SOFTWARE ARCHITECTURE
6. LISP

Tatsuya Hagino
hagino@sfc.keio.ac.jp

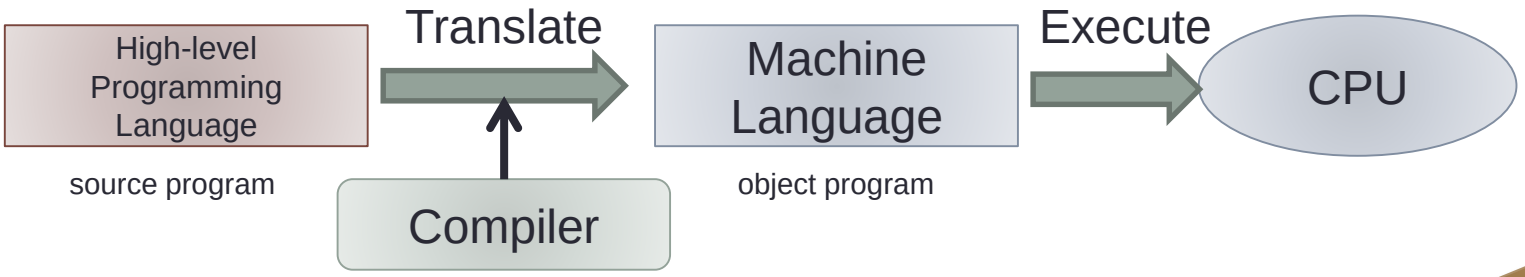
lecture URL

https://vu5.sfc.keio.ac.jp/slide/

Compiler vs Interpreter

- Compiler
  - Translate programs into machine languages
  - Compilers are used for the translation

Translate beforehand

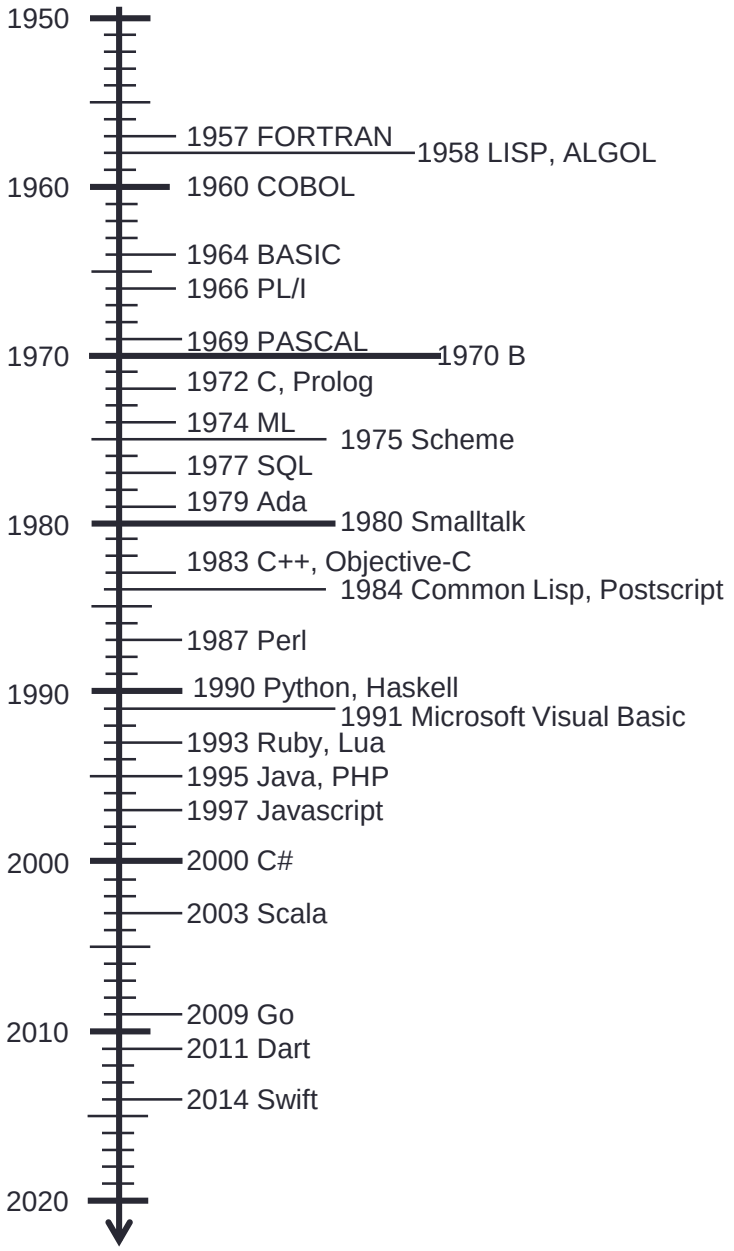


- Interpreter
  - Direct execution without translation
  - Slower execution compared with compiler
  - Easy and fast to execute when programs are changed

Interpret simultaneous

LISP

- LISP（LIST Processing）
  - Developed by John McCarthy
  - Second oldest programming language
    - The oldest one is FORTRAN
- Features
  - Based on lambda calculus
  - LISP is for symbolic processing
    - FORTRAN for numerical analysis
  - Often used in Artificial Intelligence field
- A lot of dialects
  - MACLisp
  - Common Lisp
  - Emacs Lisp
  - Scheme



Programming Languages

- Numerical analysis
  - FORTRAN
- Office processing
  - COBOL
- Operating system
  - C
- Symbolic manipulation and AI
  - LISP
- Web
  - JavaScript
  - PHP
- Script
  - Shell script
  - perl
  - ruby
  - python
- String manipulation
  - SNOBOL
- Simulation
  - SIMULA
- Embedding to devices
  - Java
- Education
  - PASCAL

