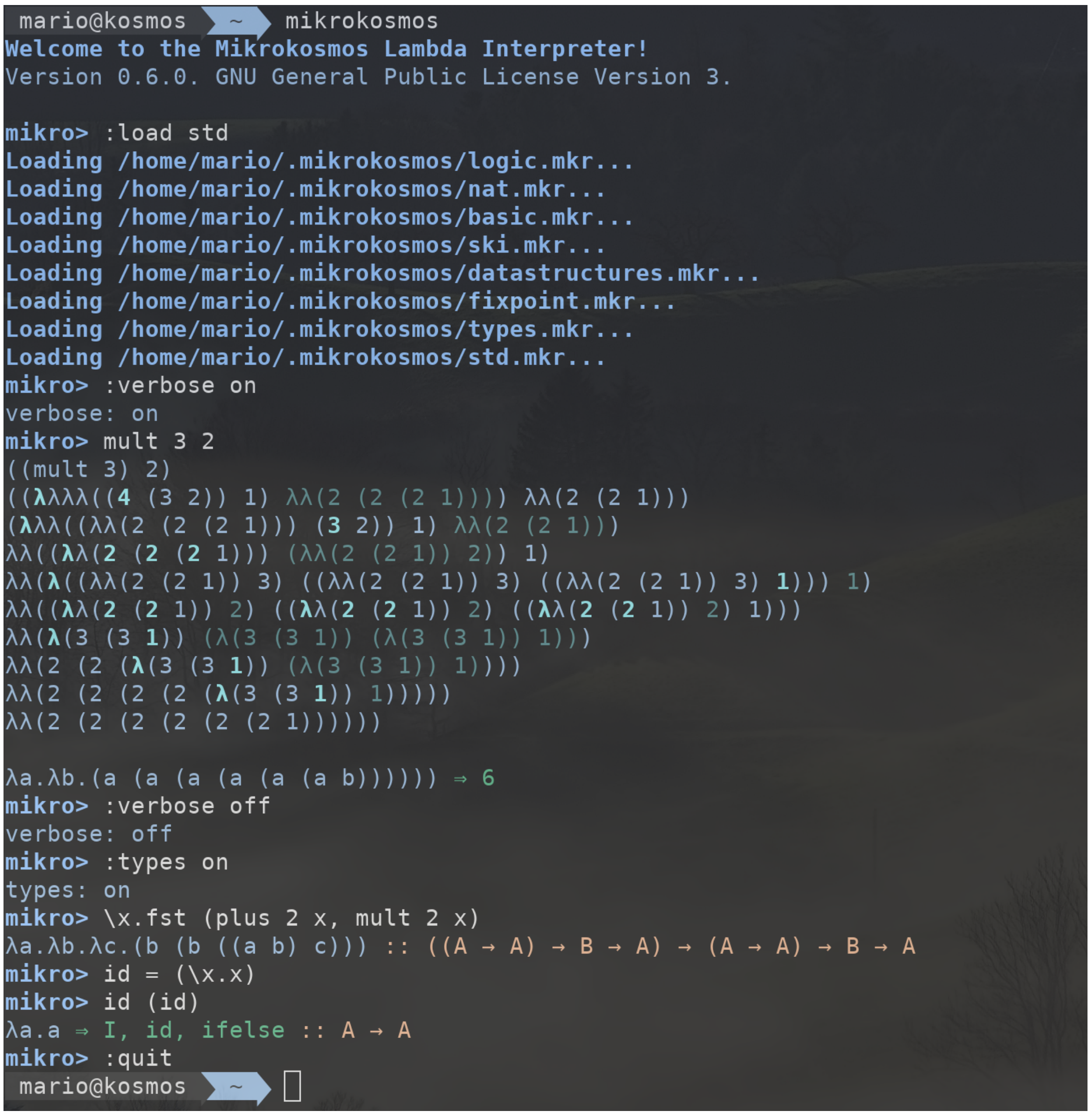
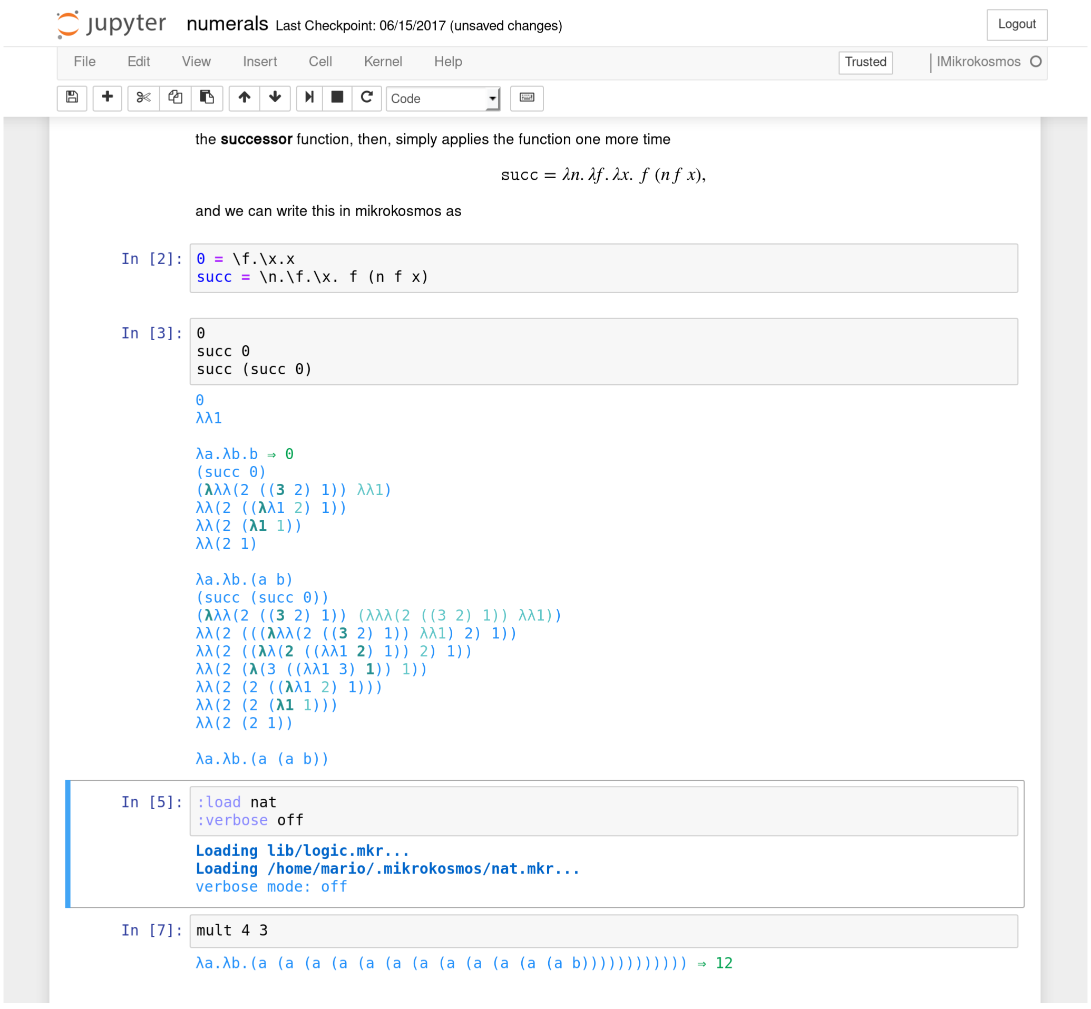


```
mario@kosmos  ~  mikrokosmos
Welcome to the Mikrokosmos Lambda Interpreter!
Version 0.6.0. GNU General Public License Version 3.

mikro> :load std
Loading /home/mario/.mikrokosmos/logic.mkr...
Loading /home/mario/.mikrokosmos/nat.mkr...
Loading /home/mario/.mikrokosmos/basic.mkr...
Loading /home/mario/.mikrokosmos/ski.mkr...
Loading /home/mario/.mikrokosmos/datastructures.mkr...
Loading /home/mario/.mikrokosmos/fixpoint.mkr...
Loading /home/mario/.mikrokosmos/types.mkr...
Loading /home/mario/.mikrokosmos/std.mkr...
mikro> :verbose on
verbose: on
mikro> mult 3 2
((mult 3) 2)
