Common Lisp and Introduction to Functional Programming
Lecture 2: Common Lisp Basics

Yuri Zhykin

Feb 11, 2021

λ-calculus 1/2

- Introduced by Alonzo Church in 1930s.
- λ-calculus is a formal system in mathematical logic for expressing computation based on function abstraction and application using **variable binding** and **substitution**.
- Church showed that all *computable* functions can be expressed in λ-calculus.
- Church and Turing showed that *Turing machine model* and *λ-calculus* are equivalent.

λ-calculus 2/2

- Primitives:
  - x - **variable** - a symbol representing a parameter or value.
  - $(\lambda x.M)$ - **abstraction** - function definition; M is a λ-*term*.
  - $(M\ N)$ - **application** - applying a function to an argument; both M and N are λ-*terms*.
- Operations:
  - $(\lambda x.M[x]) \rightarrow (\lambda y.M[y])$ - α-**reduction** - renaming a variable to avoid name collisions.
  - $((\lambda x.M)E) \rightarrow (M[x := E])$ - β-**reduction** - replacing variable with the argument expression (calculation step).
  - $f \leftrightarrow \lambda x.(f\ x)$ - η-**reduction** - drop an *abstraction* for simplicity.
- Example: $((\lambda x\ .\ 2 \cdot x)\ 12) \rightarrow (2 \cdot 12) \rightarrow 24$

λ-calculus and Lisp History

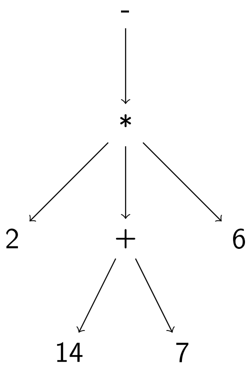
- John McCarthy - Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I, 1958-1960.
- Steve Russel implemented the evaluator for the LISP (LISt Processing) system in machine code for **IBM 704** machine.
- M-expressions ($f[g[A, B]]$) and S-expressions ($(f(gAB))$).
- First complete LISP compiler (Tim Hart, 1962) introduced incremental compilation.
- Dozens of dialects; modern ones include:
  - Common Lisp
  - Emacs Lisp
  - Clojure
  - Scheme (Racket)

Syntax 1/2

- Polish notation: operator (function name) prefixes the operands; eliminates operator precedence rules:
  - `2 * x + y`
  - `(+ (* 2 x) y)`
- Whole syntax consists of S-expressions and closely follows λ-calculus notation:
  - $((\lambda x \,.\, 2 \cdot x)\ 12)$
  - `((lambda (x) (* 2 x)) 12)`
- Lisp expressions can be of two types:
  - **atoms** evaluate to the values they denote,
  - **lists** evaluate as `(<operator> <operand1> <operand2> ...)`, except if they denote *special forms*.

Syntax 2/2

- Lisp expressions are trees of atoms `(- (* 2 (+ 14 7) 6))`



Lisp REPL 1/2

- **Read Eval Print Loop** - Lisp expression is read, evaluated, and the result is printed for the user to see.
- Literal evaluation:

```
CL-USER> "Hello, World!"
"Hello, World!"
CL-USER> 1000000
1000000
```

- Function evaluation:

```
CL-USER> (* 2 (+ 14 7))
42
CL-USER> (print "Hello, World!")
Hello, World!
"Hello, World!"
```

Lisp REPL 2/2

- Lisp has built-in variables to access evaluation history:

```
CL-USER> (+ 1 2)
3
CL-USER> (+ * 3)
6
CL-USER> (+ * 4)
10
CL-USER> (* ** ***) ;; (* 6 3)
18
```

Data Structures: Pairs

- `nil` - a constant value that represents nothing (Latin *nihil*).
- *Pair* - a combination of two values (a.k.a. *cons-cell*):

```
CL-USER> (cons 1 2)
(1 . 2)
CL-USER> (car (cons 1 2))
1
CL-USER> (cdr (cons 1 2))
2
```

- Pairs can contain any values:

```
CL-USER> (cons "John" 9000)
("John" . 9000)
CL-USER> (cons 1 nil)
(1)
```

Data Structures: Lists 1/2

- *Singly linked list*, or just *list*:

```
CL-USER> (cons 1 (cons 2 (cons 3 (cons "John" nil))))
(1 2 3 "John")
CL-USER> (list 1 2 3 "John")
(1 2 3 "John")
```

- Proper lists always have `nil` as `cdr` of the innermost `cons`:

```
CL-USER> (cons 1 (cons 2 (cons 3 "John")))
(1 2 3 . "John")
```

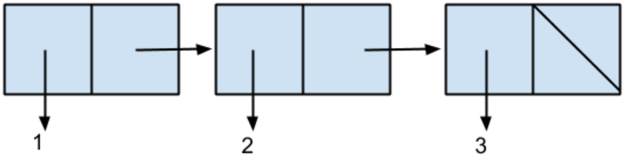
- `nil` also denotes an empty list:

```
CL-USER> (list)
nil
```

Data Structures: Lists 2/2

- **Head** and **tail** of a list can be accessed with the `car` and `cdr` function:

```
CL-USER> (car (list 1 2 3)) ;; (car (cons 1 (cons 2 ...)))
1
CL-USER> (cdr (list 1 2 3)) ;; (cdr (cons 1 (cons 2 ...)))
(2 3 "John")
```