Programming Language Paradigm

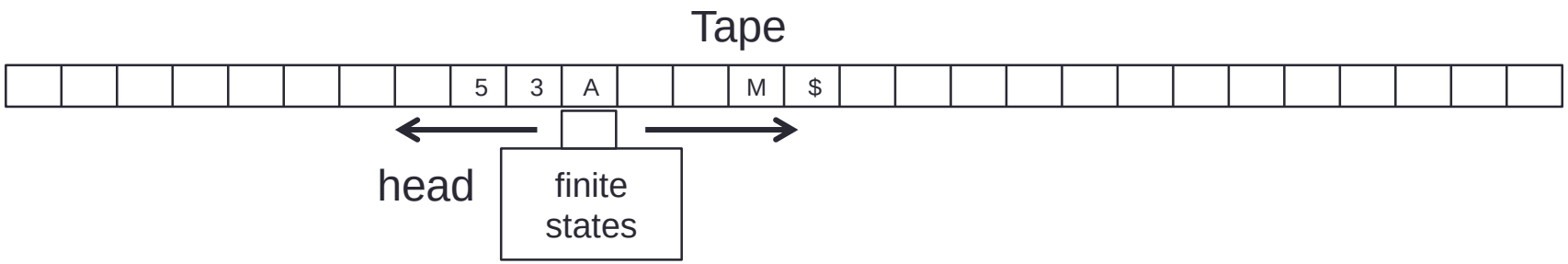
- Procedural
  - A program is a sequence of instructions.
  - The instructions are processed one by one in the order.
  - Most of the programming languages are procedural.
  - Fit with stored-program computer invented by Von Neumann.

- Functional
  - A program is a function.
  - Combine functions to solve problems.
  - The order of calculation does not matter.
  - Close to mathematical problem solving.

- Logical
  - A program is a list of logical rules.
  - The order of rules does not matter.
  - List conditions and restrictions.

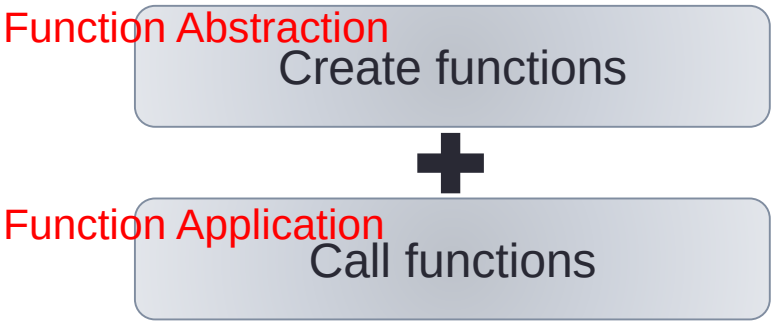
Model of Computer

- Turing Machine
  - Invented in 1936 by Alan Turing on his paper ``On computable numbers, with an application to the Entscheidungsproblem''
  - A Turing machine has a tape with infinite cells.
  - It has an head to read and write symbols in cells.
  - The head can read and write only one cell at a time.
  - The head can move left or write one cell.
  - The head has finite states.
  - With the current state and the current symbol in the cell, the head may change the symbol of the cell and moves left or right.



Other Models of Computer

- Recursive functions
  - Primitive recursive functions with μ operator
  - Primitive recursive functions
    - 0! = 1
    - (n+1)! = (n + 1) × n!
  - Not primitive recursive, but computable (needs μ operator)
    - Ack(0, n) = n + 1
    - Ack(m, 0) = Ack(m - 1, 1)
    - Ack(m, n) = Ack(m - 1, Ack(m, n - 1))
- Register machine
  - A simple machine with a finite number of registers (which can hold any number)
- Lambda calculus
  - Abstraction of functions
  - Function abstraction and application
  - αβ transformation as computation



Lambda Expression

- In Mathematics, functions are usually declared with names.
  - $f(x) = x + 5$
  - $f(3)$
- In lambda calculus, functions can be written without names.
  - $f = \lambda x . x + 5$
  - $f(3) = (\lambda x . x + 5)(3)$

```
<expr> ::= <var> |
           '(λ' <var> '.' <expr> ')' |
           '(' <expr> <expr> ')'
```

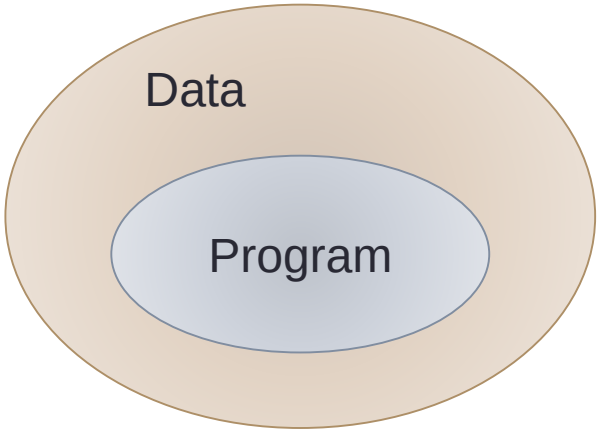
- Example
  - $(\lambda x. x)$
  - $((\lambda x. (x\ y))(\lambda z. z))$
  - $((\lambda x. (x\ x))(\lambda x. (x\ x))$

Lambda Calculus

- The computation rule is just function application（β transformation）.
  - $((\lambda x.M)N) \rightarrow M[N/x]$
- Example
  - $(\lambda x.x)y \rightarrow y$
  - $(\lambda x.xx)(\lambda y.y) \rightarrow (\lambda y.y)(\lambda y.y) \rightarrow \lambda y.y$
- Natural numbers are expresses as λ expressions.
  - $\mathbf{0} \equiv \lambda xy.y$
  - $\mathbf{1} \equiv \lambda xy.xy$
  - $\mathbf{2} \equiv \lambda xy.x(xy)$
  - $\mathbf{3} \equiv \lambda xy.x(x(xy))$
  - ….
- Arithmetic operations can be written as λ expressions.
  - $plus \equiv \lambda xyzw.(xz)((yz)w)$

