Logical Lambda Calculator

Aaron Gorenstein

December 22, 2020

Abstract

I wanted to explore different models of computation. This document explores how might we realize λ-calculus through the "engine" of first-order predicate logic. The motivation for this document was a study on some excellent books[Mic11, SS94] on both these topics. What value this study provides is largely credited to those books; what errors found here are my own fault.

This is essentially my personal notes—the exposition is not as exhaustive as I would have liked. Perhaps this can be used as a tangent lecture in some programming language course.

Contents

1 Introduction

Broad Thoughts λ-calculus is a famous and fundamental model of computation. It realizes computation through *function application.* First-order predicate-logic is another equally fundamental model of computation. In the language prolog, it enables computation through *unification.*

To be clear: These two models realize computation through different mechanisms. In the coarsest terms, λ-calculus iteratively applies variable substitution (we shall see this ultimately as β-reduction) to produce a quiescent λ-term; this represents the result of the computation. For the first-order predicate-logic case, we will search in a depth-first-search-like manner through a possible database of facts to find the most general answer to a query. Whatever this search finds—or that it cannot find an answer—represents the result of the computation.

By implementing the means of evaluating λ-calculus expressions in prolog, we shall hopefully gain a deeper insight—or at least appreciation—for the differences.

Reader Beware The history of this document is that I first wrote the major code in early 2016. Sometime Fall 2016 I added comments and uploaded it to the online repository Github[Gor20], where it still resides, inside this newer document. In late 2020, I resurrected this and decided to fully document the code via "literate coding". So the prose is much newer than the code, though in remembering and writing I refactored much of the code. Some of the details, of prolog especially, may be fuzzy, though obviously the root approach and code all works up to the tests.

2 Representing λ-Calculus in Prolog

The grammar for λ-calculus[Mic11] is strikingly self-contained:

$\langle expression \rangle$ ::= $\langle name \rangle$
  | $\langle function \rangle$
  | $\langle application \rangle$

$\langle function \rangle$ ::= λ $\langle name \rangle$. $\langle expression \rangle$

$\langle application \rangle$ ::= ($\langle expression \rangle$ $\langle expression \rangle$)

This can be copied almost verbatim into our prolog code:

```
expression(L) :- name(L).
expression(L) :- function(L).
expression(L) :- application(L).
function([lambda, V, B]) :- name(V), expression(B).
application([E1, E2])    :- expression(E1), expression(E2).
name(X)       :- not(is_list(X)), ground(X), X \= lambda.
```

Listing 1: Initial Lambda Calculus Definitions

What is striking—and this is a tool distinct in prolog—is how the structure of, e.g., a function is on the *left*-hand side of the rule. We describe the structure we expect, and then add further requirements on the right-hand side. This is contrary to the typical intuition that the LHS is the "input", and the RHS is the computation to create the output. (The code will suggest that ideas in many places, but other places we will not restrict ourselves in that manner!)

Perhaps an analogy is in functional-language-esque pattern-matching, but by the "bidirectionality" available in prolog it can be used to both break apart expressions, and build new ones.

2.1 Demonstrations

The following sentences should be true. These sentences are arbitrary, simple cases to root the above code in real-world examples.

```
name(x).
name(y).
function([lambda, a, a]).
application([a, b]).
function([lambda, a, [a, b]]).
application([[lambda, a, a], b]).
application([[lambda, a, a], [lambda, c, [lambda, c, c]]]).
not(function([a, b])).
not(function([lambda, a, b, c])).
not(application([lambda, a, b])).
not(name([a])).
```

Listing 2: Structure.tests

3 Implementing β-Reduction

The fundamental action of λ-calculus is the β-reduction, whereby all instances of the bound variable are replaced by the applicant. We define this as:

```
beta_reduction([lambda, V, B], A, R) :- replace(V, B, A, R).
```

Listing 3: Beta Reduction

In English: The β-reduction of $\lambda V.B$ applied to A gives us R, where R is B, but with all instances of V replaced by A. More immediately: we go through the body B and do the text-substitution.

