

CS442

Assignment 2

University of Waterloo

Winter 2021

This assignment tests your understanding of the content of Modules 2 and 3. You will implement a λ -calculus reducer in GNU Smalltalk. The assignment is divided into four parts, but you should submit only one file: **a2.st**.

Submit all code via UWaterloo submit, e.g.:

```
submit cs442 a2 .
```

This assignment is due on Wednesday, February 24th, by 12PM *NOON*, *NOT MIDNIGHT*, Eastern time.

The λ -Calculus in Smalltalk

You are provided with **lambda.st**, an implementation of the types for the syntax tree of the λ -calculus in GNU Smalltalk. It implements the following classes:

```
Object subclass: LambdaExp [
    isVariable.
    isAbstraction.
    isApplication.
    ifVariable: varBlock ifAbstraction: absBlock ifApplication: appBlock.
    reduceWith: block steps: steps.
    freeVars.
    printString.
]
```

```
LambdaExp subclass: LambdaVar [
    | name |
    LambdaVar class >> withName: name.
    dup.
    isVariable.
    name.
    freeVars: map.
    displayString.
]
```

```
LambdaExp subclass: LambdaAbs [
    | var body |
    LambdaAbs class >> withVar: var body: body.
    dup.
    isAbstraction.
    var.
    body.
    freeVars: map.
    displayString.
]
```

```
LambdaExp subclass: LambdaApp [
    | rator rand |
    LambdaApp class >> withRator: rator rand: rand.
```

```

dup.
isApplication.
rator.
rand.
freeVars: map.
displayString.
]

Object subclass: LambdaParser [
  | text index tok |
  LambdaParser class >> new: text.
  LambdaParser class >> parse: text.
  parse.
]

```

The `LambdaExp` class serves as a superclass for all λ -calculus expressions. Each of its methods are only meant to be used on subclasses. Its `isVariable`, `isAbstraction`, and `isApplication` methods each return `true` if the expression is of the given type, or `false` otherwise. The `ifVariable:ifAbstraction:ifApplication:` method lets you easily “switch” based on the type; each block will be evaluated only if the value is of the given type. The `freeVars` method returns the free variables of the expression, as a `Dictionary` mapping the name of the free variable to the `LambdaVar` that uses it in the expression; generally, all that you will need is the presence of a free variable, i.e., `includesKey:`. `printString` is overridden to fall through to `displayString`, and `displayString` is overridden in all child types to return a λ -calculus expression as a string, with λ replaced by `\`.

The `reduceWith:steps:` method is a simple method to reduce the expression using the block given as the first argument, maximally as many steps as is given as the second argument. Its use will become clearer in the assignment parts on reduction.

We will not repeat the behavior of the shared methods, only the unique methods per each subclass of `LambdaExp`.

Every subclass of `LambdaExp` includes a `dup` method, which duplicates the expression, including all children.

The `LambdaVar` class represents a variable. A `LambdaVar` is constructed by `LambdaVar class >> withName:`, which expects the variable name as an argument. The variable name should be a string starting with an alphabetical character, unless this expression is using deBruijn indices, in which case it should be a number. The `name` method returns this name.

The `LambdaAbs` class represents an abstraction. A `LambdaAbs` is constructed by `LambdaAbs class >> withVar:body:`, which expects a string variable name (*not* a `LambdaVar`) and a `LambdaExp` (the body) as arguments. `var` returns the variable, and `body` returns the body. If using deBruijn indices, the variable should be `' '` (the empty string).

The `LambdaApp` class represents an application. A `LambdaApp` is constructed by `LambdaApp class >> withRator:rand:`, which expects the rator and rand as `LambdaExps`. `rator` and `rand` return the rator and rand, respectively.

Finally, the `LambdaParser` class is a parser for λ -calculus strings, which converts such strings into `LambdaExps`. It can be used in one of two ways: either by constructing an instance of `LambdaParser` and calling `parse` on it, or by using `LambdaParser class >> parse:` directly. For instance, to parse the string `'\f.\x.f (f x)'`, you can use either `(LambdaParser new: '\f.\x.f (f x)') parse` or `LambdaParser parse: '\f.\x.f (f x)'`. The latter is, of course, implemented in terms of the former. Note that since this implementation of the λ -calculus allows multi-character variable names, `f x` is not the same as `fx`: the first is an application of a variable to a variable, and the second is a variable.