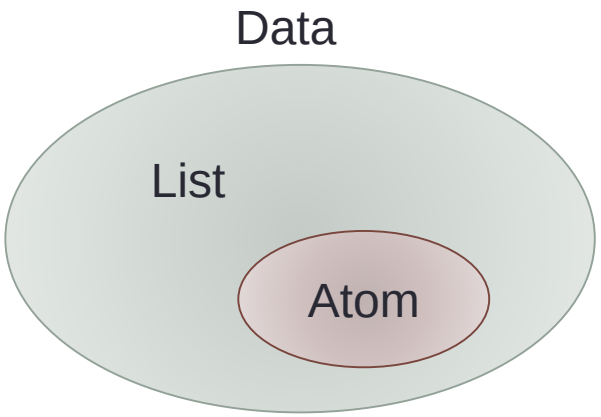
LISP Object

- There is no distinction between Data and Program.
  - A program is a data.
  - Von Neumann computer = stored-program computer



- Data
  - Atom
  - List

- Atom
  - numeric values: 0, 123, 3.14
  - strings: "abc", "Hello World!"
  - symbols: x, abc, hello

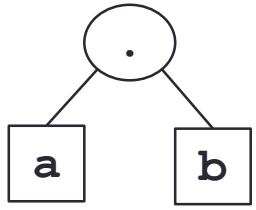


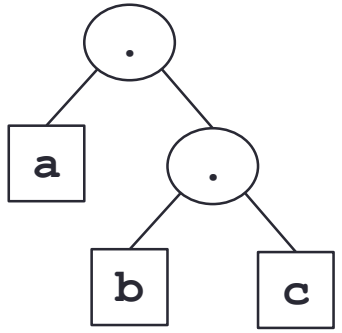
List

- List (or S expression)

```
<list> ::= '()' | <atom> | '(' <list> '.' <list> ')'
```

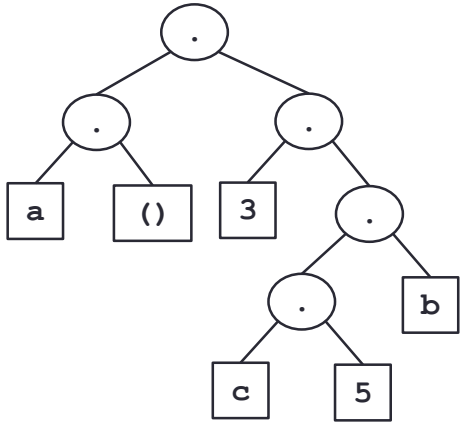
- `()` is the empty list
  - can be written as **`nil`**.



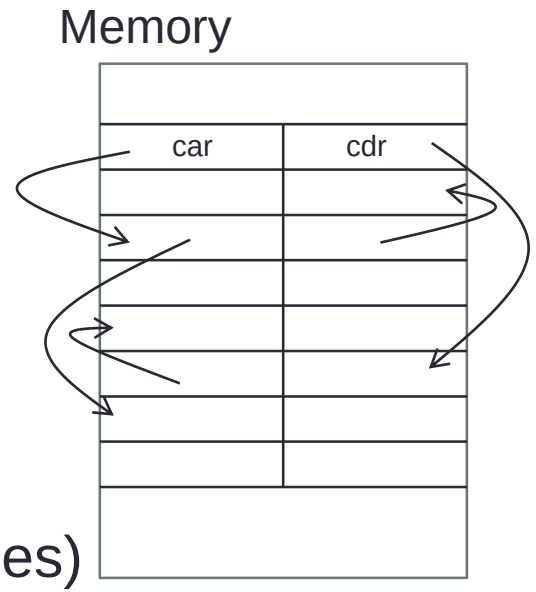


- A list is a binary tree with atoms as leaves.

- Example
  - **`(a . b)`**
  - **`(a . (b . c))`**
  - **`((a . ()) . (3 . ((c . 5) . b)))`**

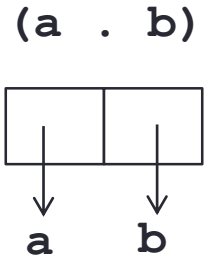


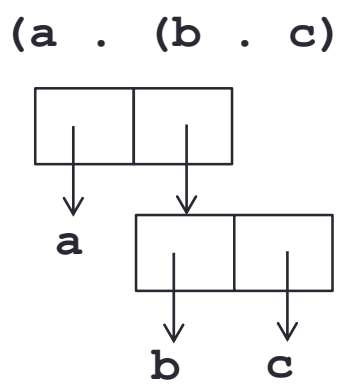
Implementation of List

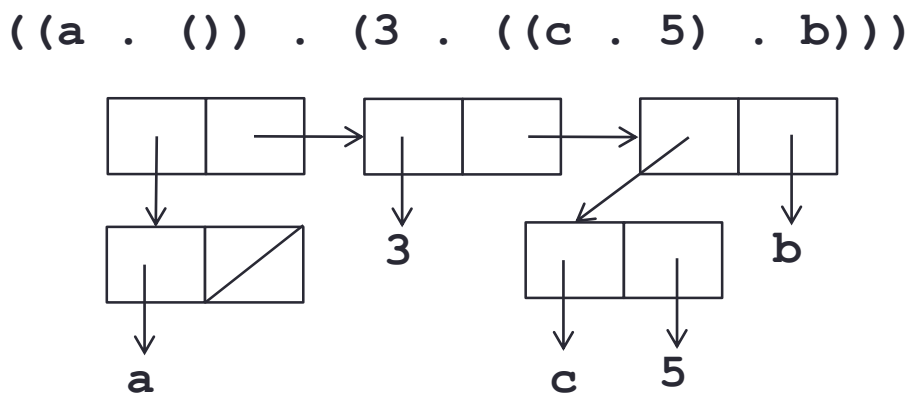


- Cons cell

- Each cons cell is a pair of pointers (i.e. addresses)
- car and cdr point to atoms or cons cells