Let us see how `replace` is realized in prolog:

```
replace(V,V,A,A) :- name(V).
replace(V,W,_,W) :- name(W), V \= W.
replace(V,[lambda, V, B],_,[lambda, V, B]).
replace(V,[lambda, W, B],A,[lambda, W, S]) :-
    W \= V,
    replace(V,B,A,S).
replace(V, [E1, E2], A, [R1, R2]) :-
    replace(V,E1,A,R1),
    replace(V,E2,A,R2).
```

Listing 4: Replace Predicate

We shall consider each rule in turn.

1. This is a nice demonstration of "the magic of prolog". Here the first parameter is `V`, *and* the expression we want to replace, the second parameter, is also `V`. In other words, we've found the name we want to replace with `A`. Consequently, the fourth parameter is the same as `A`, i.e., we "do" the substitution.

2. This is a similar situation, but the body `W` is *not* the same as the variable we're substituting. In that case, there's nothing to rename, so the out-parameter is set to `W` as well, i.e., unchanged, and we ignored the third parameter.

3. The body `B` is some function that shadows the bound variable. In this case we shouldn't do any replacement, so we completely ignore the third parameter, and the fourth is our body unchanged.

4. The body `B` is some function that *doesn't* shadow the bound variable. Our output is the same function, but with the interior body `B` changed to `S`, which is the recursive call.

5. Lastly, this is when we have an appliction: *this* case doesn't change which variables are or aren't bound, so it's a straightforward recursive definition. Note in particular how the output parameter "already uses" `R1` and `R2`, which feels unintuitive for those accustomed to imperative or functional languages.

Hopefully this is understood as a fairly concise and immediate definition of how replacements of β-reductions are realized in λ-calculus. To further elucidate, here are some β-reductions demonstration some of the naming behavior. Observe that we *must* have a function as the first parameter.

3.1 Demonstrations

```
beta_reduction([lambda, x, x], a, a).
beta_reduction([lambda, x, y], a, y).
beta_reduction([lambda, x, [lambda, x, [x, y]]], a, [lambda, x, [x, y]]).
beta_reduction([lambda, x, [lambda, y, [x, y]]], a, [lambda, y, [a, y]]).
beta_reduction([lambda, x, x], [lambda, x, [x, x]], [lambda, x, [x, x]]).
```

Listing 5: BetaReductions.tests

4 Evaluation

The complete computation for λ-calculus requires *evaluation*. Where β-reductions are only defined for functions, evaluation will take any λ-expression and give the result. There is applicative order and normal order–we shall do normal.

```
evaluate([E1, E2], R) :-
    evaluate(E1, R1), E1 \= R1,
    evaluate([R1, E2], R).
evaluate([E1, E2], R) :-
    beta_reduction(E1, E2, R1),
    evaluate(R1, R).
evaluate(L, L).
```

Listing 6: Evaluate

For conciseness the implementation of this procedure uses recursion, and exploits that prolog will always search for the *first* solution.

1. The first rule shows that when considering an application, we should first evaluate its left-hand side (in case if the left-hand side is *also* an application).

2. The second rule will eagerly try to do a β-reduction (which will fail is `E1` is not a function) and recurse on the result.

3. The third succinctly capture that, if we are no longer able to evolve the λ-expression, we're done. This includes the cases where `L` is a `name` or `function`.

This definition of evaluation opens the door to multiple answers, essentially each one doing "less and less" computation. We order of rules in the procedure means that the first answer always has the maximal amount of computation. See chapter 7 of [SS94] for a discussion of how rule-order shapes the order of solutions found.

4.1 Demonstrations

```
evaluate(a, a).
evaluate([lambda, a, a], [lambda, a, a]).
evaluate([[lambda, a, a], b], b).
evaluate([ [lambda, x, [lambda, y, [y, x]]], [[lambda, a, [a, a]], b] ] ,
↪  [lambda, y, [y, [[lambda, a, [a, a]], b]]]).
```

Listing 7: Evaluation.tests

The last line confirms that we have normal-order evaluation; observe how we don't evaluate the self-apply function.

4.2 Midpoint Conclusion

In a sense, we have achieved our goal! We have a (rather awkward) way of writing arbitrary lambda expressions and evaluate them. Are we done?

By analogy, I would suggest we have something like the ALU in a CPU. We can do the fundamental operations we're interested in, but lack memory or any reasonable interface to actually deploy this computation. The following section we introduce an *extremely* bare-bones environment so that we may more easily do interesting computation. There is still quite a distance to, say, Scheme (and indeed, we are not going to be getting there), but this will contain a few more interesting applications of prolog to illuminate interesting things in λ-calculus.