Several examples are included in the “demonstration” section of this document.

1 de Bruijn

Write a file, `a2.st`, which defines at least the following class:

```
Object subclass: Lambda [
    Lambda class >> new: exp.
    toDeBruijn.
]
```

You may (and should!) include more methods to help your implementation.

A `Lambda` will eventually be a λ -calculus reducer. For the time being, it's just a de-Bruijn-translator. A `Lambda` is created with a `LambdaExp` (`exp`), which should be stored and updated by all methods.

The `toDeBruijn` method converts a non-de-Bruijn-indexed expression to a de-Bruijn-indexed one, and both returns the de-Bruijn-indexed expression and sets the internal expression to it. All abstractions should be replaced with abstractions that have '' (the empty string) as the variable, and all variables should be replaced with numbers at the appropriate depth. Remember that de Bruijn indices are one-indexed, so $\lambda x. x$ translates as $\lambda. 1$, not $\lambda. 0$.

Any free variables in the expression should be left unchanged. Note that as a consequence of this fact, `deBruijn` can be safely repeated (i.e., it's idempotent): if de Bruijn indices are interpreted as non-de-Bruijn variable names, they will always be free, and so won't be replaced.

`toDeBruijn` may mutate the expression or any of its child objects, or it may create a new expression.

Hints

It would be difficult to implement this with `toDeBruijn` alone. You will need to implement something that carries the list of currently-defined variables, so that you can look up a variable in that list and replace it.

You may want to implement that as a method on `LambdaExp`'s children. If so, please don't modify `lambda.st`: you will not be submitting that file. You can extend an existing class using GNU Smalltalk's `extend` syntax, e.g.:

```
LambdaVar extend [
    toDeBruijn: map [
        ...
    ]
]
```

You would be wise not to *replace* any existing methods in `LambdaVar`, since our tests could use any of them, but you may safely *add* any methods you please. In fact, you can add methods to any class you want, even internal classes!

2 Applicative Order Evaluation

Extend `a2.st`, adding at least the following methods to `Lambda`:

```
Object subclass: Lambda [
    ...
    aoe.
    aoe: steps.
]
```

The `aoe` method performs a single reduction step using applicative order evaluation on the expression, and returns the reduced version. It should also update the internal expression, so that if `aoe` is called repeatedly, multiple steps of reduction are taken. If there is no reduction step for AOE (i.e., reduction is complete), then it should return `nil`, and set the `Lambda`'s internal expression to `nil` (so any further calls to `aoe` will fail).

The `aoe:` method performs a specified number of steps of AOE, and returns the result. If fewer than the specified number of steps ends the reduction, then the final reduced expression should be returned, *not* `nil`. The internal expression should be updated to the same result.

The `aoe:` method is best implemented by `reduceWith:steps:`, perhaps indirectly. The block passed to `reduceWith:` should perform a single step of reduction on its argument (which you may safely ignore if you're directly using `aoe`), and that reduction will be repeated as many times as are passed to `steps:`, returning early if `nil` is reached.

You may perform reduction directly on a λ -calculus expression, or you may perform de Bruijn rewriting first. The result of `aoe` may or may not use de Bruijn indices, by your preference. However, the input to `new:` will never use de Bruijn indices, so if you wish to use de Bruijn indices, you should probably modify `new:` (or an `init:` method it uses) to perform this step. There is no method to directly extract the expression, so it's safe to keep your internal expression in either form.

`aoe` may mutate the expression or any of its children, or may create a new expression, or any combination thereof. If you use mutation, make sure to use `dup` during substitution; having two references to the same subexpression will cause some extremely confusing behavior!

Hints

Be *very* careful about reduction steps: `aoe` should represent a *single* application of AOE's \rightarrow , not two or three! Order is also important, and you will be tested on whether you've reduced the right part of the expression.