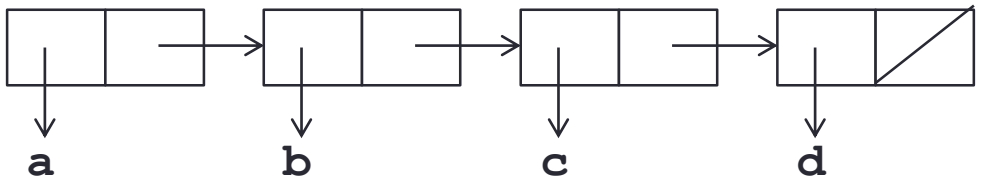




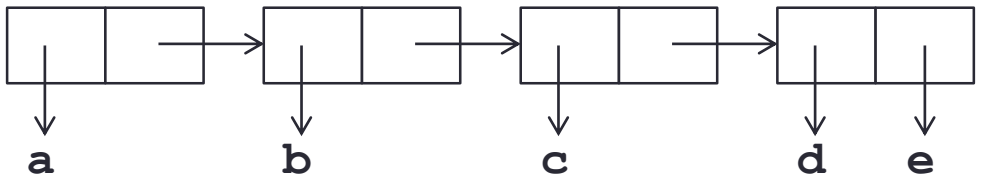
Notation of List

- **`(a b c d)`** is an abbreviation of the following S expression.
  - `(a . (b . (c . (d . ()))))`
- **`(a b c d . e )`** is an abbreviation of the following S expression.
  - `(a . (b . (c . (d . e))))`

`(a b c d)`



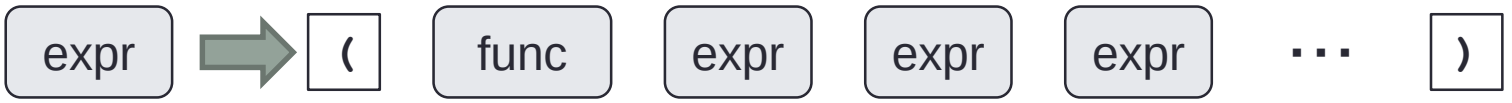
`(a b c d . e)`



Expression and Evaluation

- LISP program
  - Evaluate a given S expression.
  - No statements
    - Just expressions.
    - No if statement or while statement

- An expression is a function with its arguments as a list.
  - `(plus 1 2)`
  - `(times (plus 1 2) (plus 2 3))`



$f(x)$ Mathematics

`(f x)` LISP

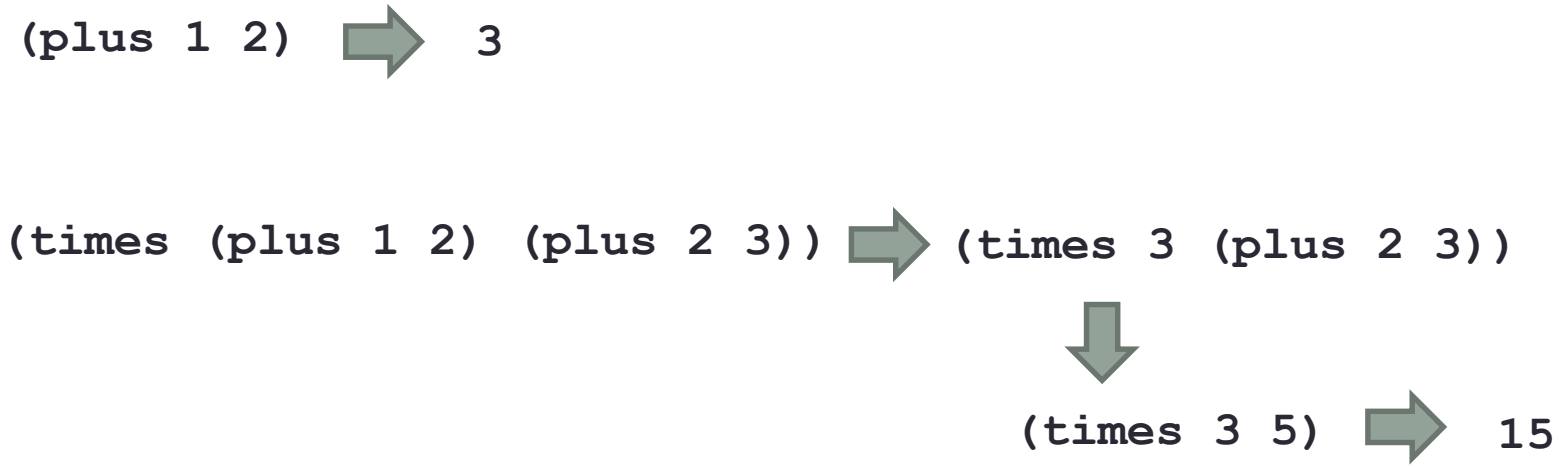
infix notation `1 + 2`

prefix notation (polish notation) `+ 1 2`

postfix notation (reverse polish notation) `1 2 +`

Evaluation of Expression

- LISP program
  - Evaluate a given S expression
  - Simplifies expressions



Basic Functions (1)

- **`quote`**: argument as value

```
(quote (a b)) => (a b)
```

  - **`quote`** can be abbreviated as `'`

```
'(a b) => (quote (a b)) => (a b)
```

- **`atom`**: checks whether it is an atom or not. returns `t` if it is and `nil` if not.

```
(atom 'abc) => t
(atom '(a b)) => nil
```

- **`eq`**: checks whether they are the same object or not.

```
(eq 'a 'a) => t
(eq 'a 'b) => nil
(eq '(a b) '(a b)) => nil
```

  - Just compares two pointers (i.e. addresses)
  - Atoms and numbers are unique.