((λλλ((4 (3 2)) 1) λλ(2 (2 (2 1)))) λλ(2 (2 1)))
(λλ((λλ(2 (2 (2 1))) (3 2)) 1) λλ(2 (2 1)))
λλ((λλ(2 (2 (2 1))) (λλ(2 (2 1)) 2)) 1)
λλ(λ((λλ(2 (2 1)) 3) ((λλ(2 (2 1)) 3) ((λλ(2 (2 1)) 3) 1))) 1)
λλ((λλ(2 (2 1)) 2) ((λλ(2 (2 1)) 2) ((λλ(2 (2 1)) 2) 1)))
λλ(λ(3 (3 1)) (λ(3 (3 1)) (λ(3 (3 1)) 1)))
λλ(2 (2 (λ(3 (3 1)) (λ(3 (3 1)) 1))))
λλ(2 (2 (2 (2 (λ(3 (3 1)) 1)))))
λλ(2 (2 (2 (2 (2 (2 1))))))

λa.λb.(a (a (a (a (a (a b)))))) ⇒ 6
mikro> :verbose off
verbose: off
mikro> :types on
types: on
mikro> \x.fst (plus 2 x, mult 2 x)
λa.λb.λc.(b (b ((a b) c))) :: ((A → A) → B → A) → (A → A) → B → A
mikro> id = (\x.x)
mikro> id (id)
λa.a ⇒ I, id, ifelse :: A → A
mikro> :quit
mario@kosmos  ~  
```

