

# Eval and Macros

# Disclaimer

We will not be talking about *programming* in Lisp yet.

That will come very soon.

We will talk about the life cycle of Lisp code and the interesting possibilities that are unique to homoiconic languages.

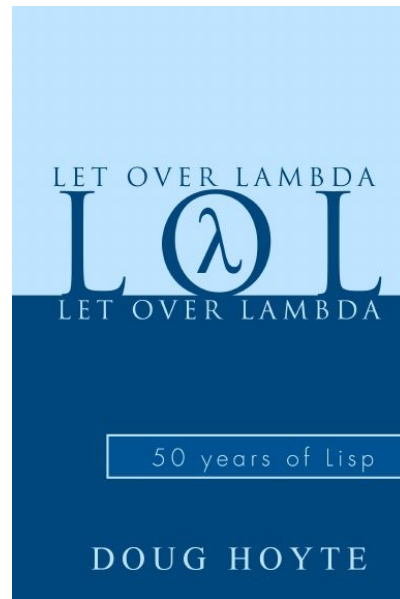
Macros

# Why talk about Lisp code lifecycle before Lisp?

Macros are the single greatest advantage that lisp has as a programming language and the single greatest advantage of any programming language. With them you can do things that you simply cannot do in other languages. Because macros can be used to transform lisp into other programming languages and back, programmers who gain experience with them discover that all other languages are just skins on top of lisp. This is the **big deal**.

*Let Over Lambda*

Doug Hoyte



*"Let Over Lambda is one of the most hardcore computer programming books out there."*

# Why talk about Lisp?

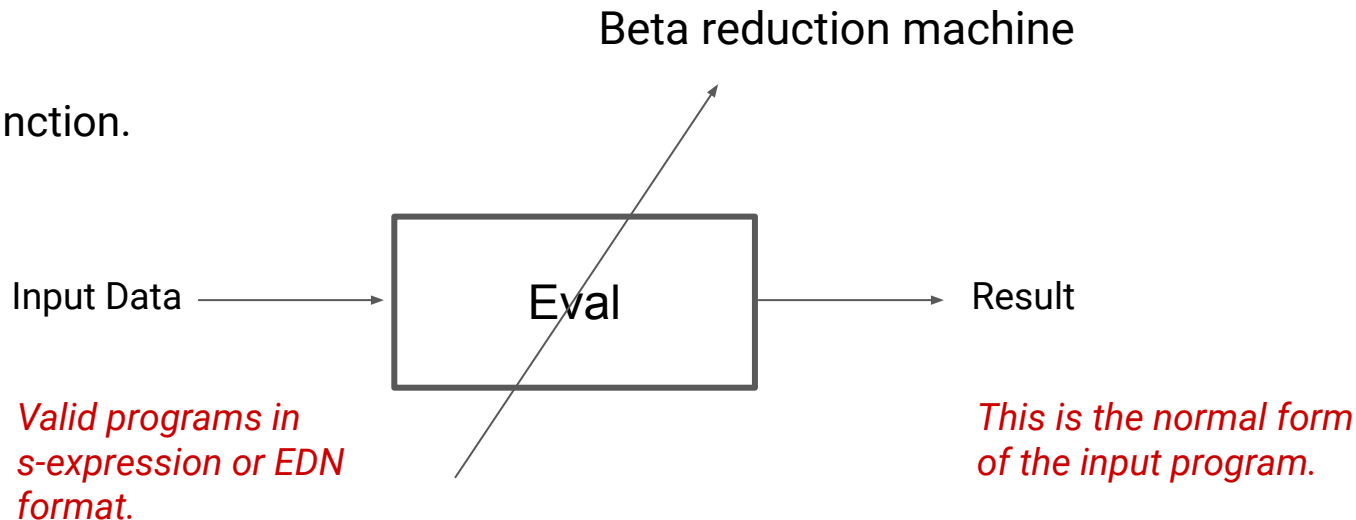
Lisp is worth learning for the profound enlightenment experience you will have when you finally get it; that experience will make you a better programmer for the rest of your days, even if you never actually use Lisp itself a lot.

*How to Become a Hacker*

Eric Raymond

# Eval of Lisp Programs

Eval is a function.



# Two forms: function application vs lists

A function application:

`(<func> <parameters ...>)`

A list:

`'(<element> <element>)`

or

`(quote (<element> <element> ...))`

# Example

$(+ 1 2 3)$

*This is a function application. It can be reduced by beta-reduction to 6.*

$'(+ 1 2 3)$

*This is a list with **four** elements.*

- 1. The built-in function  $+$*
- 2. The number 1*
- 3. The number 2*
- 4. The number 3*

*This is already in **normal form**, so it cannot be reduced.*

# Example

We will cover the syntax in detail. For now, let's just try to understand the input code with intuition.

```
(sum
  (for [i (range 10)]
    (* i i)))
```



?



