

Lambda Calculus for DrRacket

This document describes a program I developed to demonstrate [lambda calculus](#). It is implemented in the [LISP](#) dialect [SCHEME](#) as a language for the [DRRACKET IDE](#).

The project is part of a lecture series for computer scientists at the [Hamburg University of Applied Sciences](#). Lecture notes (in German) can be found at <https://weitz.de/files/mmi-skript.pdf>.

If you spot any errors in this document or the accompanying software, please send me an [email](#).

Installation

You first need to install RACKET following the instructions on <https://racket-lang.org/>.¹ Once you've done that, you should be able to follow along no matter whether you are using [Linux](#), [Windows](#), or [macOS](#). Please note that I neither have the time nor the expertise to help you if you have problems with the installation! You should be able to find assistance online.

Once the installation is finished, open a [command-line shell](#), move into the folder named `lambda` that was extracted from the [ZIP](#) archive where you found the [PDF file](#) you are reading and issue the following command (including the dot at the end):

```
raco pkg install --link .
```

That should be all.

If you want, you can also, from the same location, run a couple of tests like so:

```
raco test *.rkt
```

¹ RACKET might also be available in some other form, for example as a package of your [Linux distribution](#).

Updates

I might occasionally update the software to fix errors or add features. And the document you are reading might also get updated. The newest version will always be available at the URL <https://weitz.de/files/lambda.zip>. Check the timestamp [at the end](#).

To update the software, just replace the old files (ending in `.rkt`) with the new ones. As long as they stay in the same place, RACKET will automatically pick up the changes.

The Syntax of Terms

The following text explains how to use the program from within DRACKET. It is *not* intended to teach you lambda calculus.

Once the software has been installed, you activate it by changing the first line of the upper window (which in DRACKET is called the *definitions window*) to

```
#lang lambda
```

and press the *Run* button (or the `Ctrl-r` key combination). After this has been done, both windows will “understand” the new syntax. It works as follows:

- We will define [recursively](#) what a **term** is.
- Sequences of [ASCII](#) letters, [decimal digits](#), [underline](#) characters, or [hyphens](#) are recognized as **identifiers** as long as they start with a letter. The following are nine examples for valid identifiers. (And they are all different, i. e., the system is [case-sensitive](#).)

x	x1	x2
foo	Foo	FOO
foo-bar	foo_bar	fooBar

- Each identifier (except for `lambda`) is an **atom**.
- An **abstraction** is a sequence of the identifier `lambda` (or alternatively [the Unicode symbol \$\lambda\$](#)) followed by an identifier (which is *not* `lambda`), followed by a [dot](#), followed by a term.² There must be horizontal [whitespace](#) between `lambda` and the identifier while horizontal whitespace between the other parts is optional. Here are examples for simple abstractions.

```
lambda x . x  
 $\lambda$ x.y
```

²You can enter λ in DRACKET by typing `\lam` and then `Alt- $\backslash$` . Alternatives are explained [here](#).

The identifier before the dot is called the **variable** of the abstraction and the term behind the dot is its **body**.

- Each abstraction is a term.
- An **application** is a sequence of two atoms. The atoms must be separated by horizontal whitespace if they could otherwise be confused with a single identifier. Here are examples for simple applications.

```
x y
foo Foo
```

- Each application is a term.
- A term enclosed in parentheses is an atom.

Now, with the full definition in place, we have many more examples of terms. Here are a few. (Note that in all three cases the space behind the first closing parenthesis isn't strictly necessary but might increase readability.)

```
(λx.x) y
(lambda x.y) (lambda y.x)
(x y) z
```

Parentheses

Mathematicians are lazy and want to avoid unnecessary work. A typical symptom of this is the invention of rules for when parentheses can be omitted.³ We will now do that as well. Look at the following two terms:

```
lambda x.(y z)
(lambda x.y) z
```

The first one is an abstraction with the variable x and the body $y\ z$ (which is an application), the second term is an application where the right atom is the identifier z and the left atom is the abstraction $\lambda x.y$.

The first rule is that application (i. e. [concatenation](#)) has higher [precedence](#) than abstraction. This means the system will assume we mean the first of the two terms above if we just type this:⁴

```
lambda x.y z
```

³Like writing $3 + 4 \cdot 5$ instead of $3 + (4 \cdot 5)$.

⁴In other words, the space between y and z “pulls” the y with more “force” than the dot.

Now look at these two terms:

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

Both are applications built from identifiers and shorter applications and they are different. The usual convention here is that application is **left-associative**. This means that the system will assume we mean the first alternative if we just type this:

```
x y z
```

That's already all there is to know about parentheses, but it's probably enough to be a bit confused in the beginning. When the system displays terms, it will always try to use as few parentheses as possible. You, on the other hand, should be a bit more generous with them, especially if you're unsure about the precedence rules. Gratuitous parentheses won't hurt.

