application (of a function to an argument) whose λ appears leftmost in (i.e., closest to the beginning of) the term.

  $$\Big(\lambda x.(\lambda z.((\lambda w.w)(\lambda v.v)))\Big)\Big((\lambda u.u)(\lambda y.y)\Big) \rightarrow_\beta \lambda z.((\lambda w.w)(\lambda v.v)) \rightarrow_\beta \lambda z.(\lambda v.v)$$

  Normal-order reduction is interesting because it has the following guarantee: if there is any sequence of reductions to a term which cannot be reduced further, then normal-order reduction will eventually reach this term.

- **Applicative-Order Reduction**: reduce the application whose λ occurs leftmost if the argument of this application cannot be further reduced; otherwise, first reduce the argument by one step (using applicative-order reduction).

  $$\begin{aligned}&\Big(\lambda x.(\lambda z.((\lambda w.w)(\lambda v.v)))\Big)\Big((\lambda u.u)(\lambda y.y)\Big)\\ &\rightarrow_\beta \Big(\lambda x.(\lambda z.((\lambda w.w)(\lambda v.v)))\Big)\Big(\lambda y.y\Big) \rightarrow_\beta \lambda z.((\lambda w.w)(\lambda v.v)) \rightarrow_\beta \lambda z.(\lambda v.v)\end{aligned}$$

  Applicative reduction is does not have the same termination guarantee as normal-order reduction, but in many cases it is more efficient.

- **Call-by-Name Evaluation**: The same as normal-order reduction, except we never reduce applications inside the body of a function. (This strategy may stop before all reductions are gone.)

  $$\Big(\lambda x.(\lambda z.((\lambda w.w)(\lambda v.v)))\Big)\Big((\lambda u.u)(\lambda y.y)\Big) \rightarrow_\beta \lambda z.((\lambda w.w)(\lambda v.v))$$

  This is a lazier version of normal-order reduction. If the result is a function that is not being applied to anything, why bother working on the function?

- **Call-by-Value Evaluation**: The same as applicative-order reduction, except that we never reduce applications inside the body of a function. (This strategy may stop before all reductions are gone.)

$$\Big(\lambda x.(\lambda z.((\lambda w.w)(\lambda v.v)))\Big)\Big((\lambda u.u)(\lambda y.y)\Big)$$
$$\rightarrow_\beta \Big(\lambda x.(\lambda z.((\lambda w.w)(\lambda v.v)))\Big)\Big(\lambda y.y\Big) \rightarrow_\beta \lambda z.((\lambda w.w)(\lambda v.v))$$

  This is a lazier version of applicative reduction.

We will talk more about all four of these strategies in class.

Write four functions corresponding to the above four strategies:

```
normal      : absyn -> absyn option
applicative : absyn -> absyn option
cbn         : absyn -> absyn option
cbv         : absyn -> absyn option
```

These functions should return `NONE` if the argument cannot be reduced according to the given strategy, and `SOME` e' if the argument can be reduced one step to get the new term e'.

You may find the following idiom to be useful:

```
(case (recursive call) of
   NONE => (do something)
 | SOME e' => (do something else))
```

It is always wise to put parentheses around `case` statements.

4 Multistep β-Reduction (20%)

Write a function `doit` which takes a reduction strategy (i.e., one of the four functions above), a parser, and a filename as sequential arguments. The function should repeatedly do one-step reduction using the given strategy, until no further reductions are possible; if this occurs, return the final λ-expression. Your function should have the following name and type (or a more general polymorphic type):

```
doit : (absyn -> absyn option) -> (string -> absyn) -> absyn
```

5 Factorial (15%)

Create a file `fact3` containing a lambda term in the ML-like syntax that computes the result of applying the factorial function applied to the number 3. Your code should be well-commented enough to convince the graders that you have the right λ-term.

Your `doit` function should reduce this to the Church numeral for 6 when using normal-order reduction. (Call-by-name evaluation might stop with a huge term remaining; applicative-order reduction probably won't terminate; call-by-value evaluation might have either of these problems.)

6 Fibonnaci (15%)

Create a file `fib4` containing a lambda term in the Scheme-like syntax that computes the result of applying the Fibonnaci function applied to the number 4. Your code should be well-commented enough to convince the graders that you have the right λ-term.

Again, the code should reduce to the correct Church numeral using many steps of normal-order reduction.