Program is Data

```
(LAMBDA (fn) (fn  fn))
              ↙    ↘
      variable     variable
         ↓             ↓
     function        form
        ↙               ↘
```

Uses arg. as a <u>function</u>   Uses arg. as an <u>s-expression</u>

program   data

of	(LAMBDA (x) (CAR x))	is	LAMBDA
of	CAR	is	*undefined*
of	(LAMBDA (x) x)	is	(LAMBDA (x) x)
of	(LAMBDA (x) (x x))	is	*undefined*

– *non-terminating computation*

Using Programs as Data

- Taking apart a program
  for analysis
- Putting together a programs at Run Time
  Then Running it.
- Merging together programs!

Using Program "Code"

```
→ (DEF foo (LAMBDA (x) (CADR x)))
foo
→ (foo '(A B C))
B
→ (GETD 'foo)
(LAMBDA (x) (CADR x))
→ (CONS (CAR (GETD 'foo)) '((u y z) y))
(LAMBDA (u y z) y)
→ (PUTD 'bar (CONS (CAR (GETD 'foo)) '((u y z) y)))
(LAMBDA (u y z) y)
→ (bar 4 (+ 2 3) '(t Q))
5
→ (DEF N-args (LAMBDA (x) (LENGTH (CADR (GETD x)))))
N-args
→ (N-args 'foo)
1
→ (N-args 'bar)
3
→ (N-args 'N-args)
2
```


Compiler vs Interpreter

a COMPILER translates
Source Program into Executable Object Code

Steps in Compiler-Based System:
- read program
- check syntax
- compile: produce "object code"
  (discard "source" program)
- run object code

an INTERPRETER directly executes
(Source) Program

Steps:
- **read** next form
- **eval**uate (aka "execute") it
- **print** value

Notes:
1. form may be program – which is stashed as s-expr
2. only *run-time* checks

Lisp System

- Many *Lisps* have compilers.

- Most (all?) *Lisps* include INTERPRETERs. **READ-EVAL-PRINT** Loop

Eg:
```
→ (CONS (CAR '(A B)) '(C D))
(A C D)

→ (SETQ y 5)
5

→ (DEFUN foo (x) (CADR x))
foo

→ (foo '(A (B)))
B
```


Lisp Interpretation: `EVAL*`

- Interpretation is based on Evaluation which maps S-expr into S-expr

Eg: `(CONS (CAR '(A B)) '(C D))` $\leadsto$ `(A C D)`

- Can write a *Lisp* Function to do it!
  `EVAL*` of ⟨form⟩ is ⟨form⟩'s value.

- Use Subset of *Lisp*:

```
⟨form⟩ ::=  (QUOTE ⟨s-exp⟩) |
            (CAR ⟨form⟩) | (CDR ⟨form⟩) |
            (CONS ⟨form⟩ ⟨form⟩) |
            t | nil
```

Eg:
```
→ (EVAL* '(CONS t nil)
(t)

→ (EVAL* '(CONS (CAR '(A B)) '(C D)))
(A C D)
```


`EVAL*` wrt Variables

Problem: What is
`→ (EVAL* '(CONS x '(B C)))` ?

Solution: Specify the **CONTEXT**
which identifies `x` with its value, . . .

- Represent CONTEXT as AssocList
  `( (x . 'foo) (y . (t nil)) (z . nil) )`

  Means value of $\left\{ \begin{array}{l} \text{x} \\ \text{y} \\ \text{z} \end{array} \right.$ is 'foo, (t nil), nil

- `EVAL` takes 2 args: form + context

```
→ (EVAL '(CONS x '(B C))
        '( (x . t) (y . (t nil)) (z . nil) ))
(t B C)
```


`EVAL` in General

- `EVAL` form + context $\leadsto$ s-expr

  (Available in FranzLisp; but context ≠ alist.)

→ e ⟺ → (eval 'e nil) s   s `EVAL` of `e` (with `nil` context) is `s`

- `EVAL` is a function;
  Can use like any other function!