Basic Functions (2)

- **cons**: creates a cons cell

```
(cons 'a 'b) => (a . b)
(cons 'a '(b c)) => (a b c)
```

- **car**: returns the left hand object of the cell

```
(car '(a b)) => a
```

- **cdr**: returns the right hand object of the cell

```
(cdr '(a b)) => (b)
(cdr '(a b c)) => (b c)
```

- **cond**: conditional expression

```
(cond ((eq x 'a) 'yes)
      ((eq x 'b) 'no)
      ('t 'unknown))
```

Function Definition

- Lambda notation

```
(lambda (x y) (cons x (cons y '())))
```

  - argument: `x`, `y` body: `(cons x (cons y '()))`

- Function application

```
((lambda (x y) (cons x (cons y '()))) 1 2)
   => (1 2)
```

  - bind values to arguments and evaluate the body.

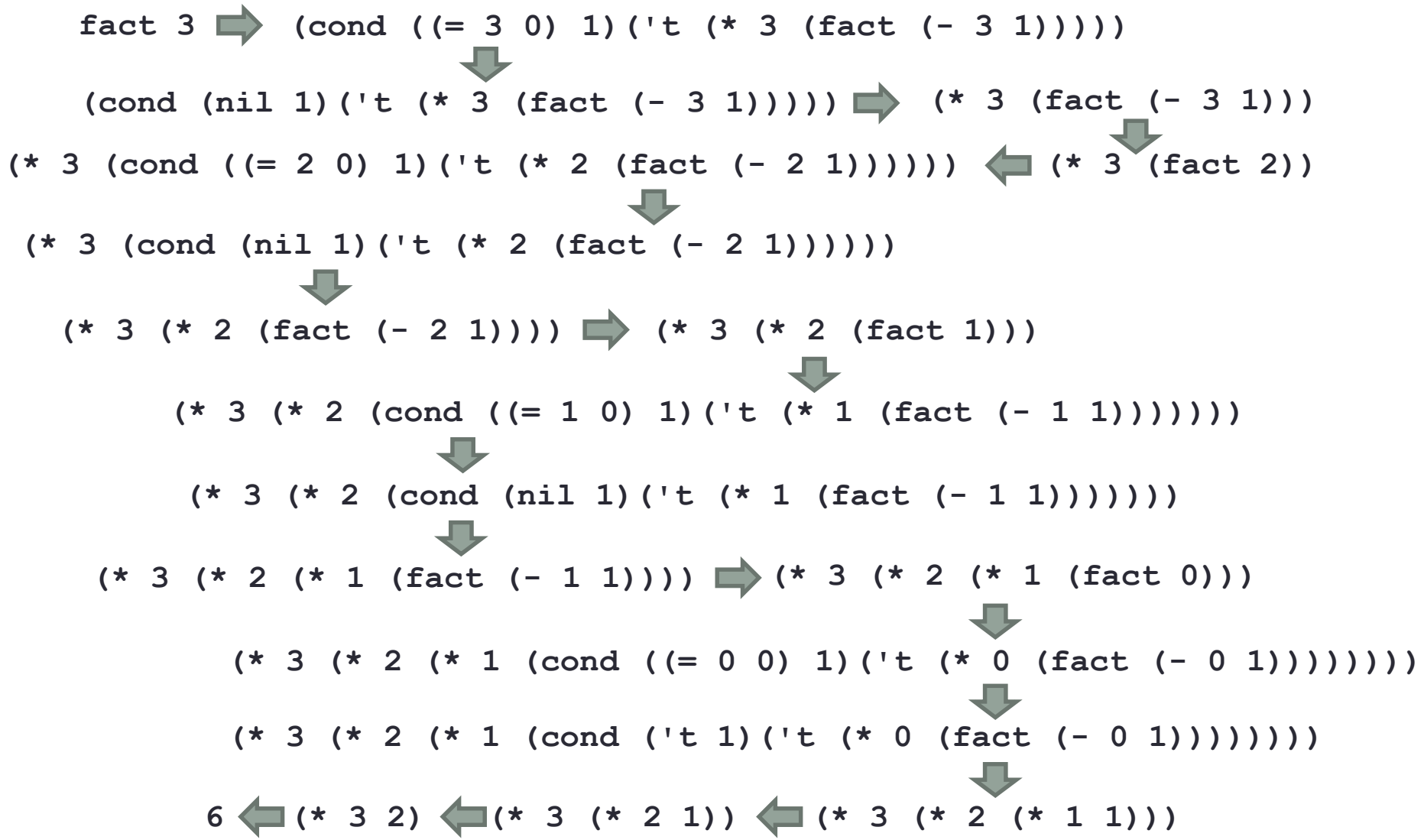
- Define functions with names

```
(define fact
   (lambda (x)
      (cond ((= x 0) 1)
            ('t (* x (fact (- x 1)))))))
```

  - allow recursive call (no loop statement)

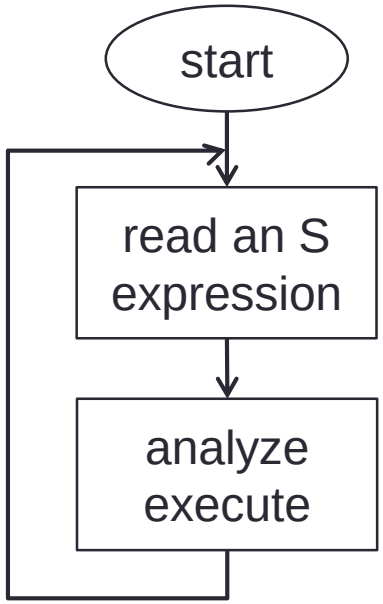
Function Evaluation

```
(define fact
   (lambda (x)
      (cond ((= x 0) 1)
            ('t (* x (fact (- x 1)))))))
```



LISP Interpreter

- LISP interpreter directly executes programs.