Figure 2: Mikrokosmos interpreter session.


jupyter
numerals Last Checkpoint: 06/15/2017 (unsaved changes)
Logout
File Edit View Insert Cell Kernel Help
Trusted
IMikrokosmos
Code

the **successor** function, then, simply applies the function one more time

$$\text{succ} = \lambda n.\,\lambda f.\,\lambda x.\; f\;(n\,f\;x),$$

and we can write this in mikrokosmos as

```
In [2]: 0 = \f.\x.x
        succ = \n.\f.\x. f (n f x)
```

```
In [3]: 0
        succ 0
        succ (succ 0)
```

```
0
λλ1

λa.λb.b ⇒ 0
(succ 0)
(λλλ(2 ((3 2) 1)) λλ1)
λλ(2 ((λλ1 2) 1))
λλ(2 (λ1 1))
λλ(2 1)

λa.λb.(a b)
(succ (succ 0))
(λλλ(2 ((3 2) 1)) (λλλ(2 ((3 2) 1)) λλ1))
λλ(2 (((λλλ(2 ((3 2) 1)) λλ1) 2) 1))
λλ(2 ((λλ(2 ((λλ1 2) 1)) 2) 1))
λλ(2 (λ(3 ((λλ1 3) 1)) 1))
λλ(2 (2 ((λλ1 2) 1)))
λλ(2 (2 (λ1 1)))
λλ(2 (2 1))

λa.λb.(a (a b))
```

```
In [5]: :load nat
        :verbose off
```

```
Loading lib/logic.mkr...
Loading /home/mario/.mikrokosmos/nat.mkr...
verbose mode: off
```

```
In [7]: mult 4 3
```

```
λa.λb.(a (a (a (a (a (a (a (a (a (a (a (a b)))))))))))) ⇒ 12
```

Figure 3: Jupyter notebook Mikrokosmos session.

```
for line in code.split('\n'):
    # Send code to mikrokosmos
    self.mikro.sendline(line)
    self.mikro.expect('mikro> ')

    # Receive and filter output from mikrokosmos
    partialoutput = self.mikro.before
    partialoutput = partialoutput.decode('utf8')
    output = output + partialoutput
```

A pip installable package has been created following the Python Packaging Authority guidelines. [3] This allows the kernel to be installed directly using the pip python package manager.

```
sudo -H pip install imikrokosmos
```

7.4 CODEMIRROR LEXER

CodeMirror [4] is a text editor for the browser implemented in Javascript. It is used internally by the Jupyter Notebook.

A CodeMirror lexer for Mikrokosmos has been written. It uses Javascript regular expressions and signals the ocurrence of any kind of operator to CodeMirror. It enables syntax highlighting for Mikrokosmos code on Jupyter Notebooks. It comes bundled with the kernel specification and no additional installation is required.

```
CodeMirror.defineSimpleMode("mikrokosmos", {
    start: [
    // Comments
    {regex: /\#.*/,
    token: "comment"},
    // Interpreter
    {regex: /\:load|\:verbose|\:ski|\:restart|\:types|\:color/,
    token: "atom"},
    // Binding
    {regex: /(.*?)(\s*)(=)(\s*)(.*?)$/,
    token: ["def",null,"operator",null,"variable"]},
    // Operators
    {regex: /[=!]+/,
    token: "operator"},
    ],
    meta: {
```

3 : The PyPA packaging user guide can be found in its official page: https://packaging.python.org/
4 : Documentation for CodeMirror can be found in its official page: https://codemirror.net/

```
        dontIndentStates: ["comment"],
        lineComment: "#"
    }
}
```

7.5 JUPYTERHUB

JupyterHub manages multiple instances of independent single-user Jupyter notebooks. We used it to serve Mikrokosmos notebooks and tutorials to students studying λ-calculus.