- Can take only 1 arg
  as if context = `nil`

- Examples: p127-28

Examples of EVAL – I

```
→ '(CONS 'a '(b c))
(CONS 'a '(b c))
→ (setq x '(list '+ 3 4))
(list '+ 3 4)
→ (eval 'x)
(list '+ 3 4)
→ (eval (eval 'x))
(+ 3 4)
→ (eval (eval x))
7
→ setq y 'x)
x
→ (eval 'y)
x
→ (eval y)
(list '+ 3 4)
```

```
→ (EVAL '(CONS 'a '(b c)))
(a b c)
→ 'x
x
→ x
(list '+ 3 4)
→ (eval x)
(+ 3 4)
→ (eval '(eval x))
(+ 3 4)

→ (eval '(QUOTE y))
y
→ (eval (eval y))
(+ 3 4)
```


Examples of EVAL – II

```
→ (EVAL 'x   '( (y .  x) (z .  A) (x .  P)) )
P
→ (EVAL '(CONS (CAR x) y)
        '( (x .  (A B C)) (y .  (D E))) )
(A D E)
→ (EVAL '(QUOTE x)
        '( (y .  x) (z .  A) (x .  P)) )
x
→ ( (LAMBDA (x c) (EVAL x c))
    'W  '( (W . A) (X . B) ) )
A
→ ( (LAMBDA (x c) (EVAL 'W c))
    'fred '( (W . A) (X . B) ) )
A
→ ( (LAMBDA (x c) (EVAL x c))
    '(QUOTE W)  '( (W . A) (X . B) ) )
W
→ ( (LAMBDA (x c) (EVAL (EVAL x nil) c))
    '(QUOTE W)  '( (W . A) (X . B) ) )
A
```


Application of EVAL: Extending Language

- Want to define `IF` s.t.

  (IF e_1 e_2 e_3)

  behaves like

  (COND (e_1 e_2) (t e_3))

- Consider *function*:

```
(DEFUN if (test v1 v2)
  (COND  (test v1)
         (t v2)))
```

Problem: All Args get evaluated!

```
(if (ATOM x) x (CAR x))
```

↖ evaluated even when `test` ≡ `(ATOM x)` is true!

Solution: Control evaluation of args
Eval 1^{st} arg
If true ⇒ eval 2^{nd} arg
If false ⇒ eval 3^{rd} arg

NLAMBDA Special Form

Problem: Need to have access to
- Args *unevaluated*
- Context (to pass to EVAL)

Solution: Special form:

$$(\underbrace{\texttt{(NLAMBDA (v1 v2) }\langle\texttt{form}\rangle\texttt{)}}_{\text{the defn of special form}}\ \ \underbrace{s_1 \cdots s_n}_{\text{arbitrary s-exprs}}\ )$$

- Kinda like λ-expression,
  but ALWAYS 2 arguments
  1^{st} ≡ list of *unevaluated* args
  2^{nd} ≡ current context

Using NLAMBDA to Define If

```
(DEF  if (NLAMBDA (args ctx)
      (COND  ((EVAL (CAR args) ctx)
                (EVAL (CADR args) ctx) )
             ( t (EVAL (CADDR args) ctx)))))
```

```
→ (if (atom x) x (car x))
...
→ ((NLAMBDA (args ctx) (COND ( (EVAL (CAR args) ctx)
    (EVAL (CADR args) ctx) )( t (EVAL (CADDR args) ctx))))
   (atom x) x (car x))
...
```

Notes:

- NLAMBDA is part of special form
- args is list of 3 forms
- Must explicitly call EVAL as needed
- Can be done "in place"
  [w/o using DEF]

NLAMBDA in Action – I

```
→ ( (LAMBDA (w) (if (ATOM w) w (CAR w)))  'A)
    Enter λ with   { w ← A }
      Enter if  with { args ← ((ATOM w) w (CAR w))
                       ctx  ← ( (w .  A) ) }
        Enter EVAL  with { form ← (ATOM w)
                           ctx  ← ( (w .  A) ) }
        Exit EVAL  with T
        Enter EVAL  with { form ← w
                           ctx  ← ( (w .  A) ) }
        Exit EVAL  with A
      Exit if  with A
    Exit λ  with A
A
```


