lispy_docs.md

Lispy Documentation

Introduction

This webpage contains the documentation of my lisp implementation which was invented by Daniel Holden. The documentation is divided into sections and the index is available after this introduction. This implementation is very different from all the typical lisps and there are some noticable differences. This guide assumes that you atleast know what Lisps are. However if you don't, I recommend searching about it. It is not completely necessary to understand lisps for you to use Lispy but a little knowledge won't do you any harm.

Also please note that I am in no way a Lisp expert and apologise to all of the Lisp evangelists who found mistakes due at my ignorance or lack of knowledge.

Lispy is a dynamically typed interpreted language implemented in C. It is just meant for fun/tinkering language rather than being a "work" or "production" oriented language. This implementation lacks many of the useful features that make up a Programming Language. The focus here, is on the documentation so the deficiencies will be discussed in a later section.

Documentation Index

Getting started

Modes

Lispy can be in two modes -- shell and scripting. Shell mode is one where you all code is executed per line like the Python interpreter. Scripting mode is where you execute `lispy` binary with a file having extension `.bsf` . The Following section will give some examples:

Hello World - Scripting

A standard hello world program looks like this in lispy.

```
(println "Hello, World!")
```

Then we store above text in a file called `hello.bsf` and then we run it using the following command.

```
$ lispy hello.bsf
```

and the output should give us

```
$ "Hello, World!"
```

Hello World - Shell

In the shell mode first we execute lispy

```
$ lispy
Lispy version 1.0.2
Press CTRL + C to exit
>>>
```

This is the output you should get. After this we're going to print hello world using `println` builtin

```
>>> (println "Hello, World!")
"Hello, World!"
()
```

Note that in shell mode, `()` indicates success.

Another thing to be noted is that `(println "Hello, World!")` in shell mode, is the same as, `println "Hello, World!"` . The parentheses or brackets can be omitted in shell mode. However this is note true for scripting mode and you must include the parentheses `()` in your scripts.

Comments

Lines starting with `semi-colon ( ; )` are comments in the lisp. These lines are ignored by the interpreter

For example:

```
; This will print "X doesnt equal to y"
(def {x y} 24 23)
(if (== x y)
  {println "X equals to y"}
{println "X doesnt equal to y"})
```

In the above example, the line after the semi-colon is ignored and does not affect the code at all and is meant to provide readability only.

Operators

The operators in lispy follow polish notation i.e they come before their operands. For example, To add two numbers we must do

```
>>> (add 2 2)
4.00000
```

or

```
>>> (+ 2 2)
4.00000
```

Mathematical Operators

There are 8 arithmetic operators and the following table shows the comprehensive list of mathematical operators in lispy.

Operator	Name	Alt name	Usage	Return Value Type
Addition	add	+	(add 2 2)	S-expression
Subtraction	sub	-	(sub 6 2)	S-expression
Multiplication	mul	*	(mul 23 9)	S-expression
Division	div	/	(/ 26 2)	S-expression
Remainder / Modulus	rem	%	(% 26 2)	S-expression
Power	pow	^	(^ 4 5)	S-expression

Please note that `Name` and `Alt name` of operators can be used interchangeably.

Relational Operators

There are 8 relational operators and the following table shows the comprehensive list Relation operators in lispy.

Operator	Name	Alt name	Usage	Return Value Type	Return values
Greater than	>	NA	(> 10 2)	S-expression	1.00000 if true 0.00000 if false
Lesser than	<	NA	(< 4 7)	S-expression	1.00000 if true 0.00000 if false
Greater than or equals to	>=	NA	(>= 23 9)	S-expression	1.00000 if equal or greater 0.00000 if false
Lesser than or equals to	<=	NA	(<= 24 56)	S-expression	1.00000 if equal or lesser 0.00000 if false
Comparison (Equals to)	==	NA	(== 24 24)	S-expression	1.00000 if true 0.00000 if false
Comparison (does not equals to)	!=	NA	(!= 24 23)	S-expression	1.00000 if true 0.00000 if false
Minimum	min	NA	(min 4 2 8 21 45)	S-expression	Minimum number in input
Maximum	max	NA	(max 4 5 6 23 65)	S-expression	Maximum number in input

Logical Operators

Please note that these operators are not defined in the core lispy language, These are a part of the Lispy Standard Library. I won't be going into much detail about the library.

