Lisp Macros

Aidan Hall

23rd November 2023

Lisp

```
(define (fibonacci n)
  (if (< n 2)
      1
      (+ (fibonacci (- n 1))
         (fibonacci (- n 2)))))
(fibonacci 5)
```

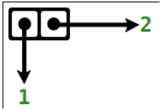
```
8
```

Pairs

- Pairs in Lisp are called `cons` cells.

  -
    ```
    (cons 1 2)
    ```

```
(1 . 2)
```



- The first and second components are called the `car` and `cdr`.

```
(car (cons 1 2))
```

```
1
```

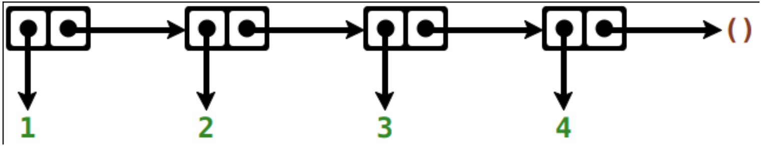
```
(cdr (cons 1 2))
```

```
2
```

List Structure

- Linked lists are built from `cons` cells (pairs), and are terminated by `()`, the empty list[1].

```
(list 1 2 3 4)
```



---

[1] `list` is a function that makes a list of its arguments.

Quotation

- When quoted with ’, an expression is not evaluated.

-
  ```
  (let ((x 5)) x)
  ```

  ```
  5
  ```

-
  ```
  (let ((x 5)) 'x)
  ```

  ```
  x
  ```

Quasiquotation

- It is sometimes convenient to evaluate parts of a quoted expression. For this, we use the quasiquote (‘) and unquote (,) operators.

```
`(1 2 ,(+ 3 4))
```

```
  (1 2 7)
```

- If an expression evaluates to a list, the result can be spliced into the surrounding quoted form with the `,@` operator.

```
`(1 2 ,(list 3 4))
```

```
  (1 2 (3 4))
```

```
`(1 2 ,@(list 3 4))
```

```
  (1 2 3 4)
```

Code as Data

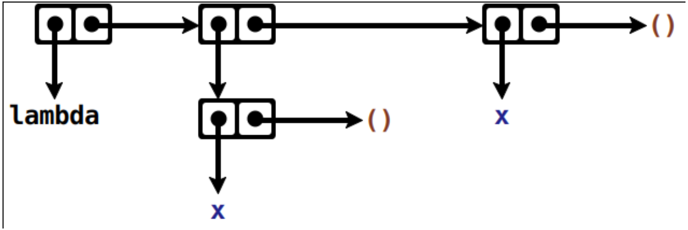
- Lisp code can be represented as Lisp data.

  -
    ```
    (lambda (x) x)
    ```

    ```
    #<procedure 654a139a36e8 at <unknown port>:8:1 (x)>
    ```

  -
    ```
    '(lambda (x) x)
    ```

    ```
    (lambda (x) x)
    ```



Macros

- Macros are Lisp functions that run at compile time and produce Lisp code.

-
  ```
  (defmacro plus1 (x)
    `(+ 1 ,x))
  (macroexpand-1 '(plus1 5))
  ```

  ```
     (+ 1 5)
  ```
- Their arguments are not evaluated before being passed to them.
  - This means the arguments don't need to be valid Lisp code.
- Since you have access to the whole language in macros, they can perform arbitrary transformations on the input.

Implementing `let` as a Macro

We briefly saw that a let expression is equivalent to a λ application:

```
(let ((a 2) (b 3))
  (+ a a b))
```

is equivalent to:

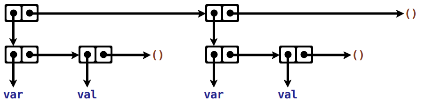
```
((lambda (a b)
   (+ a a b))
 2 3)
```

We can use a macro to transform the first expression into the second!

Splitting up the Variable Bindings

- In a `let` expression, the variable bindings are in the following form:

```
'((var val) (var val))
```



- We can get the variable names by taking the car of each element:

```
(map car '((var val) (var val)))
```

```
(var var)
```

- We can get the values with cadr: `(cadr x)` = `(car (cdr x))`.

```
(map cadr '((var val) (var val)))
```

```
(val val)
```

`mylet`

Now we can write the macro:

```
(defmacro mylet (binds . body)
  `((lambda ,(map car binds)
      ,@body)
    ,@(map cadr binds)))

(mylet ((a 2) (b 3))
  (display a) (newline)
  (display b) (newline)
  (display (+ a a b)) (newline))
```

```
2
3
7
```

Why should you care about macros?

- Macros allow you to extend the syntax of a programming language.
- Rather than waiting for the language creators to implement a feature, you can do it yourself.
- Modern languages are gradually catching up to Lisp:
  - C++ templates and `constexpr` functions.
  - Rust macros.

Thank You

Any questions?

Common Lisp version of `mylet`

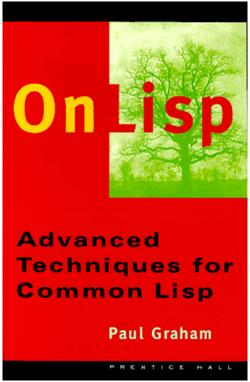
Scheme:

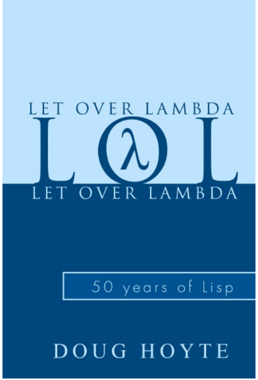
```
(defmacro mylet (binds . body)
  `((lambda ,(map car binds)
       ,@body)
     ,@(map cadr binds)))
```

Common Lisp:

```
(defmacro mylet (binds &rest body)
  `(funcall
    (lambda ,(mapcar #'car binds)
      ,@body)
    ,@(mapcar #'cadr binds)))
```

References & Notes





- On Lisp by Paul Graham (building languages on Lisp).
- Let Over Lambda by Doug Hoyte (building languages on On Lisp).
- My essay will be on Common Lisp, but I am using Scheme for this presentation.