Reads an S expression and creates a list.

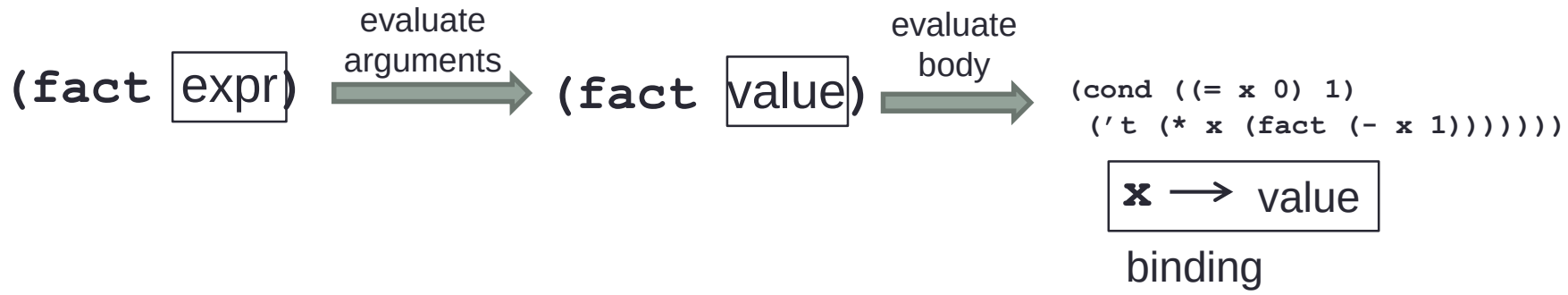
Simplifies S expressions and applies functions.

Execution

- Basic functions
  - Evaluate arguments to values
  - Apply basic functions

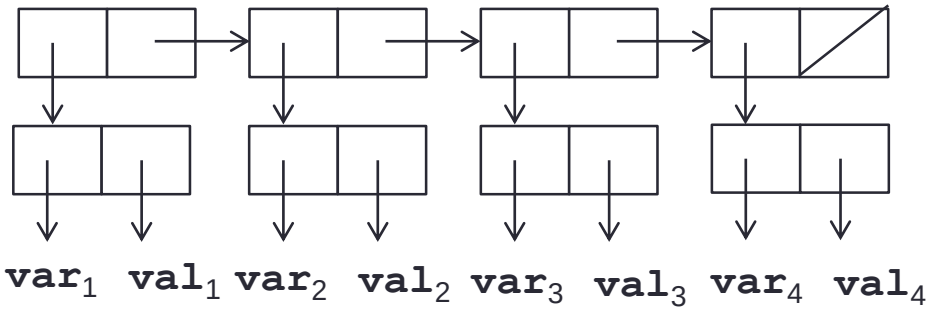
- User defined functions
  - Evaluate arguments to values
  - Evaluate the function body with binding the values to formal arguments

```
(define fact
   (lambda (x)
      (cond ((= x 0) 1)
            ('t (* x (fact (- x 1)))))))
```

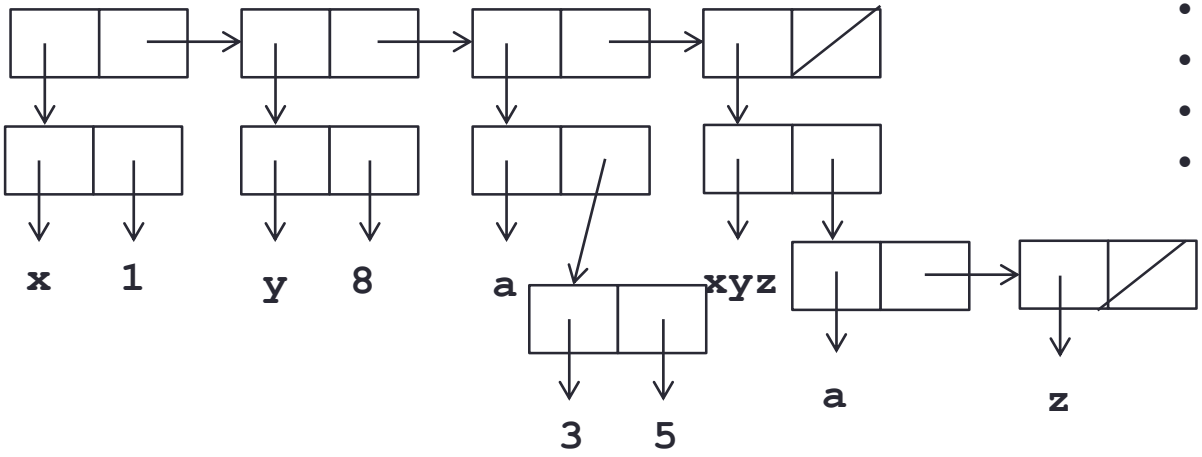


Associative List

- Expressions are evaluated with a value binding.
  - A value binding is called an environment.
  - An environment holds values of variables.
  - Associative list
    - associative memory
    - array with name indexes