Reduction requires substitution, so you will presumably want to implement a substitution method. Substitution requires the ability to create a "new" variable name. Our version of the λ -calculus in this assignment is more forgiving in variable names than in Module 2; in particular, they may be of any length. One simple technique to generate fresh variable names is to carry a counter, and increment it every time you generate a new name. An even simpler technique uses a Smalltalk-specific trick: every object has a `hash`, and the hash is unique to that object. So, if you need a new name, you can use the `hash` method to get a unique number, so long as you're calling it on an adequately unique object. For instance, the canonical implementation of `LambdaAbs >> aoe` (which is called by the canonical `Lambda >> aoe`) includes these statements:

```
nvar := var , (self hash asString).
body := body substitute: var for: (LambdaVar withName: nvar).
var := nvar.
```

As with `toDeBruijn`, you may find reduction easier to do by extending the children of `LambdaExp`.

Most demonstrations of the λ -calculus use shorthand, e.g. `[[true]]`, which isn't truly part of the λ -calculus, and so isn't supported by our reducer. If you have an expression E that uses the shorthand $x = M$, you can rewrite E as $(\lambda x. E)M$. For example, we can rewrite $\lambda l. l$ `[[true]]` as $(\lambda true. \lambda l. l \text{ true})(\lambda x. \lambda y. x)$. See the "demonstration" section for some larger examples.

3 Normal Order Reduction

Extend `a2.st`, adding at least the following methods to `Lambda`:

```
Object subclass: Lambda [
    ...
    nor.
    nor: steps.
]
```

The `nor` and `nor:` methods behave like `aoe` and `aoe:`, but using normal order reduction instead of applicative order evaluation. Like `aoe` and `aoe:`, they may mutate the expression, or create a new one.

Note that although it may be strange to do so, mixing and matching `aoe` and `nor` is perfectly valid. You must assure that nothing prevents this.

Hints

You must reduce the outermost *reducible* expression. To know whether to reduce the current expression or recurse deeper, you need only to check some types. Remember, an expression is a redex if it's an application and its rator is an abstraction. The `isAbstraction` method is there for exactly this check!

4 η -Reduction

Extend `a2.st`, adding at least the following methods to `Lambda`:

```
Object subclass: Lambda [
    ...
    eta.
    eta: steps.
]
```

The `eta` and `eta:` methods behave like `aoe` and `aoe:` or `nor` and `nor:`, but using η -reduction instead of β -reduction. Reduce the leftmost, innermost η -reducible expression.