In order to install Mikrokosmos on a server and use it as `root` user, we need

- to clone the libraries into `/usr/lib/mikrokosmos`. They should be available system-wide.
- to install the Mikrokosmos interpreter into `/usr/local/bin`. In this case, we chose not to install Mikrokosmos from source, but simply copy the binaries and check the availability of the `ncurses` library.
- to install the Mikrokosmos Jupyter kernel as usual.

Our server used a SSL certificate; and OAuth autentication via GitHub. Mikrokosmos tutorials were installed for every student.

7.6 CALLING MIKROKOSMOS FROM JAVASCRIPT

The GHCjs[5] compiler allows transpiling from Haskell to Javascript. Its foreign function interface allows a Haskell function to be passed as a continuation to a Javascript function.

A particular version of the `Main.hs` module of Mikrokosmos was written in order to provide a `mikrokosmos` function, callable from Javascript. This version includes the standard libraries automatically and reads blocks of texts as independent Mikrokosmos commands. The relevant use of the foreign function interface is showed in the following code

```haskell
foreign import javascript unsafe "mikrokosmos = $1"
    set_mikrokosmos :: Callback a -> IO ()
```

which provides `mikrokosmos` as a Javascript function once the code is transpiled. In particular, the following is an example of how to call Mikrokosmos from Javascript

```javascript
button.onclick = function () {
   editor.save();
   outputcode.getDoc().setValue(mikrokosmos(inputarea.value).mkroutput);
```

5 : The GHCjs documentation is available on its web page `https://github.com/ghcjs/ghcjs`

```
    textAreaAdjust(outputarea);
}
```

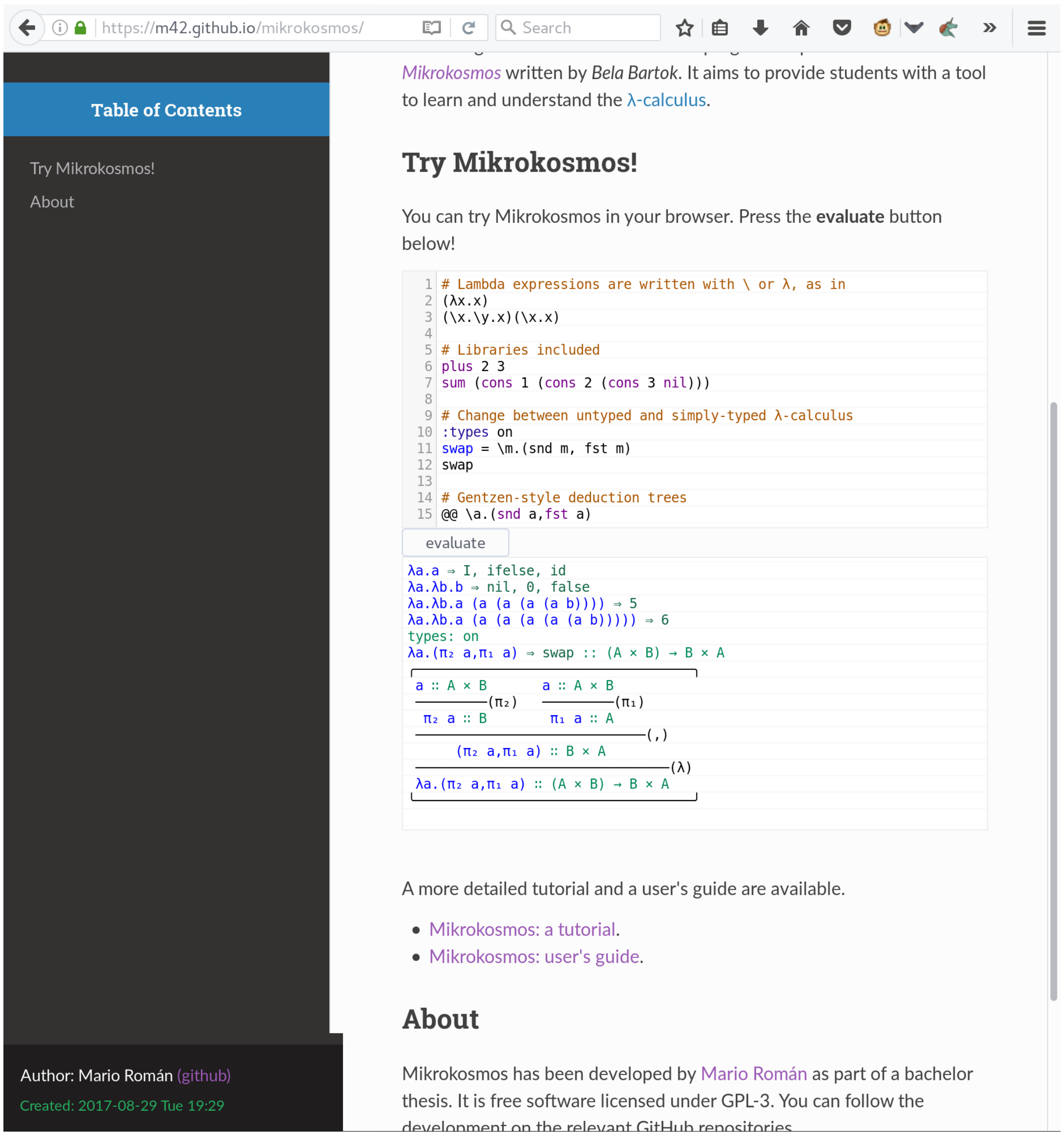
A small script has been written in Javascript to help with the task of embedding Mikrokosmos into a web page. It and can be included directly from

