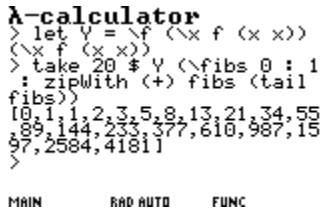
# λ-calculator

λ-calculator is a lambda expression evaluator for the TI-89/92+/V200:




It draws a lot of inspiration from Haskell, but takes several liberties. Like Haskell, it is purely functional and lazy. However, unlike Haskell, it is dynamically typed. This has a handful of advantages, and several disadvantages. One advantage is, you can write the Y combinator unmodified, and it will work. Namely:

```
let Y = \f (\x f (x x)) (\x f (x x))
```

λ-calculator is far from perfect, and may crash your calculator, so use with caution (meaning archive your files so you won't lose them).

## Installation

- If you have a TI-89 or TI-89 Titanium:
  - Install **`lambda.89z`** and **`lambda.89y`**
- If you have a TI-92+ :
  - Install **`lambda.9xz`** and **`lambda.9xy`**
- If you have a Voyage 200:
  - Install **`lambda.v2z`** and **`lambda.v2y`**

**`prelude.89t`** is optional, but recommended, as it provides a few functions to look at and use (as well as the `λ-calculator` splash text).

**`samples`** is optional, and contains more sample code to look at and use.

Running `lambda()` will start the interpreter (which takes about 6 seconds to load), and will run the commands in the TEXT file named `prelude` (if present). Alternatively, `lambda("filename")` will run the commands in the TEXT file named `filename` rather than `prelude`.

**Note:** Archive your files before running. Doing so will give the interpreter more memory to work with, and *will save your files if the calculator crashes*.

# Syntax

- **Lambda:** `\var expr`
  Note: The λ character may also be used.
  - `\x x + 1`
  - `\x \y x + y`
- **Function application:** `f x`
  - `f (g x)`
  - `(+) 1 2`
- **Infix:** `expr1 op expr2`
  Note: fixity is like in Haskell, except vars can be made infix, operators can be made prefix, and there is no `` `div` `` syntax.
  - `2 + 2`
  - `1 * 2 + 3 * 4 == 14`
  - `15 div 5`
- **Section:** Just like in Haskell.
  - `map (*2) (1..10)`
  - `foldl (+) (1..10)`
  - ```
    mapM_ (putStrLn . ("We are " ++))
          ["One", "Strong", "Jaguars"]
    ```
- **If-then:** `if predicate then expr1 else expr2`
  - `if x<=y then x else y`
  - `if x==0 then 0 else if y==0 then undefined else x/y`
- **Integer:**
  - `123`
  - `0xDEAD` *(hexadecimal)*
  - `0c31` *(octal)*
  - `0b1010` *(binary)*
- **List:** `[x (, y (, …))]`
  Note: the `[1..10]` syntax is not implemented. However, there are `..` and `...` operators which cleverly (ab)use infix and sections so you can say `(1..10)` and `(1...)`.
  - `[]`
  - `[1,2,3]`
  - `[[1,2],[3,4]]`
  - `[5, "hello"]` *(allowed due to dynamic typing)*
- **String:** Similar to Haskell string literals.
  Note: "gaps" and escapes like `\NUL ... \^A \^@ \^[` are not implemented.
  - `putStr "Hello, world\n"`
- **Char:** Single-character literal
  - `putChar '\n'`
    ```

## Syntax

Additionally, at the top level only:

- **Let binding:**
  - `let fix = \f f (fix f)`
- **Action binding:**
  - `line <- getLine`
- **Fixity declaration:**
  - Left-associative:
    - `infixl 6 +`
    - `infixl 7 div`
  - `Right-associative:`
    - `infixr 5 :`
  - `Non-associative:`
    - `infix 4 ==`
  - `Prefix:`
    - `prefix +`
      (makes it so you can say + 1 2 (but not 1 + 2) in future commands)
- **Import:**
  - `import filename`
  - `import dir\filename`

# Features

λ-calculator is a rather limited language at its core:

- The only types are:
  - Integer (64-bit)
    - No floats, no arbitrary-precision integers.
  - Char (8-bit)
  - Boolean
  - `()` (just like Haskell's () )
  - List
  - Function
  - IO action
  - return
    - This is like Haskell's return. It's a rather ugly hack to provide monads in a dynamically-typed setting.
  - undefined (throws an error)
- Lists don't get garbage collected during traversal.
  For example, `sum (1..1000)` runs out of memory.
  Note: This is not a flaw in the garbage collector, but in how the evaluator is written. When a function is called with a list argument, the list head lingers in the stack, even though it's never used again.

However, for what it is, it has a lot of bells and whistles, including, but not limited to:

- A lot of functions straight out of Haskell (press `CATALOG` to see the exhaustive list).
  - However, `read` and `show` (conversion from strings to/from values) are not implemented, and are dearly missed.
- Line editing via the AMS text editor, and history!
- Monads for IO, list, and function:
  - `getLine >>= putStrLn`
    (get a line, and echo it)
  - `sequence $ replicate 4 [0,1]`
    (count to 16 in binary)
  - `(zip <*> tail) (1..5)`
    (list pairs of adjacent items)
- File IO with `readFile`/`writeFile`/`appendFile` and their binary variants.
  - However, it's not really that useful due to the interpreter's slowness, poor memory management, and lack of `read` and `show`.
- A few AMS-specific actions, such as ngetchx and setFont.