5 Introducing an Environment

The simplest environment would be to "save" an atom, such as `identity`, and associate it with a value, such as `[lambda, x, x]`. This is so that in future cases where we see the (unbound) atom `identity`, we would replace it with the value. This sort of machinery is conventionally called "sugaring" (and its removal, which I'll call "desugaring"). It adds no real additional computational power, just convenience. This is a very primitive macro system, in a sense.

We can start with some basic values. Classic definitions for, e.g., logical values include:

```
(define true (lambda x (lambda y x)))
(define false (lambda x (lambda y y)))
(define and (lambda x (lambda y ((x y) false))))
(define not (lambda x ((x false) true)))
```

Listing 8: Logic Definitions

A fuller exploration of these, including how these values were determined, can be found in [Mic11].

Assuming we have those names mapping, we would like to replace those names with their associated expressions before evaluation. Of course, λ-calculus machinery is excellent at replacing names with other values:

```
desugar([N,E], L, R) :- beta_reduction([lambda, N, L], E, R).
desugar_all(L, D, R) :- foldl(desugar, D, L, R).
```

Listing 9: Desugaring

For each name, λ-expression pair (N, E), we essentially compute: $R = ((\lambda N.L)E)$, which means any unbound N in L is replaced by the λ-expression E. That's exactly what we want, machinery-wise.

So given a sugar-using λ expression, we can now succinctly take out the sugar, opening the door for further `evaluate`-ing. However, the *result* of that computation would be unsugared. So an expression like `((and false) true)`, which we would hope give us `false`, would in fact give us `lambda, x, [lambda, y, y]`. We would like resugar those results. In spirit, this is "merely" doing beta-reduction in reverse:

```
resugar([N, E], L, N) :- isomorphic(E, L).
resugar(M, [lambda, V, B], [lambda, V, SB]) :- resugar(M, B, SB).
resugar(M, [E1, E2], [R1, R2]) :-
    resugar(M, E1, R1),
    resugar(M, E2, R2).
resugar(_, L, L).
resugar_all(L, D, R) :- foldl(resugar, D, L, S), S \= L, resugar_all(S, D, R).
resugar_all(L, _, L).
```

Listing 10: Resugaring

How does this resugaring work? Consider each rule:

1. The first rule is the main machinery: if the λ-expression `L` is isomorphic to the expression `E` that is sugared as name `N`, we should output `N`.
2. Rules 2 and 3 are the recursive exploration of the λ-expression.
4. Rule 4 ensures that we always succeed at resugaring, even if we fail to do any actual substitution.
5. We will maximally apply resugaring, and then if there was any change try again.
6. Again we ensure that we'll always succeed at resugaring even if no changes were detected.

An unfortunate detail of this is that this is *quite* inefficient. It is likely a prolog expert can take a look at this procedure or `isomorphic` (listing 13) and improve the performance. An obvious experiment is to put the `isomorphic` rule "lower down" the list, but in my limited experiments that merely puts the slowdown elsewhere. An alternative is to replace the first rule with `resugar([N, L], L, N).`[1] This works for our examples, and is much faster, but without `isomorphic` we risk being confused by otherwise-equivalent λ-expressions having different variable names.

It is likely, though I have no considered it formally, that this is not a rigorous resugaring method. We are in essence trying to tile a tree, (recursively?) and I would bet that's at least NP-hard, if not worse. Another formulation may be treating the macros as grammar productions, and finding the best parse of this tree. I would describe the above algorithm as a greedy approach, but it is enough to get us what we want for this demonstration document.

5.1 Evaluation with Sugar

We can combine the desguar, resugar, and evaluation procedures to define:

```
compute(L, D, R) :-
    desugar_all(L, D, DL),
    evaluate(DL, S),
    resugar_all(S, D, R).
```

Listing 11: Compute

[1]The key here is that we are not calling `isomorphic`.

Where `D` is some sequence of `N, E` (name, value) pairs that we've defined elsewhere.

This is the complete computational *and* expressive power of the code in this document. There are still limitations—see section 10—but an interested (and patient...) reader can use this to explore λ-calculus in prolog with a workable way of referring to the higher-level definitions.