Operator	Usage	Description
and	(and 0 1) or (and 1 0) or (and 1 1)	Logical AND
or	(or 0 1) or (or 1 0) or (or 1 1)	Logical OR
not	(not 2)	Logical not

Variables

Variables are quite interestingly implemented in this lisp version. The variables and functions are defined in two environments -- Parent and Child. The Parent environment contains all the builtin functions and variables etc. The lisp first checks if the user input within Parent environment and if it is there, it calls that functions or displays that variable. However if it is not, Then it goes to child environment and again checks if there is a user defined function or variables there and then calls/displays it. Again if it is not there, It then defines the variable with the input from user and if some variable was already present, Its value is replaced with the newer one. There is another interesting feat. due to this Parent-Child structure that we will talk about sooner in this section.

Now, onto user defined variables. Variables can be defined using the `def` keyword or builtin.

Function	Usage	Description	Return value
def	(def {num} 25)	Defines a userspace variable Takes a quote-expression first argument then a s-expr, q-expr, or a string	() on success

Some examples:

```
; A number as a variable
>>> (def {num} 25)
()
>>> ;Printing the variable now
>>> num
25.00000

; A string as a variable
>>> (def {name_string} "My Name")
()
>>> name_string
"My Name"

; A Q-expression / List as a variable
>>> (def {list} {2 4 5 6 7})
()
>>> list
{2 4 5 6 7}

; Assigning a variable to a variable
>>> (def {num} name_string)
()
>>> num
"My Name"
```

Function Aliasing

The `def` keyword can also be used to rename builtin functions as something else. This is due to the Parent-Child environment structure that was described earlier.

For example, We can alias `mul` as `add-n-times`

```
>>> (def {add-n-times} mul)
()
>>> add-n-times 2 2
4.00000
>>> add-n-times 24 3
72.00000
```

As you can see, function aliasing is quite interesting and can be used to name large sets of instructions into a more compact and readable form. This is the end of this section, Moving on...

The above examples should make it pretty clear on how to use variables. Before we jump into functions, I would here like to clear up the difference between a S-expr and Q-expr before we dive into the depth of functions (Well they're not that deep....)

An Important Difference

If you've been following along the documentation until now, and you're unfamiliar with lisps. You must be wondering what exactly is the difference between **S-expr and Q-expr**. If you were smart enough and have read some examples, you must've already figured something out and that is a Q-expr is just a list. And if you did, then congratulations! You are correct. Lisps internally store their source code and data as lists and that is one of the reasons why they're so powerful, their ability to mix source code with data is amazing, However what I said isn't exactly the way things work.

S-Expr

The syntax is always enclosed in parentheses `()` as you may have observed. S-expressions are just a nested list of data and that data may be syntax, variables, input, q-expr, or nested s-expr themselves. The atoms and cons cells aren't implemented in a typical way in Lispy as they are in a "typical lisp". It is a huge tree with a lot of branches and then they have nodes and then nodes themselves have branches. In context to lispy, you may want to read Chapter-7 of BuildYourOwnLisp where Daniel explains the inner structure of the tree/s-expressions.

Q-Expr

Q-expressions, enclosed in curly brackets are a form of data type, a list. The list itself may contain nested lists, strings, numbers. Another thing to note here is that instead of using `commas -- ,` to separate values, we use spaces, this helps in saving a lot of character space internally in the buffer and the result is almost identical with the same readability as commas.

Functions

There are many useful builtin functions in lispy. There are some functions described in the Standard Library of Lispy as well but we will not be taking a look at the standard library for quite a while. This time we will be describing functions one-by-one instead of making a small table. Earlier operators need not much explaining as they quite speak for themselves but the same is sometimes not true for the functions defined in the lisp.

List Functions

List

Function	Usage	Description	Return Type
list	(list 14 23 12 56 "hello_world")	Convert data to Q-expr or list	Q-expr

Head

Function	Usage	Description	Return Type
head	head {14 23 12 56}	Get the first element of a Q-expr	Q-expr

Tail

Function	Usage	Description	Return Type
tail	tail {14 23 12 56}	Deletes the first element from Q-expr and returns the remaining data as it is.	Q-expr

Join

Function	Usage	Description	Return Type
join	join {14 23 12 56} {43 12 67}	Join two Q-expr and return them combined.	Q-expr

Eval

Function	Usage	Description	Return Type
eval	eval {println "Hello, World!"}	Take a Q-expr and evaluate it as a S-expr.	S-expr

The above function might be hard to understand so let me interject for a moment and explain how it works, as the description says, it evaluates lists or q-expressions. Normally, If `{println "hello, World!"}` were a normal q-expr i.e without `eval`. It would have just been a normal q-expr of length 2 and its content would remain untouched unless explicitly done with List functions. Its contents would not have been evaluated.

