LLambdai Documentation

Waylon Cude

September 2017

1 Introduction

This is the reference for the LLambda programming language. LLambda is a programming language based off of lambda calculus with dynamic typing. It has Lisp-inspired lists, `let` declarations similar to Haskell, and a basic loop expression to simplify looping. LLambda also has infix notation in the same style as Haskell.

LLambda was originally created as a hybrid learning project and irc programming language, so there were some interesting design decisions made in the development process. LLambda is in heavy development and as such this documentation will likely be out of date in the near future.

2 Types

LLambda has 6 types:

- Number
- String
- Nil
- Func
- List
- IO

Values of the Number type store a floating point number with double precision. In addition to being able to use Numbers for math, Numbers are also the primary method of triggering conditionals: 0 functions as false, while every other number functions as true.

The String type represents a UTF-8 string. The only operation you can currently perform on strings is concatenation, limiting their usefulness. However, strings are currently the easiest way to use IO in LLambda. There is currently no way to access individual characters of a String, and they are likely going to be deprecated in the future and replaced with lists of characters.

Nil is a special value that always functions as false. Nil values are also used to represent the end of a List.

Func is the type of both built-in functions and lambda abstractions. Lambda abstractions are what you use to define your own functions, while built-in functions are built in to the interpreter and always available. There are a number of built-in functions, which will be presented later in this reference. There is also a standard library which contains functions written in LLambda, these can be loaded in with the `use` expression and then called like normal functions.

The List type represents a Lisp-style list. Lists are constructed with `cons` cells, similar to two-element tuples. You can access the first element of the cons cell with `car` and the second element with `cdr`. Lists are typically ended with a nil value to make traversing easy. However, you can stick any type in either side. The following example constructs a list, then multiples the two elements of the list together:

```
let list = cons 2 (cons 4 nil);
let first = car list;
let second = car ( cdr list );
( * first second )
```

You can mix types inside of Lists, so the following list is valid:

```
cons "string" (cons 4 nil).
```

The last type is IO. An IO object represents an IO stream that you can read and write to. There are different functions for creating different IO objects, like files or network connections. Once you have an IO object to work with you can use the `read` and `write` functions to work with it. This is currently the only way of doing IO. IO makes use of crazy side-effects, and is likely to be reworked in the future.

3 The Language

3.1 Variables

Variables in LLambdai are created through `let` statements or through variable binding. The syntax of `let` is `let N = Y;`, where N is a variable name and Y is an expression. The values of Y gets assigned to the variable N, which you can use freely through the rest of your program. Lambdas let you achieve the same thing. In fact, `let` statements are actually just syntactic sugar over this process. The let statement `let x = 2; x` corresponds to the following code:

```
( \x.x ) 2
```

3.2 Functions

The syntax for calling functions is `(F X ...)`, where F is a function and X ... are the values that are being passed in to the function. To call the multiplication function, you could write this:

```
* 5 4
```

Lambdas are how you define your own functions. Their syntax is:
`\x .... T`
T is a term, and x ... is the list of arguments to the function. When you call a function the values you pass in will get assigned to the variables in the argument list, and the term T will get evaluated. The following lambda will double whatever you pass in:
`\x.( * x 2 )`
You can store lambdas inside of variables for easier access:
```
let double = \x.( * x 2 );
double 4
```
There is also a range of built-in functions, which are explained in a later section.

You can also call functions with infix notation. To do this, you put the first argument before the function name and surround the function name with backticks. The following is an example of infix addition:
`5 ‘+‘ 10`
Currently all functions, including single-character operators, have to be surrounded by backticks. Hopefully in the future operators will not require backticks.

3.3 The Standard Library

There is a small library of functions written in LLambda. You can load modules of the standard library with the `use` expression, `use` takes the name of the module you want to load and is ended with a semicolon. To load the list module of the standard library you would use the following code:
`use "std::list" ;` After loading a module you can call all of its functions as you would any other function.

3.4 Conditionals

LLambda has a single built-in conditional function, `ife`, which takes a condition and two expressions. If the condition evaluates to true then the first expression is ran, otherwise the second expression is ran. If you use a number as the condition the the first expression will get ran if the number is not 0, and the second expression will get ran if the number is 0. If you give a nil as your condition then the second expression will always be executed. If you pass anything else as your condition then the first expression will be executed. There are two comparison operators, `<` and `>`, which function as you expect. You can use subtraction as a comparison operator, it functions like not-equals:
```
ife (- 105 90) "105 is not 90" "not returned"
```
This single conditional should be powerful enough to build more ergonomic conditionals, and should be sufficient to provide Turing completeness.