With the viewpoint that the "sugaring" is a limited form of a macro-system, we can see clearly how macros are distinct from the true β-reductions. They're literally a separate clause. This can help make clear why, even as macros in real languages like scheme, seem so similar to procedures yet still have distinguished functionality.

6 The α-Reduction

Of course if we have a macro for `lambda, y, y`, we would want to it to "match" against even if `L` is `lambda, x, x`. That is why we have the clause `isomorphic` as part of the resugaring procedure. We would like those to be considered equal even as they technically differ in variable names. This is an excellent reason to introduce α-reduction:

```
alpha_reduction(L, L) :- name(L).
alpha_reduction([E1, E2], [R1, R2]) :-
    alpha_reduction(E1, R1),
    alpha_reduction(E2, R2).
alpha_reduction([lambda, V, B], [lambda, X, ABB]) :-
    gensym(alpha_,X),
    replace(V, B, X, BB),
    alpha_reduction(BB, ABB).
```

Listing 12: Alpha Reduction

Observe that the only work is done in the last rule. The runtime-provided `gensym` is used to create the next unique atom, beginning with the prefix `alpha_`, in `X`. We then replace the *variable* `V` with `X`, and reconstitute our function. There is no reason why we can't use `beta_reduction` on the input function, except that it is unneeded.

We can now define when two λ-expressions are isomorphic:

```
canon(L, R) :- reset_gensym(alpha_), alpha_reduction(L, R).
isomorphic(A, B) :- canon(A, C), canon(B, C).
```

Listing 13: Isomorphic

6.1 Demonstrations

Here are some quick demonstration/tests of what we can consider isomorphic.

```
isomorphic(x, x).
isomorphic([lambda, x, x], [lambda, y, y]).
not(isomorphic([lambda, x, x], [lambda, y, [y, y]])).
not(isomorphic(x, y)).
```

Listing 14: AlphaReduction.tests

We now have a framework to start building up more familiar math systems!

7 Implementing Boolean Algebra

Following in the excellent footsteps of [Mic11], we will start with Boolean algebra. Note that we are using a more Scheme-like syntax for our parentheses. We will in fact implement a parser that can compile this more traditional syntax into our prolog objects, demonstrated in section 9.

Recall the logic definitions from earlier, in listing 8. We can use these to implement my favorite logical function, `nand`:

```
 <<insert listing 8 (Logic Definitions)>>
(define nand (lambda x (lambda y (not ((and x) y)))))
(execute ((nand true) true))
(execute ((nand true) false))
(execute ((nand false) true))
(execute ((nand false) false))
(halt)
```

Listing 15: Nand.filetest

This file is a simple test that can fed into our final product at the end of this document, and we can hand-verify that we get the values we want.

An interesting complication, from those accustomed to stricter programming environments, is that the definition of `nand` can be incorrectly defined: `(((not and) x) y)`, as shown here.

```
 <<insert listing 8 (Logic Definitions)>>
(define nand (lambda x (lambda y (((not and) x) y))))
(execute ((nand true) true))
(execute ((nand true) false))
(execute ((nand false) true))
(execute ((nand false) false))
(halt)
```

Listing 16: WrongNand.filetest

This gives us a gibberish result: the four executions yield true-false-true-false, as it happens. In some sense it's not surprising—we're applying non-commutative functions in the wrong order—but in another sense it is surprising that anything happens at all!

8 Implementing Arithmetic

As a final demonstration, we can implement some of the basic building blocks of arithmetic. Again, as always, the definitions here are all from the excellent reference book.[Mic11]

```
 <<insert listing 8 (Logic Definitions)>>
(define zero (lambda a a))
(define succ (lambda n (lambda s ((s false) n))))
(define one (succ zero))
(define pred1 (lambda n (n false)))
```

Listing 17: Arithmetic

We have a sparse demonstration:

```
 <<insert listing 17 (Arithmetic)>>
(execute (succ zero))
(execute (pred1 one))
(halt)
```

Listing 18: Arithmetic.filetest

This allows us to hand-verify that the value after `zero` is `one`, and the value before `one` is `zero`. A great insight.

Arithmetic motivates enabling the creating of recursive functions. While that is all purely in λ calculus, see section 10 for some discussion. We do not implement recursion in this document.

Hopefully this and the previous section help demonstrate what we can really start to do even with the tiny amount of prolog we've already written.