`https://m42.github.io/mikrokosmos-js/mikrobox.js`

using GitHub as a CDN. It will convert any HTML script tag written as follows

```
<div class="mikrojs-console">
<script type="text/mikrokosmos">
(λx.x)
... your code
</script>
</div>
```

into a CodeMirror pad where Mikrokosmos can be executed. The Mikrokosmos tutorials are an example of this feature and can be seen on Figure 4.



https://m42.github.io/mikrokosmos/

Search

Mikrokosmos written by *Bela Bartok*. It aims to provide students with a tool to learn and understand the λ-calculus.

Table of Contents

Try Mikrokosmos!

About

Try Mikrokosmos!

You can try Mikrokosmos in your browser. Press the **evaluate** button below!

```
# Lambda expressions are written with \ or λ, as in
(λx.x)
(\x.\y.x)(\x.x)

# Libraries included
plus 2 3
sum (cons 1 (cons 2 (cons 3 nil)))

# Change between untyped and simply-typed λ-calculus
:types on
swap = \m.(snd m, fst m)
swap

# Gentzen-style deduction trees
@@ \a.(snd a,fst a)
```

evaluate

```
λa.a ⇒ I, ifelse, id
λa.λb.b ⇒ nil, 0, false
λa.λb.a (a (a (a b)))) ⇒ 5
λa.λb.a (a (a (a (a b))))) ⇒ 6
types: on
λa.(π₂ a,π₁ a) ⇒ swap :: (A × B) → B × A
┌───────────────────────────────────────┐
 a :: A × B       a :: A × B
 ─────────(π₂)    ─────────(π₁)
  π₂ a :: B        π₁ a :: A
 ─────────────────────────────(,)
       (π₂ a,π₁ a) :: B × A
 ────────────────────────────────(λ)
 λa.(π₂ a,π₁ a) :: (A × B) → B × A
└───────────────────────────────────────┘
```

A more detailed tutorial and a user's guide are available.

- Mikrokosmos: a tutorial.
- Mikrokosmos: user's guide.

About

Mikrokosmos has been developed by Mario Román as part of a bachelor thesis. It is free software licensed under GPL-3. You can follow the

Author: Mario Román (github)

Created: 2017-08-29 Tue 19:29

Figure 4: Mikrokosmos embedded into a web page.