With the `eval` function, it gets evaluated. Another example

```
>>> eval {tail {23 25 26}}
{25 26}
```

Cons

Function	Usage	Description	Return Type
cons	cons 21 {22 23 24 25}	Takes a value and a q-expr and appends the value to front.	Q-expr

Len

Function	Usage	Description	Return Type
len	len {12 12 32 215 76 45}	Takes a Q-expr as argument and returns the length of expression	Integer

Init

Function	Usage	Description	Return Type
init	init {12 12 32 215 76 45}	Takes a Q-Expr and removes the last element from it	Q-expr

Above all concludes the builtin list functions of Lispy.

User Defined Functions

We can define our own functions using the `fun` function. It is defined in the standard library. There is also another way of defining user-functions using the `\` or `lambda` function that is built into the lisp. We won't be going into much detail about lambda functions here but I will explain it briefly and show an example then move to the more easier and readable form of defining functions using `fun` .

Lambda

Function	Usage	Description	Return Type
\	(\ {x y} {* x y}	Lambda function takes two q-expr as arguments, the first is the formal arguments the 2nd is the function body.	lambda function

As you can see, to define a function that takes two arguments and multiplies them, we use the above code. Then to call it we have to reuse the above code:

```
>>> (\ {x y} {* x y}) 10 20
200.00000
```

As you can see readability is quite hard and there is a lot of redundant code. Then we can name it using the `def` keyword described earlier.

```
>>> def {mul} (\ {x y} {+ x y})
```

Then we can just call it by its name along with its arguments

```
>>> mul 5 2
```

This method defining functions is quite tedious and repetitive and hard to read. Therefore the standard library has a function `fun` which allows the user to define other functions.

Fun

Function	Usage	Description	Return Type

Function	Usage	Description	Return Type
fun	(fun {mul x y} {* x y}	Defines a function. Takes two q-expr arguments. First one is function name and formal arguments. First argument must have function name as the first member of q-expr then the variables. The 2nd argument is function body.	lambda function

The function takes two q-expr arguments same as lambda and then defines a function in child environment.

Using fun requires you to `load` the standard library which is present in `/usr/bin/stdlib.bsf` . It can be loaded with `load "/usr/bin/stdlib.bsf"` on top of the file. `load` function will be described in the later sections.

The fun function is implemented as following:

```
>>> fun
>>> (\ {f b} {def (head f) (\ (tail f) b)})
```

Conditionals

If

if is a conditional statement, part of Lispy. It is described briefly:

Function	Usage	Description	Return Type
if	(if (== x y) {println "X equals to y} {println "X doesnt equal to y}	Takes 2 arguments and 1 optional argument. First argument is a S-Expr and is the condition for if statement Second argument is a Q-Expr which are the statements to be executed. Third argument is optional and the else part of the if statement. It is also a Q-Expr.	

The else statement is implicit and is not explicitly defined in the language. It is always the statement that follows the first `{}` Q-expr.

Example: Store the following in a `test.bsf` file

```
; This will print "X equals to y"
(def {x y} 23 23)
(if (== x y)
  {println "X equals to y"}
{println "X doesnt equal to y"})
```

Then we run it with

```
$ lispy test.bsf
"X equals to y"
```

Else Example:

Store the following in a `test.bsf` file

```
; This will print "X doesnt equal to y"
(def {x y} 24 23)
(if (== x y)
  {println "X equals to y"}
{println "X doesnt equal to y"})
```

Then we run it with

```
$ lispy test.bsf
"X doesn't equal to y"
```

Output & File loading in shell mode and scripts

Lispy has two builtin functions defined in it which allow to send output to the user. The Functions are `println` and `error`. As from the names it is quite clear what their purpose is. Both functions are described briefly here.

Println

This function prints each argument separated by a space and then prints a newline character to finish. It returns the empty expression.

```
(def {x y} 23 23)
(if (== x y)
  {println "X equals to y"}
```

If we run above code using `lispy`, The line X equals to y will be printed.

Printing variables

We can print variables using the `println` statement. We can simply just give it a name of the variable and it will print it.

```
>>> def {num1} 2
()
>>> println num1
2.00000

; Variable operations or one liners
>>> println (/ num1 2)
1.00000
()
```

Error

This builtin is meant to return errors to the user in case anything goes wrong during the execution. Its syntax is very similar to `println`. It takes a string as argument and prints it if the control flow reaches the error statement.

```
(def {x y} 3 2)
(if (== (% x y) 0)
  {println "x / y is true"}
{error "X is not divisible by Y"})
```

Loading

Exisitng Lispy code can be loaded into the present running environment using `load` function. Here is a brief table:

Function	Usage	Description	Return Type
load	load "/usr/bin/stdlib.bsf"	Takes a single string argument which should be a file or a file path. Then the code is loaded into the present running environment.	