- 例



- **x** is **1**
- **y** is **1**
- **a** is **(3 . 5)**
- **xyz** is **(a z)**

LISP Interpreter in LISP (1)

```
(define null (lambda (x) (eq x nil)))

(define and (lambda (x y) (cond (x (cond (y 't) ('t nil))) ('t nil))))

(define not (lambda (x) (cond (x nil)('t 't))))

(define append (lambda (x y)
  (cond ((null x) y)
  ('t (cons (car x) (append (cdr x) y))))))

(define list (lambda (x y) (cons x (cons y nil))))

(define pair (lambda (x y)
  (cond ((and (null x) (null y)) nil)
        ((and (not (atom x)) (not (atom y)))
         (cons (list (car x) (car y)) (pair (cdr x) (cdr y)))))))

(define assoc (lambda (x y)
  (cond ((eq (caar y) x) (cadar y))
        ('t (assoc x (cdr y))))))

(define caar (lambda (x) (car (car x))))
(define cadr (lambda (x) (car (cdr x))))
(define cadar (lambda (x) (cadr (car x))))
(define caddr (lambda (x) (cadr (cdr x))))
(define caddar (lambda (x) (caddr (car x))))
```

LISP Interpreter in LISP (2)

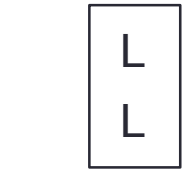
```
(define evcon (lambda (c a)
  (cond ((eval (caar c) a) (eval (cadar c) a))
        ('t (evcon (cdr c) a)))))

(define evlis (lambda (m a)
  (cond ((null m) nil)
        ('t (cons (eval (car m) a) (evlis (cdr m) a))))))

(define eval (lambda (e a)
  (cond ((atom e) (assoc e a))
        ((atom (car e))
         (cond ((eq (car e) 'quote) (cadr e))
               ((eq (car e) 'atom) (atom (eval (cadr e) a)))
               ((eq (car e) 'eq) (eq (eval (cadr e) a) (eval (caddr e) a)))
               ((eq (car e) 'car)(car (eval (cadr e) a)))
               ((eq (car e) 'cdr) (cdr (eval (cadr e) a)))
               ((eq (car e) 'cons) (cons (eval (cadr e) a) (eval (caddr e) a)))
               ((eq (car e) 'cond) (evcon (cdr e) a))
               ('t (eval (cons (assoc (car e) a) (cdr e)) a))))
        ((eq (caar e) 'label) (eval (cons (caddar e) (cdr e))
                                    (cons (list (cadar e) (car e)) a)))
        ((eq (caar e) 'lambda)
         (eval (caddar e) (append (pair (cadar e) (evlis (cdr e) a)) a))))))
```

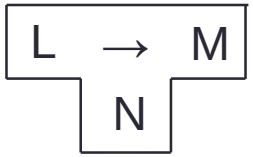
LISP in LISP

- LISP is explained in LISP.
  - **`eval`** is the function of evaluation.
  - **`(eval e a)`**: evaluate **`e`** under **`a`** (value binding)



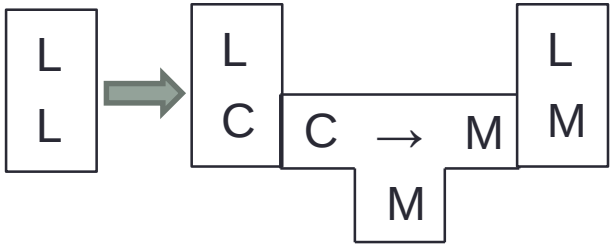
LISP interpreter
written in Lisp

- T scheme



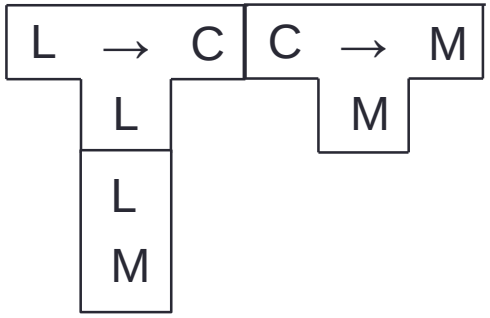
L to M compiler
written in N

L interpreter
written in N



LISP interpreter
written in C

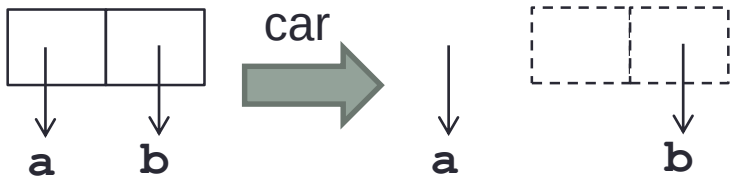
extend LISP interpreter



LISP compiler
written in LISP

Garbage Collection

- Garbage
  - **`cons`** consumes a cons cell
  - Unreferenced cons cells become garbage.
  - **`(car (cons 'a 'b))`**



- Collect and recycle garbage
  - Garbage Collection (GC)