8

PROGRAMMING ENVIRONMENT

8.1 CABAL, STACK AND HADDOCK

The Mikrokosmos documentation as a Haskell library is included in its own code. It uses **Haddock**, a tool that generates documentation from annotated Haskell code; it is the *de facto* standard for Haskell software.

Dependencies and packaging details for Mikrokosmos are specified in a file distributed with the source code called `mikrokosmos.cabal`. It is used by the package managers **stack** and **cabal** to provide the necessary libraries even if they are not available system-wide. The **stack** tool is also used to package the software, which is uploaded to ***Hackage***.

8.2 TESTING

Tasty is the Haskell testing framework of our choice for this project. It allows the user to create a comprehensive test suite combining multiple types of tests. The Mikrokosmos code is tested using the following techniques

- **unit tests**, in which individual core functions are tested independently of the rest of the application;
- **property-based testing**, in which multiple test cases are created automatically in order to verfiy that a specified property always holds;
- **golden tests**, a special case of unit tests in which the expected results of an IO action, as described on a file, are checked to match the actual ones.

We are using the **HUnit** library for unit tests. It tests particular cases of type inference, unification and parsing. The following is an example of unit test, as found in `tests.hs`. It checks that the type inference of the identity term is correct.

```haskell
-- Checks that the type of λx.x is exactly A → A
testCase "Identity type inference" $
  typeinference (Lambda (Var 1)) @?= Just (Arrow (Tvar 0) (Tvar 0))
```

We are using the **QuickCheck** library for property-based tests. It tests transformation properties of lambda expressions. In the following example, it tests that any De Bruijn expression keeps its meaning when translated into a λ-term.

```
-- Tests if translation preserves meaning
QC.testProperty "Expression -> named -> expression" $
  \expr -> toBruijn emptyContext (nameExp expr) == expr
```

We are using the **tasty-golden** package for golden tests. Mikrokosmos can be passed a file as an argument to interpret it and show only the results. This feature is used to create a golden test in which the interpreter is asked to provide the correct interpretation of a given file. This file is called `testing.mkr`, and contains library definitions and multiple tests. Its expected output is `testing.golden`. For example, the following Mikrokosmos code can be found on the file

```
:types on
caseof (inr 3) (plus 2) (mult 2)
```

and the expected output is

```
-- types: on
-- λa.λb.(a (a (a (a (a (a b)))))) ⇒ 6 :: (A → A) → A → A
```

8.3 VERSION CONTROL AND CONTINUOUS INTEGRATION

Mikrokosmos uses **git** as its version control system and the code, which is licensed under GPLv3, can be publicly accessed on the following GitHub repository:

https://github.com/M42/mikrokosmos

Development takes place on the `development` git branch and permanent changes are released into the `master` branch. Some more minor repositories have been used in the development; they directly depend on the main one

- https://github.com/m42/mikrokosmos-js
- https://github.com/M42/jupyter-mikrokosmos
- https://github.com/M42/mikrokosmos-lib

The code uses the **Travis CI** continuous integration system to run tests and check that the software builds correctly after each change and in a reproducible way on a fresh Linux installation provided by the service.

9

PROGRAMMING IN UNTYPED λ-CALCULUS

This section explains how to use untyped λ-calculus to encode data structures and useful data, such as booleans, linked lists, natural numbers or binary trees. All this is done on pure λ-calculus avoiding the addition of new syntax or axioms.

This presentation follows the Mikrokosmos tutorial on λ-calculus, which aims to teach how it is possible to program using untyped λ-calculus without discussing technical topics such as those we have discussed on the chapter on untyped λ-calculus. It also follows the exposition on [Sel13] of the usual Church encodings.

All the code on this section is valid Mikrokosmos code.

9.1 BASIC SYNTAX

In the interpreter, λ-abstractions are written with the symbol \, representing a λ. This is a convention used on some functional languages such as Haskell or Agda. Any alphanumeric string can be a variable and can be defined to represent a particular λ-term using the = operator.

As a first example, we define the identity function (`id`), function composition (`compose`) and a constant function on two arguments which always returns the first one untouched (`const`).

```
id = \x.x
compose = \f.\g.\x.f (g x)
const = \x.\y.x
```

Evaluation of terms will be presented as comments to the code,

```
compose id id
-- [1]: λa.a ⇒ id
```

It is important to notice that multiple argument functions are defined as higher one-argument functions that return another functions as arguments. These intermediate functions are also valid λ-terms. For example