Loading the standard library into the current environment:

```
; Script mode
(load "/usr/bin/stdlib.bsf")

; Freely use stdlib functions now.
```

- Shell mode

```
>>> load "/usr/bin/stdlib.bsf"
```

```
; freely use stdlib in shell mode
```

Standard Library

The Lispy standard library is present in `/usr/bin` directory. It is named `stdlib.bsf` and it contains various definitions of useful functions. These functions are not a part of core `lispy` and lispy can very well function without these.

Function	Usage	Description
and	(and 0 1) or (and 1 0) or (and 1 1)	Logical AND
or	(or 0 1) or (or 1 0) or (or 1 1)	Logical OR
not	(not 2)	Logical not
nil	(println nil)	Empty {}
false	(println false)	Defined as 0
true	(println true)	Defined as 1
fun	(fun {mul x y} {* x y}	Defines functions
unpack	(unpack + {1 2 3 4})	Appends operator to Q-expr then evaluates the Q-expr
pack	(pack tail 1 2 3 4)	Takes a function and some arguments. Converts arguments to Q-expr and applies function.
curry	(curry + {1 2 3 4})	Same as unpack, different name
uncurry	(pack tail 1 2 3 4)	Same as pack, different name
flip	(flip div 12 2)	Takes one function as argument and then two arguments on which the function will perform However, before the function runs, the input order is switched.
ghost	(ghost + 2 2)	Does absolutely nothing. Stays in front of expressions and like a ghost. Not interacting with anything.
comp	(comp tail head {1 2 3 4}	Takes two functions as arguments and then another argument to the last function. The last argument is passed to 2nd functino and then the resulting argument is passed onto the function
fst	(fst {1 2 3 4})	Actually extracts the first element of the list. In form of a usable value S-expr instead Q-expr
snd	(snd {1 2 3 4})	Same as above except 2nd element
trd	(trd {1 2 3 4})	Same as above except 3rd element
nth	(nth 3 {1 2 3 4})	Takes two arguments. First is the position and 2nd is the list. Extracts the position element from list
last	(last {1 2 3 4})	Returns last element of list
take	(take 3 {1 2 3 4})	Takes two arguments. First is position and 2nd is the list. Removes the position element.
drop	(drop 3 {1 2 3 4 5})	Takes two arguments. First is position and 2nd is the list. Drops elements until position.
split	(split 3 {1 2 3 4 5 6 7}	Takes two arguments. First is position and 2nd is the list. Splits the list at position.
elem	(elem 2 {2 4 5 6}	Takes two arguments. First is a value and 2nd is list. Compares the value with all elements of list.

Function	Usage	Description
		If found, returns 1 else, return 0
map	(map println {"hello" "World"})	Takes two arguments. First is a function and 2nd may be any expression. Then it maps the function to each element of expression.
filter	(filter (\ {x} {> x 2}) {5 2 11 3 -4 1})	Takes two arguments. First is a functional condition. A functional condition is a condition in form of a function. Second is a list. It takes the condition and compares with each element of function. And then return a Q-expr where condition was true.
foldl	(fun {sum l} {foldl + 0 l})	Supplied with a function f, a base value z and a list l Each element is of list l is merged with base value z. The merging is dependent upon the function specified. When the merging is done, The elements are merged with themselves. Note that merging means applying the supplied function.
select	select {(== i 0) "st"} {(== i 1) "nd"} {(== i 3) "rd"} {otherwise "th"}	takes in zero or more two-element lists as input. For each two element list in the arguments it first evaluates the first element of the pair. If this is true then it evaluates and returns the second item, otherwise it performs the same thing again on the rest of the list.
case	(case x {0 "Monday"} {1 "Tuesday"} {2 "Wednesday"} {3 "Thursday"})	Similar to C switch statements. Takes a constant value and variable q-exprs. Compares and returns values.
Fibonacci	(fib n)	Prints n number of fibonacci Sequence.

Deficiencies as a Programming Language

- Lack of types or Type-lessness
- Currently lispy wraps only native C `double` and `char*` types for computation.
- No Loops
- Lack of user defined types
- Difference of syntax and other implementations from standard/classic lisps
- Lack of operating system interaction
- Garbage Collection
- Pool Allocation (much use of malloc)
- Variable hastable (linear search of variables in environment is currently in use)
- Static Typing
- Lexical Scoping

Conclusion

Well this concludes our guide like documentation. I have tried to keep everything simple and clear. If you don't understand any part of it be sure to contact me. This took me complete **7+ hours** of constant screen time to write all the documentation. I don't know if anyone will read this part or it'll just remain on some dark corner of the internet. Anyways, I had much fun writing this documentation. I hope you learnt something from it or just wanted to use lispy. This was a long and tiring task. After this I'm gonna write some makefiles for lispy and push the changes and hope everything works out.

References

- BuildYourOwnLisp
- Lisp Wikipedia
- My old readme