# Example

We will cover the syntax in detail. For now, let's just try to understand the input code with intuition.

```
(sum  
  (for [i (range 10)]  
    (* i i)))
```

```
(sum . . .)
```

*This is a function application.  
We will need to evaluate the  
parameter, and then use  
beta-reduction.*

# Example


The Eval function will recursively evaluate sub-expressions.

```
(sum
  (for [i (range 10)]
    (* i i)))
```

# Example

The Eval function will recursively evaluate sub-expressions.

```
(sum
  (for [i (range 10)]
    (* i i)))
```



[0 1 2 3 4 5 6 7 8 9]

# Example

The Eval function will recursively evaluate sub-expressions.

```
(sum  
  (for [i (range 10)]  
    (* i i)))
```



```
'(0 1 4 16 25 36 49 64 81)
```

# Example

Finally we get to the top level of the program, and produce the normal form of the entire input expression.

```
(sum  
  (for [i (range 10)]  
    (* i i)))
```



```
(sum '(0 1 4 9 16 ...))
```



285

*This is the normal  
form of returned  
by Eval.*

# Lisp programs from 10,000 feet view

Any Lisp program can be a single expression, which is a composition of sub-expressions.

*But for practical reasons of readability and modularity, Lisp programs come as a collection of expressions that are composed together.*

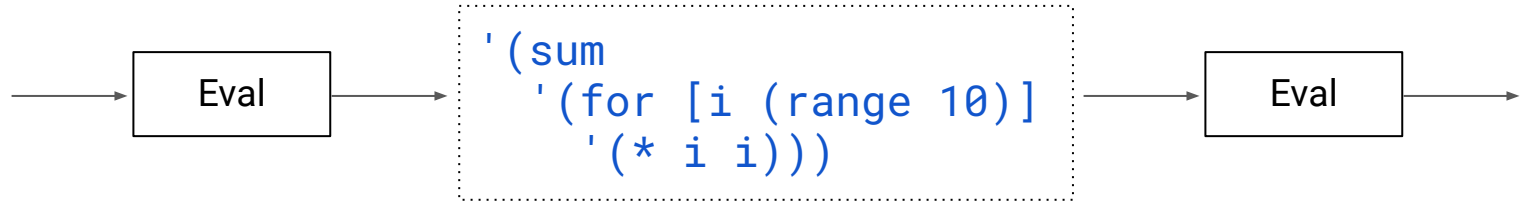
The **eval** function will take the expressions as data, and compute by performing reduction.

*Interactive programming since 1965.*

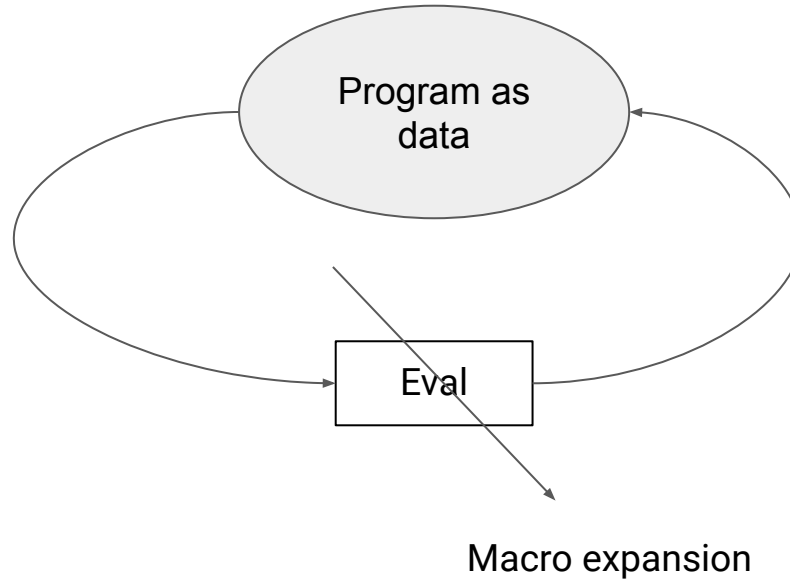
*This is very different from compiled languages.*

# Code generation

What if **eval** produces code?



# Self-modifying code






# Macro Programming

**Lisp code** that contains **invalid program segments**.

```
(simulate-python  
  (let total = 0)  
  (for i in range(10):  
    (total += i))  
  (println(total)))
```



```
graph LR; A["(simulate-python (let total = 0) (for i in range(10): (total += i)) (println(total)))"] --> B["Macro expansion"]; B --> C["(let [total (sum (range 10))] (println total))"]
```

Macro  
expansion

**100% Lisp code:**

```
(let [total (sum (range 10))]  
  (println total))
```

# Lisp is powerful

Lisp has been growing for over 50 years, and it's only gaining popularity.

- The smallest language possible with minimal syntax
- Macros allow the syntax to grow with the application domain
- Functional programming constructs that fit the needs of today's challenges