3.5 Loops

There are two ways to repeat code in LLambda. The first is recursion. To double a number three times you could do something like the following, which will return 8:

```
let x = 1;
let count = 3;
let double = \x count .
      ife count (double (* x 2) (- count 1) ) (x)
double x count
```

Recursion is usually your best option, but if you need something faster you can use the built in `loop` expression. `loop` takes a number of times to repeat and a series of `let` statements that will be repeated. Originally loop would just repeat the code a certain number of times, expanding

```
loop 3 { let x = * x 2 }
```

to

```
let x = * x 2;
let x = * x 2;
let x = * x 2;
```

This has since been optimized into a for-loop in the interpreter, but the effect is still the same. Notably, you can use any expression that evaluates to a number as `loop` expressions are evaluated at run-time instead of parse-time. The following code doubles x three times, then returns the number 8:

```
let x = 1; loop 3 { let x = ( * x 2 ) } x
```

4 Built-in Functions

4.1 Math

4.1.1 *

The `*` operator multiplies two numbers together.

4.1.2 /

The `/` operator divides one number by another.

4.1.3 -

The `-` operator subtracts one number from another

4.1.4 +

The `+` operator can do a few different things, depending on the types of the arguments. If you pass in two numbers the it will do addition. If you pass in either a string and a number or two strings then it will perform string concatenation. The following example returns the string "Hello World!":

```
+ "Hello " "World!"
```
The following example returns 15:
```
+ 7 8
```

4.1.5 %

The `%` operator performs the modulus operation.

4.1.6 floor

`floor` takes the floor of a number, rounding it down to the nearest integer.

4.1.7 ceil

`ceil` takes the ceiling of a number, rounding it up to the nearest integer.

4.1.8 round

`round` rounds a number to the nearest integer.

4.1.9 rand

`rand` returns a random number in the interval [0, 1)

4.2 Conditionals

4.2.1 >

The `>` operator takes two arguments and returns 1 if the first argument is greater than the second argument and 0 otherwise. The following example returns 0:
```
< 5 8
```

4.2.2 <

The `<` operator takes two arguments and returns 1 if the first argument is less than the second argument. The following example returns 1:
```
< 5 8
```

4.2.3 ife

`ife` takes three arguments. The first argument is a condition to check. The next two arguments are expressions. If the condition evaluates to true then the first expression is executed. If the condition evaluates to false then the second expression is executed.
The following example will return the string "false":
```
ife 0 "true" "false"
```
The following example will return the string "true":
```
ife (5 < 8) "true" "false"
```

4.3 List processing

4.3.1 cons

`cons` is used for the construction of lists. It returns a list where the first argument is the head of the list and the second argument is the tail of the list. Nil is the standard way to end a list, but this is not required by the List type. To make a single argument list you would use the following code: `cons 3 nil` . To make a two argument list you would use the following code: `cons 2 (cons 3 nil)`

4.3.2 car

`car` returns the head of a list.
`car ( cons 2 nil )` returns 2.

4.3.3 cdr

`cdr` returns the tail of a list.
`cdr ( cons 2 nil )` returns nil.

4.4 IO

4.4.1 read

`read` reads a single byte from an IO object. `read` returns a Number.

4.4.2 write

`write` either writes a String to an IO object, or writes a single byte. The following example writes the string "test" to an IO object called io: `write io "test"` The following example writes the newline character to io: `write io 10`

4.4.3 connect

`connect` establishes a TCP connection to the given ip address and returns an IO object. The following example makes a GET request to a webserver running on localhost and the returns nil:

```
let io = connect "localhost:80";
let result = write io "GET /";
let result = write io 10;
nil
```

5 The Standard Library

5.1 Introduction

The standard library is in a very early state. You can use modules from the standard library with the `use` command, which will look something like `use "std::list";` To add your own modules you have to create a file with your library functions, then add the file into the `libs` vector in `src/libs.rs`.

5.2 List

The List module of the standard library contains some extra list processing functions. It is available under "std::list".

5.2.1 let list1 = \a . ...

`list1` lets you create a single element list. This is the same as doing `cons a nil`.

5.2.2 let list2 = \a b . ...

`list2` does the same thing as `list1`, but it creates a list with two elements instead of one.

5.2.3 let list3 = \a b c . ...

`list3` does the same thing as `list1`, but it creates a list with three elements instead of one.

5.2.4 let list4 = \a b c d . ...

`list4` does the same thing as `list1`, but it creates a list with four elements instead of one.

5.2.5 let map = \c list . ...

The map function takes a closure, c, and a list. `map` applies the closure to every element in the list. `map \x . ( * x 2 ) ( list2 4 8 )` produces the list `cons 8 ( cons 16 nil )`.

5.2.6 let fold = \c list . ...

The map function takes a closure, c, and a list. `fold` applies the closure to two elements at a time, starting from the tail of the list. The following example sums up all the elements of a list: `fold + ( list2 4 8 )` .