9 Parsing

The previous sections relied on us translating the Scheme-like lists into prolog-like lists. We shall implement the parser for that here.

In classic parsing style, we'll implement a tokenizer, and then a grammar.

9.1 Tokenizer

If memory serves (this is me writing in 2020 trying to remember what I did in 2016) I am greatly indebted to Sterling and Shapiro[SS94] in designing a prolog-implemented parser. (Certainly I'm greatly indebted to that book regardless!)

```
read_s(S)             :- get_char(C),                read_s(C, S).
read_s('\n',[]).
read_s(' ',S)         :- get_char(D),                read_s(D, S).
read_s('(',['('|S]) :- get_char(D),                read_s(D, S).
read_s(')',[')'|S]) :- get_char(D),                read_s(D, S).
read_s(C,[N|S])       :- name_char(C), read_name(C,N,D), read_s(D, S).
```

Listing 19: Tokenizer

The tokenizer is a driver that reads in new characters, and matches them into tokens: parens, names, or end-of-line. The clause `get_char` is the entry point into the outside world. Again think of the non-chronological perspective. It's almost like we're building up `S`. Observe that we use the *goal* to basically append to `S`. The rules, in order:

1. The output is `S` in the topmost clause. We read in a character and pass it as the first parameter. In that sense, the first parameter is `read_s/2` is the "just-read" character.

2. If we just read a newline, we're done, and our output is the empty sequence of tokens.

3. If we read a whitespace, discard it.

4. If we read a parens, we push it on to our sequence (the tail `S` is defined from the later calls of `read_s`—consider the base case of a newline).

5. The other parens case.

6. Reading a "name" requires additional reasoning, but the end result is the same: a single atom, `N`, is pushed on to our sequence of tokens.

The only "interesting" token, then, is that determined by `read_name`. The clauses for that predicate are as follows:

```
read_name(C,N,E) :-
    read_name_chars(S,C,E),
    atom_chars(N,S).
read_name_chars([C|S], C, E) :-
    name_char(C), !, % needed for io
    get_char(D),
    read_name_chars(S, D, E).
read_name_chars([],C,C) :- not(name_char(C)).
name_char(C) :- char_code(C,N), N >= 65, N =< 90. % upper case
name_char(C) :- char_code(C,N), N >= 97, N =< 122. % lower case.
name_char(C) :- char_code(C,N), N >= 48, N =< 57. % decimals
```

Listing 20: Read Name

The output parameter, such that it is, is `N` in the first rule. That is an atom, resulted by the character `C` *prepended* on to the string `S` built up by `read_name_chars`. The character `E` is the evidence that we've stopped reading valid-name-characters (see the last clause of `read_name_chars`). This procedure is not dissimiliar to the larger tokenizer machinery.

I can't explain the need for `!`, except the comment included from 2016.

This concludes the tokenizer.

9.2 Parser

Now given a sequence (concretely, a list) of tokens, we can distill them into the nested lists we want for our internal representation. The tricky thing is matching parentheses.

```
parse(S,L) :- parse(S, [], [L]).
parse(['('|T], R, [M|L]) :- parse(T, [')'|S], M), parse(S, R, L).
parse([')'|T], [')'|T], []).
parse([N|T], R, [N|L]) :- N \= ')', N \= '(', parse(T, R, L).
parse([],[],[]).
```

Listing 21: Parser

How does this work? That's an excellent question. We shall go line-by-line:

1. This is the entry point: A parse is valid when the sequence-of-tokens `S` yields the λ-expression `L`. This is implemented by deferring to `parse/3`, where the first parameter is some prefix of tokens, the second is the unmatched suffix of tokens, and the third parameter is the λ-expression of the prefix. So the entry-point parse is valid when there is no unparsed suffix.

2. This first rule for `parse/3` handles when the next token it sees is open-parens. This is where much of the trickiness happens. We essentially defer to a sub-parse of the tail `T` of that prefix, ending with the matching parens `')'`. The result of that sub-parse is `M`, which we wrap in a list (because it was in parens). The remaining suffix of that sub-parse, `S`, is itself parsed to give us the *tail* following that `M`.

3. When we reach the end of a parens, that is the start of some intermediate list we build up, so the result of that parse is [].

4. This handles any token not `(` or `)`, i.e., `lambda` or a `name`. In those cases, we simply push them on to our existing output list.