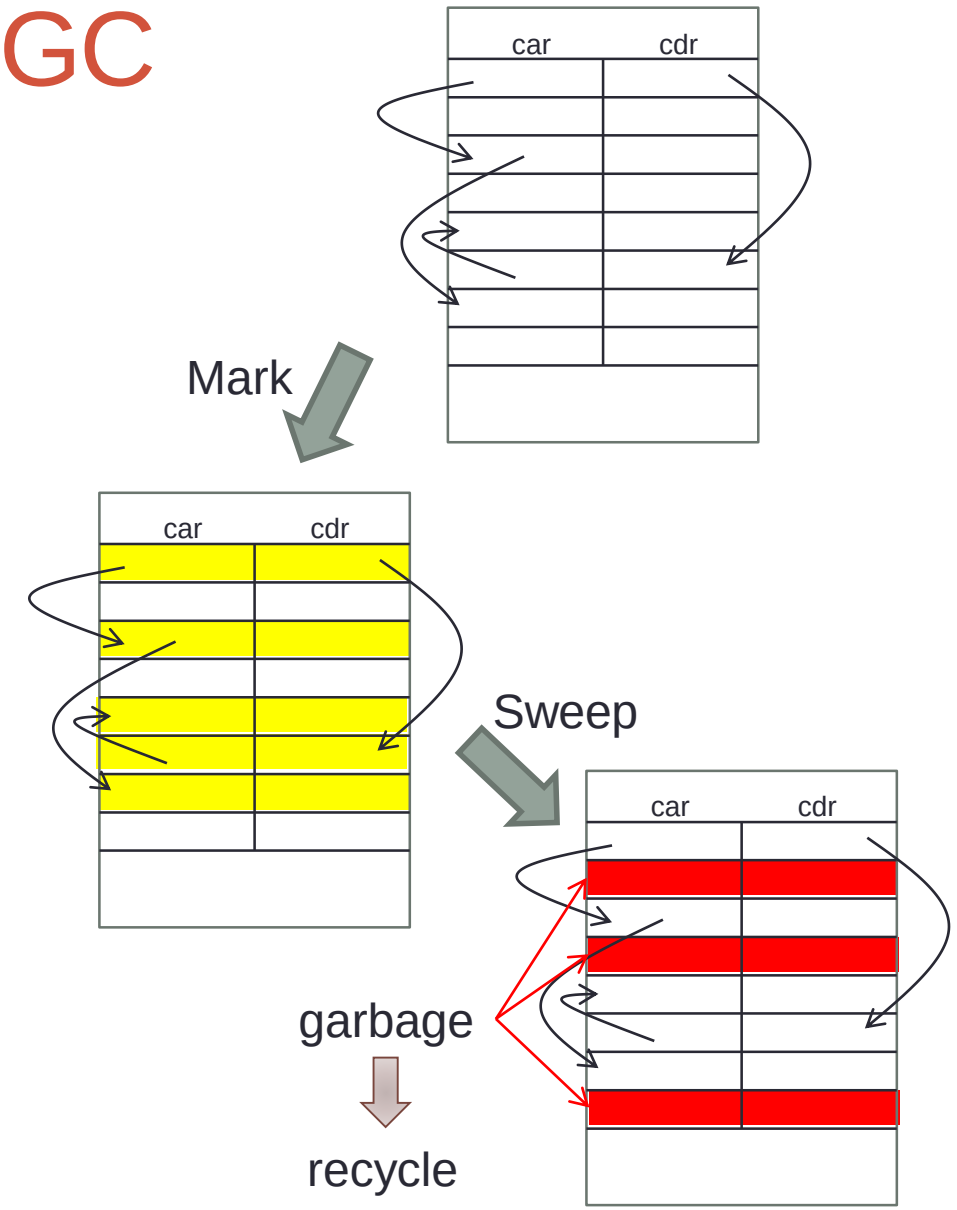
- Garbage collection algorithm
  - mark and sweep
  - compaction GC
  - copy GC
  - reference GC
  - realtime GC

Mark and Sweep GC

- Mark cells which are used.
- Collect unmarked cells.

```
struct cons {
   struct cons *car;
   struct cons *cdr;
   int marked;
} cell[N], *free;

mark(struct cons *x) {
   if (x->marked) return;
   x->marked = 1;
   mark(x->car);
   mark(x->cdr);
}
sweek() {
   free = 0;
   for (i = 0; i < N; i++) {
      if (!cell[i].marked) {
         cell[i].car = free;
         free = &cell[i];
      }
   }
}
```

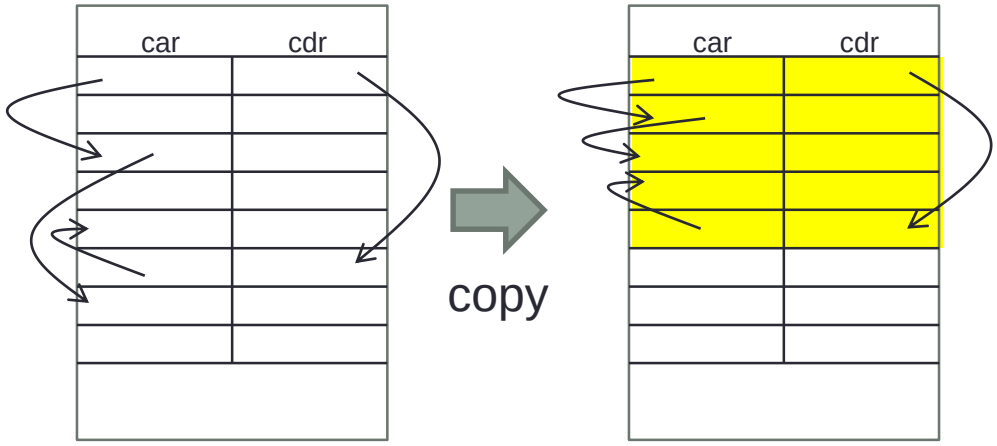


Copying GC

- Copy referenced cells to new memory region.
  - Can gather the referenced cells together.
  - The half of the memory is not used.

```
struct cons {
   struct cons *car;
   struct cons *cdr;
   int copied;
};

copy(struct cons *x) {
   struct cons *y, *car, *cdr;
   if (x->copied) return x->car;
   car = x->car;
   cdr = x->cdr;
   x->car = y = new();
   x->copied = 1;
   y->car = copy(car);
   y->cdr = copy(cdr);
   return y;
}
```



Other GC

- Compaction GC
  - Move cells to remove unused space.
  - Time consuming
- Reference Count
  - Each cell has a reference counter which holds the number of pointers which points to the cell.
  - When the reference counter becomes zero, it is recycled.
  - Lists with cycle cannot be recycled.
- Realtime GC
  - Do not stop the evaluation which GC
  - GC as background

Summary

- LISP
  - Basic functions
  - Lambda expressions
  - LISP interpreter
  - Garbage collection

- Mid Term Exam
  - 2018/6/5 from 9:30 (duration 45min)
  - You may refer the lecture slides.
  - You may no use any PC or electric devices.