Oh, there's one more thing: As a general rule, vertical whitespace (i. e., the end of a line) will terminate the **parsing** of a term. The system won't accept something like this:

```
lambda x .
x
```

However, if an opening parenthesis has no closing counterpart, terms are allowed to continue on the next line. This means that the following *is* allowed (and equivalent to $\lambda x. x$):

```
(lambda x .
x)
```

Reduction

The main job of the program is **reduction**. If you type a term into the lower window (called *interactions window* in DRACKET), the system will perform *normal order reduction* until the term is β -normal (if that is possible). Try entering the following terms (one by one) into the interactions window and see what happens.

```
(lambda x.x) y
(lambda x.x x) z
lambda x.lambda y.x
((lambda x.lambda y.x) z) w
```

You can also enter terms into the definitions window and press the *Run* button.⁵ The system

⁵Note that if you make changes in the definitions window without pressing *Run*, the system will show a warning if you afterwards do something in the interactions window. That's standard DRACKET behavior and not a special

will then show the reductions in the interactions window.

You can also precede a term with `->` (or the [Unicode rightwards arrow](#)). In this case, the system will show the reduction step by step. Try this:

```
-> (((lambda x.lambda y.lambda z.x z y) one) three) two
```

This will only work in the interactions window, though.

Comments

The [number sign](#) `#` serves as the [comment](#) character. Everything including this character up to (but excluding) the end of the line is ignored by the system.

This means the following would be legal in the definitions window:

```
# The next term would have fit on one line
(lambda x                # x is the variable
  .                      # we need a dot here
  x)                    # and that's the body
```

It will result in $\lambda x. x$ in the interactions window.

New Variables

Try this:

```
(lambda x . (lambda y . x)) y
```

The result should look like $\lambda y1. y$. The main point of this example is that the result is *not* $\lambda y. y$ which we would get from a naive substitution in the sense of “replace all occurrences of x in $\lambda y. x$ with y .” That would have changed the meaning of the inner abstraction!

The system spots such situations (called *variable capture*) and introduces new variables to circumvent them. If the variable was called y , then it tries $y1$, $y2$, and so on until it finds one that doesn't cause problems.⁶

Abstractions with several variables

The system will accept “illegal” terms like

feature of the lambda language.

⁶To keep the implementation simple, it avoids *all* variables although this is not always strictly necessary.

```
lambda x y.(y x)
```

and interpret them as [currying](#), i. e., the output in this case will be the following:

```
λx.λy.y x
```

Abbreviations

As it can get tedious to type lambda terms, you can introduce abbreviations for them. This is done in the definitions window by writing an identifier followed by either `:=` (more common) or just `=` (shorter) followed by a term. The identifier can then be used anywhere instead of this term. As an example, type this into the definitions window and then press *Run*:

```
I := lambda x.x
```

You can now type

```
I z
```

in the interactions window and you will get `z` as a result.

Note that this is not *textual*, but *structural* substitution. This implies two things: First, you can only abbreviate *terms*, not arbitrary sequences of characters. For example, you cannot abbreviate `lambda x` or something like that. And second, the abbreviation is substituted as an *atom* which means that it is treated as if there were parentheses around it.⁷

Abbreviations can refer to other abbreviations. For example, this works:

```
U = λx.x x
F = lambda x . U x (U x)
```

Now try `F w` or even better `-> F w`.

Sometimes, you want to see all the details *without* abbreviations, though. You can achieve this by replacing `->` with `-->`. This also applies to the Church numerals introduced in the [next section](#).

By convention, abbreviations use uppercase identifiers while variables are typically lowercase letters. But that's not a rule enforced by the system.

Church numerals

The system recognizes [Church numerals](#), i. e., it will for example convert the input `2` to this:

⁷If resolving `I z` had lead to `lambda x.x z`, the result would have been different!

```
λf.λn.f (f n)
```

Church numerals will also be shown in the system's output, but only once at least one [abbreviation](#) has been defined. This is done as a kind of “safety measure” in order not to confuse beginners who entered something like this:

```
lambda s.lambda x.x
```

They should not see 0 as an answer ...

When the System Seems to Hang

The ZIP file contains a file `example.rkt` with some definitions from the [Wikipedia page about the lambda calculus](#). These definitions include the following:

```
U := λx.x x  
Omega := U U
```

Omega (usually written Ω) is the smallest term that has no normal form. This means that the system will get into an endless loop trying to reduce it. There's no way to avoid something like that.⁸

The good news is that DRRACKET has a working *Stop* button in the upper right corner. Try it by entering `-> Omega` in the interactions window – and then stop the computation⁹ before your CPU gets too hot ...

Prof. Dr. Edmund Weitz

Hamburg, September 2, 2026

⁸Unless you want to introduce some arbitrary limit where normalization has to stop after, say, 42 steps.

⁹It *will* be stopped but it might take a while until you see this because of all the output that was already generated and not yet displayed.