Quoting

- Lisp expressions can be "quoted" - the evaluation rules will not by applied to them:

```
CL-USER> (quote (+ 1 2))
(+ 1 2)
CL-USER> '(+ 1 2)
(+ 1 2)
```

- List of *atoms* that represents a Lisp expression can be evaluated with the `eval` function:

```
CL-USER> (eval '(+ 1 2))
3
```

Symbols 1/2

- What exactly are atoms that are not literals?

```
CL-USER> (type-of (car '(cons 1 2)))
symbol
```

- Symbol is a core data structure of a Lisp system.
- Symbols represent names within a Lisp system and are grouped into *packages*.
- Symbol object contains 3 main fields:
  - `symbol-name`,
  - `symbol-value`,
  - `symbol-function`,

Symbols 2/2

- When non-atomic Lisp expression is evaluated, the function is retrieved from the `function` field of the symbol in the head of the list.
- Lisp languages that separate `symbol-value` from `symbol-function` are called Lisp-2 languages, as opposed to Lisp-1 languages that have a single namespaces for symbol values and symbol functions.
- In Lisp-2 languages, the function that will be called in each expression of the form

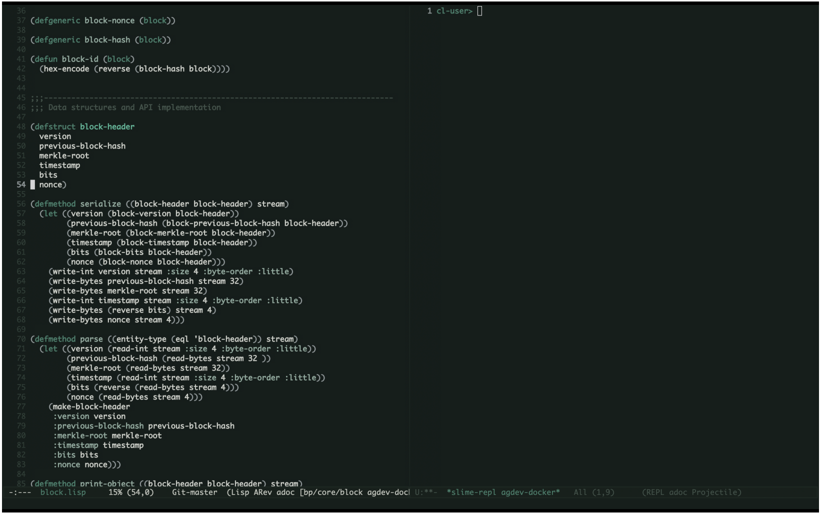
  ```
  (<function> <argument1> <argument2> ...)
  ```

  is known at compilation time, which allows compilers to generate very efficient machine code.

Lisp Development Environment

- **SBCL/Allegro CL** - Lisp implementation (compiler and interpreter).
- **Emacs** - extensible text editor written in C and Emacs Lisp; around since 1976; approximately 5000 packages available.
- **SLIME** - Emacs package that provides Lisp IDE functionality (REPL, cross-reference, selective compilation in source files, etc).
- **Paredit** - Emacs package that provides structural editing capabilities (operate on whole S-expressions instead of characters).

Emacs + SLIME



```lisp
(defgeneric block-nonce (block))

(defgeneric block-hash (block))

(defun block-id (block)
  (hex-encode (reverse (block-hash block))))


;;;-------------------------------------------------------------------------------
;;; Data structures and API implementation

(defstruct block-header
  version
  previous-block-hash
  merkle-root
  timestamp
  bits
  nonce)

(defmethod serialize ((block-header block-header) stream)
  (let ((version (block-version block-header))
        (previous-block-hash (block-previous-block-hash block-header))
        (merkle-root (block-merkle-root block-header))
        (timestamp (block-timestamp block-header))
        (bits (block-bits block-header))
        (nonce (block-nonce block-header)))
    (write-int version stream :size 4 :byte-order :little)
    (write-bytes previous-block-hash stream 32)
    (write-bytes merkle-root stream 32)
    (write-int timestamp stream :size 4 :byte-order :little)
    (write-bytes (reverse bits) stream 4)
    (write-bytes nonce stream 4)))

(defmethod parse ((entity-type (eql 'block-header)) stream)
  (let ((version (read-int stream :size 4 :byte-order :little))
        (previous-block-hash (read-bytes stream 32 ))
        (merkle-root (read-bytes stream 32))
        (timestamp (read-int stream :size 4 :byte-order :little))
        (bits (reverse (read-bytes stream 4)))
        (nonce (read-bytes stream 4)))
    (make-block-header
     :version version
     :previous-block-hash previous-block-hash
     :merkle-root merkle-root
     :timestamp timestamp
     :bits bits
     :nonce nonce)))

(defmethod print-object ((block-header block-header) stream)
```

```
1 cl-user>
```

```
-:---  block.lisp     15% (54,0)    Git-master  (Lisp ARev adoc [bp/core/block agdev-doc  U:**-  *slime-repl agdev-docker*   All (1,9)     (REPL adoc Projectile)
```

The End

Thank you!