5. The base case.

Phew! How is it that this seems harder than the λ-calculator!?

We also want to print out our internal lambda representation the same way. Note that `atomic_list_concat` is provided by our environment, and given a (flat) list of tokens concatenates them into an atom.

```
lambda_to_atom(L, R) :- parse(S, L), atomic_list_concat(S, ' ', R).
```

Listing 22: Lambda Printer

Isn't that neat, we use the parser to "unparse" the λ-expression?

Finally, we want to present a simple user-interactive loop. This is an extremely primitive REPL, in a sense. We have a few different commands we can unify against:

```
execute_command(['define', N, L], D, [[N,R]|D]) :- compute(L, D, R).
execute_command(['execute', L], D,D) :-
    compute(L, D, R),
    lambda_to_atom(R,O),
    write(O), nl.
execute_command(['halt'],_,_) :- halt.
```

Listing 23: Repl Commands

The middle parameter D is the list of our defines—this is our environment. The interaction, such that it is, against this procedure is as follows:

```
main_loop(OD) :-
    read_s(S),
    parse(S,L),
    execute_command(L,OD,ND),
    main_loop(ND).
main_loop(OD) :- write('Parse error'), nl, main_loop(OD).
main :- main_loop([]), halt.
```

Listing 24: Main Loop

Observe that `main` serves as our entry point.

This completes our entire λ-calculator in prolog! The entire file (see appendix A) is the concatenation of these figures:

```
% insert listing 1 (Initial Lambda Calculus Definitions)
% insert listing 3 (Beta Reduction)
% insert listing 4 (Replace Predicate)
% insert listing 6 (Evaluate)
% insert listing 12 (Alpha Reduction)
% insert listing 13 (Isomorphic)
% insert listing 9 (Desugaring)
% insert listing 10 (Resugaring)
% insert listing 19 (Tokenizer)
% insert listing 20 (Read Name)
% insert listing 21 (Parser)
% insert listing 22 (Lambda Printer)
% insert listing 11 (Compute)
% insert listing 23 (Repl Commands)
% insert listing 24 (Main Loop)
```

Listing 25: lambda.pl

I hope I was able to share at least some novel insights and enabled some interesting thoughts.

10 Extensions

I do not intend to extend the ideas here further, but as a conclusion I will list what I see as next steps.

Express resugaring as β-reduction To me the most fascinating part of prolog is how, for instance, our `parse` procedure could also "unparse". The `append` procedure is the canonical example of this bi-directionality, as far as I can tell. So I wonder if a single `sugar` procedure, perhaps intertwined with `isomorphic` and `beta_reduction`, could cover both `resugar` and `desugar`.

Extend things to enable recursive definitions The implementation of arithmetic in λ-calculus motivates recursion[SS94], which in turn reveals some fascinating limits and interactions of macros versus evaluation. Seeing what needs to change, if anything, in the computer here to support that would be a natural extension.

Extend the parser to enable fewer parens A sort of meta-sugaring is providing a more sophisticated parser that can infer implicit parens. I think this is interesting in that, as far as I can tell, it would need an implementation "outside" the existing macro system. If memory serves, that would be a "reader" macro, rather than a . . . macro-macro.

Continue the functionality in [Mic11] Perhaps with the previous two items completed, more of [Mic11] can be implemented in this extended environment. Lists, types, and ultimately all of scheme lay before us!

Implement a λ-expression-walker A λ-expression is basically a tree–and our α, β reductions are essentially walks of such a tree. Maybe this would blur the lines between prolog and scheme in a way that detracts from the document, but providing a second-order predicate to fold or map over a λ-expression may be neat.

Smarter Isomorphism Perhaps a more efficient tree-isomorphism algorithm can be implemented and ultimately a faster resugaring.

And I'm sure there are other neat directions people can take this.