Demonstration

The following demonstrates a possible interaction with `Lambda`:

```
st> | x s l |
st> x := LambdaParser parse: '(\mul.\two.mul two two) (\m.\n.\f.m(n f)) (\f.\x.f (f x))'.
(((\mul.\two.((mul two) two))) (\m.\n.(\f.(m (n f))))) (\f.\x.(f (f x))))
st> l := Lambda new: x.
a Lambda
st> l aoe.
((\two.(((\m.\n.(\f.(m (n f))))) two) two)) (\f.\x.(f (f x))))
st> l aoe.
(((\m.\n.(\f.(m (n f))))) (\f.\x.(f (f x)))) (\f.\x.(f (f x))))
st> l aoe.
(((\n.(\f.((\f.\x.(f (f x)))) (n f)))) (\f.\x.(f (f x))))
st> x := l aoe.
(\f.((\f.\x.(f (f x)))) ((\f.\x.(f (f x)))) f))
st> x toDeBruijn.
(\.(\.(\.(2 (2 1)))) ((\.(2 (2 1)))) 1)))
st> l aoe.
nil
st> x := LambdaParser parse: '(\mul.\two.mul two two) (\m.\n.\f.m(n f)) (\f.\x.f (f x))'.
(((\mul.\two.((mul two) two))) (\m.\n.(\f.(m (n f))))) (\f.\x.(f (f x))))
st> l := Lambda new: x.
a Lambda
st> x := l aoe: 1000.
(\f.((\f.\x.(f (f x)))) ((\f.\x.(f (f x)))) f))
st> s := x displayString.
'(\f.((\f.\x.(f (f x)))) ((\f.\x.(f (f x)))) f))'
st> x toDeBruijn.
(\.(\.(\.(2 (2 1)))) ((\.(2 (2 1)))) 1)))
st> x := LambdaParser parse: s, 'f x'.
(((\f.((\f.\x.(f (f x)))) ((\f.\x.(f (f x)))) f)) f) x)
st> l := Lambda new: x.
a Lambda
st> l aoe.
(((\f.\x.(f (f x)))) ((\f.\x.(f (f x)))) f) x)
st> l aoe.
(((\f.\x.(f (f x)))) (\x.(f (f x)))) x)
st> l aoe.
((\x.((\x.(f (f x)))) ((\x.(f (f x)))) x)) x)
st> l aoe.
((\x.(f (f x))) ((\x.(f (f x)))) x))
st> l aoe.
((\x.(f (f x))) (f (f x)))
st> l aoe.
(f (f (f (f x))))
st> x := LambdaParser parse: '(\head.\tail.\cons.\zero.\two.(\succ.(\pred.(pred two)) (\n.(tail(n(\p.cons(succ(head p)) (head p))(cons zero zero))))) (\n.\f.\x.n f(f x))) (\l.l\l.x.y.x) (\l.l\l.x.y.y) (\h.\t.\s.s h t) (\f.\x.x) (\f.\x.f (f x)) f x'.
(((((((\head.\tail.(\cons.(\zero.\two.((\succ.((\pred.(pred two)) (\n.(tail ((n (\p.((cons (succ (head p)) (head p)) (cons zero zero))))) (\n.(\f.\x.n f(f x))) (\l.l\l.x.y.x) (\l.l\l.x.y.y) (\h.\t.\s.s h t) (\f.\x.x) (\f.\x.f (f x)) f x')))))))))))) (\n.(\f.\x.((n f) (f x))))))))) (\l.(l (\x.(\y.x))) (\l.(l (\x.(\y.y))) (\h.\t.\s.((s h t)))) (\f.\x.x)) (\f.\x.(f (f x)))) f) x)
```

```

st> l := Lambda new: x.
a Lambda
st> x := l aoe: 1000.
(f x)
st> x toDeBruijn.
(f x)
st> x := LambdaParser parse: '(\mul.\two.mul two two) (\m.\n.\f.m(n f)) (\f.\x.f (f x))'.
(((\mul.\two.((mul two two))) (\m.\n.(\f.(m (n f))))) (\f.\x.(f (f x)))))
st> l := Lambda new: x.
a Lambda
st> l nor.
((\two.(((\m.\n.(\f.(m (n f))))) two) two)) (\f.\x.(f (f x)))))
st> l nor.
(((\m.\n.(\f.(m (n f))))) (\f.\x.(f (f x))))) (\f.\x.(f (f x)))))
st> l nor.
((\n.(\f.((\f.\x.(f (f x))))) (n f))) (\f.\x.(f (f x)))))
st> l nor.
(\f.((\f.\x.(f (f x))))) ((\f.\x.(f (f x))) f)))
st> l nor.
(\f.\x.(((\f.\x.(f (f x))) f) (((\f.\x.(f (f x))) f) x))))
st> l nor.
(\f.\x.((\x.(f (f x))) ((\f.\x.(f (f x))) f) x))))
st> l nor.
(\f.\x.(f (f ((\f.\x.(f (f x))) f) x))))
st> l nor.
(\f.\x.(f (f ((\x.(f (f x))) x))))
st> x := l nor.
(\f.\x.(f (f (f (f x)))))
st> x toDeBruijn.
(\.(\.(2 (2 (2 (2 1)))))
st> l nor.
nil
st> x := LambdaParser parse: '(\mul.\two.mul two two) (\m.\n.\f.m(n f)) (\f.\x.f (f x))'.