Thanks for reading! I'd welcome any feedback via an "issues" report on the github repo.[Gor20]

A Complete Code Listing

```
expression(L) :- name(L).
expression(L) :- function(L).
expression(L) :- application(L).
function([lambda, V, B]) :- name(V), expression(B).
application([E1, E2])    :- expression(E1), expression(E2).
name(X)       :- not(is_list(X)), ground(X), X \= lambda.
beta_reduction([lambda, V, B], A, R) :- replace(V, B, A, R).
replace(V,V,A,A) :- name(V).
replace(V,W,_,W) :- name(W), V \= W.
replace(V,[lambda, V, B],_,[lambda, V, B]).
replace(V,[lambda, W, B],A,[lambda, W, S]) :-
    W \= V,
    replace(V,B,A,S).
replace(V, [E1, E2], A, [R1, R2]) :-
    replace(V,E1,A,R1),
    replace(V,E2,A,R2).
evaluate([E1, E2], R) :-
    evaluate(E1, R1), E1 \= R1,
    evaluate([R1, E2], R).
evaluate([E1, E2], R) :-
    beta_reduction(E1, E2, R1),
    evaluate(R1, R).
evaluate(L, L).
alpha_reduction(L, L) :- name(L).
alpha_reduction([E1, E2], [R1, R2]) :-
    alpha_reduction(E1, R1),
    alpha_reduction(E2, R2).
alpha_reduction([lambda, V, B], [lambda, X, ABB]) :-
    gensym(alpha_,X),
    replace(V, B, X, BB),
    alpha_reduction(BB, ABB).
canon(L, R) :- reset_gensym(alpha_), alpha_reduction(L, R).
isomorphic(A, B) :- canon(A, C), canon(B, C).
desugar([N,E], L, R) :- beta_reduction([lambda, N, L], E, R).
desugar_all(L, D, R) :- foldl(desugar, D, L, R).
resugar([N, E], L, N) :- isomorphic(E, L).
resugar(M, [lambda, V, B], [lambda, V, SB]) :- resugar(M, B, SB).
resugar(M, [E1, E2], [R1, R2]) :-
    resugar(M, E1, R1),
    resugar(M, E2, R2).
resugar(_, L, L).
resugar_all(L, D, R) :- foldl(resugar, D, L, S), S \= L, resugar_all(S, D, R).
resugar_all(L, _, L).
read_s(S)           :- get_char(C),                    read_s(C, S).
read_s('\n',[]).
read_s(' ',S)       :- get_char(D),                      read_s(D, S).
read_s('(',['('|S]) :- get_char(D),                      read_s(D, S).
read_s(')',[')'|S]) :- get_char(D),                      read_s(D, S).
read_s(C,[N|S])     :- name_char(C), read_name(C,N,D), read_s(D, S).
read_name(C,N,E) :-
    read_name_chars(S,C,E),
    atom_chars(N,S).
read_name_chars([C|S], C, E) :-
    name_char(C), !, % needed for io
    get_char(D),
    read_name_chars(S, D, E).
read_name_chars([],C,C) :- not(name_char(C)).
name_char(C) :- char_code(C,N), N >= 65, N =< 90. % upper case
name_char(C) :- char_code(C,N), N >= 97, N =< 122. % lower case.
name_char(C) :- char_code(C,N), N >= 48, N =< 57. % decimals
parse(S,L) :- parse(S, [], [L]).
parse(['('|T], R, [M|L]) :- parse(T, [')'|S], M), parse(S, R, L).
parse([')'|T], [')'|T], []).
parse([N|T], R, [N|L]) :- N \= ')', N \= '(', parse(T, R, L).
parse([],[],[]).
lambda_to_atom(L, R) :- parse(S, L), atomic_list_concat(S, ' ', R).
compute(L, D, R) :-
    desugar_all(L, D, DL),
    evaluate(DL, S),
    resugar_all(S, D, R).
execute_command(['define', N, L], D, [[N,R]|D]) :- compute(L, D, R).
execute_command(['execute', L], D,D) :-
    compute(L, D, R),
    lambda_to_atom(R,O),
    write(O), nl.
execute_command(['halt'],_,_) :- halt.
main_loop(OD) :-
    read_s(S),
    parse(S,L),
    execute_command(L,OD,ND),
    main_loop(ND).
main_loop(OD) :- write('Parse error'), nl, main_loop(OD).
main :- main_loop([]), halt.
```

References

[Gor20] Aaron Gorenstein. Logical lambda calculator. `https://github.com/agorenst/logical-lambda-calculator`, 2020.

[Mic11] Greg Michaelson. *An Introduction to Functional Programming Through Lambda Calculus*. Dover, 2011.

[SS94] Leon Sterling and Ehud Shapiro. *The Art of Prolog*. The MIT Press, 1994.