(((\mul.\two.((mul two two))) (\m.\n.(\f.(m (n f))))) (\f.\x.(f (f x)))))
st> l := Lambda new: x.
a Lambda
st> x := l nor: 1000.
(\f.\x.(f (f (f (f x)))))
st> s := x displayString.
'(\f.\x.(f (f (f (f x)))))'
st> x toDeBruijn.
(\.(\.(2 (2 (2 (2 1)))))
st> x := LambdaParser parse: s , 'f x'.
(((\f.\x.(f (f (f (f x))))) f) x)
st> l := Lambda new: x.
a Lambda
st> l nor: 1000.
(f (f (f (f x))))
st> x := LambdaParser parse: '(\head.\tail.\cons.\zero.\two.(\succ.(\pred.(pred two)) (\n.(tail(n(\p.cons(succ(head p))
(head p))(cons zero zero)))) (\n.\f.\x.n f(f x))) (\l.l\l.x.y.x) (\l.l\l.x.y.y) (\h.\t.\s.s h t) (\f.\x.x) (\f.\x.f
(f x)) f x'.
((((((((head.\tail.\cons.\zero.\two.(\succ.((\pred.(pred two)) (\n.(tail ((n (\p.((cons (succ (head p)) (head
p)))) ((cons zero zero)))) (\n.(\f.\x.((n f) (f x))))) (\l.(l (\x.(\y.x)))) (\l.(l (\x.(\y.y)))) (\
h.\t.\s.((s h) t)))) (\f.\x.x)) (\f.\x.(f (f x)))) f) x)
st> l := Lambda new: x.
a Lambda
st> x := l nor: 1000.
(f x)
st> x toDeBruijn.
(f x)
st> x := LambdaParser parse: '(\head.\tail.\cons.\isNull.\nil.\zero.\succ.(\Y.\F.(\len.(len (cons zero (cons zero nil))
)) (Y F)) (\f.(\x.f(x x))(\x.f(x x))) (\f.\l.(isNull l) zero (succ (f(tail l)))) (\l.l\l.x.y.x) (\l.l\l.x.y.y)
(\h.\t.\s.s h t) (\l.l\h.\t.\x.y.y) (\s.\x.\y.x) (\f.\x.x) (\n.\f.\x.n f(f x))'.
((((((((((((head.\tail.\cons.\isNull.\nil.\zero.\succ.(((Y.\F.((len.(len ((cons zero) ((cons zero) nil)))) (Y F)
))) (\f.((\x.(f (x x))) (\x.(f (x x))))) (\f.\l.((isNull l) zero) (succ (f (tail l)))))))))) (\l.(l (\x.(\
y.x)))) (\l.(l (\x.(\y.y)))) (\h.\t.\s.((s h) t)))) (\l.(l (\h.\t.\x.(\y.y)))) (\s.\x.(\y.x)) (\f.\x
x.x)) (\n.(\f.\x.((n f) (f x)))))
st> l := Lambda new: x.
a Lambda
st> l nor: 1000.
(\f.\x.(f (f x)))
st> x := LambdaParser parse: '\m.\n.\f.\x.m(n f)x'.
(\m.\n.(\f.\x.((m (n f) x))))
st> l := Lambda new: x.
a Lambda
st> l eta.
(\m.\n.(\f.(m (n f))))
st> l eta.
nil
st> x := LambdaParser parse: '\m.\n.\f.\x.n m f x'.
(\m.\n.(\f.\x.(((n m) f) x))))
st> l := Lambda new: x.
a Lambda
st> l eta.

```

```
(\m.(\n.(\f.((n m) f))))
st> l eta.
(\m.(\n.(n m)))
st> l eta.
nil
st>
```

A Note on Testing

Because substitution creates fresh variable names, there are multiple (in fact, infinite) correct values for the result of any step of reduction which includes bound variables. You are not required to keep *any* bound variable names, even if they are unambiguous; you may choose to use de Bruijn indices for all reduction, but you may also rename any bound variable if you wish to. As such, testing will require `toDeBruijn`, as that's the only way of assuring that all names are consistent (in that there aren't any names). If your `toDeBruijn` method doesn't work, your grade on `aoe`, `nor`, and `eta` may suffer, simply because we cannot reach a baseline for testing. Make sure `toDeBruijn` works!

Rights

Copyright © 2020, 2021 University of Waterloo.
 This assignment is intended for CS442 at University of Waterloo.
 Any other use requires permission from the above named copyright holder(s).