NLAMBDA in Action – II

```
→ ( (LAMBDA (w) (if (ATOM w) w (CAR w))) '(B C))
    Enter λ with   { w ← (B C) }
      Enter if  with { args ← ((ATOM w) w (CAR w))
                       ctx  ← ( (w .  (B C)) ) }
        Enter EVAL  with { form ← (ATOM w)
                           ctx  ← ( (w .  (B C)) ) }
        Exit EVAL  with nil
        Enter EVAL  with { form ← (CAR w)
                           ctx  ← ( (w .  (B C)) ) }
        Exit EVAL  with B
      Exit if  with B
    Exit λ  with B
B
```


NLAMBDA vs LAMBDA

- Can define LAMBDA-expression
  to take any number of args;
  NLAMBDA-expression always takes 2 args.

- LAMBDA always evaluates all args
  (as values are passed)
  NLAMBDA simply passes expressions
  [They appear as a list, as NLAMBDA's 1st arg]

(Hence, body of NLAMBDA can SELECTIVELY decide whether to evaluate each arg.)

- NLAMBDA is **NOT** a function
  (nor is LAMBDA)

NLAMBDA in FranzLisp

- NLAMBDA *IS* in FranzLisp
  (also called FEXPR)

```
(DEF name (NLAMBDA (args ctx) ⟨form⟩))

(DEFUN name fexpr (args ctx) ⟨form⟩)
```

- Can take only 1 arg
  (list of uneval-ed args)
  Here, as if *CONTEXT* is NIL.

- Also LEXPRs
  Any number of args, but all evaluated.

- Many uses:
  - Used in def'n of EVAL
  - Is sufficient to define other special forms:

Using NLAMBDA to Define QUOTE

```
(DEF kwote (NLAMBDA (args ctx) (CAR args)))
```

→ ((LAMBDA (x) (kwote x)) 'A)

Enter λ with { x ← A }

Enter kwote with { args ← (x) ; ctx ← ((x . A)) }

EVALuate (CAR args) with

{ args ← (x) ... }

Return x

Exit kwote with x

Exit λ with x

x

Behaves like QUOTE:

→ (kwote s) → (quote s)

s s

Using NLAMBDA to Define Other Special Forms

- Define LIST:

```
(DEF list (NLAMBDA (args ctx)
 (COND ((NULL args)   nil)
       (t (CONS (EVAL (CAR args) ctx)
                (EVAL (CONS 'list (CDR args))
                      ctx)  )))))
```

[Could use MapCar, APPLY within function.]

[Could use LEXPR, as all args always evaluated.]

- Define OR:

```
(DEF or (NLAMBDA (args ctx)
 (COND ((NULL args)          nil)
       ((EVAL (CAR args) ctx) (EVAL (CAR args) ctx))
       (t (EVAL          (CONS 'or (CDR args)) ctx)
))))
```

[Could use AnyOf, APPLY within function.]

EVAL vs APPLY

- APPLY does for functions what
  EVAL does for forms.

  APPLY takes
  - a function name,
  - a list of arguments, and
  - a context; and

  applies the function to the arguments
  (using context as needed).

  EVAL: form + context ⇝ s-expr

  APPLY: function + args + context ⇝ s-expr

$\xrightarrow[s]{} \underline{\texttt{(f s}_1 \cdots \texttt{s}_n\texttt{)}} \iff \xrightarrow[s]{} \underline{\texttt{(APPLY 'f '(s}_1 \cdots \texttt{s}_n\texttt{) nil)}}$

APPLY of f on $s_1 \cdots s_n$ (with nil context) is s

- Examples of APPLY: p133-34

Examples of APPLY

```
→ (APPLY 'CONS   '(A (B C))  nil)
(A B C)

→ (APPLY 'CONS   '(X (C D E))  '((X . P)) )
(X C D E)

→ (APPLY '(LAMBDA (a b) (CONS X b)
         '(Y (C D E))  '((X . P)) )
(P C D E)

→ (APPLY 'APPEND  '((A B)(C D E)) '((X . P)) )
(A B C D E)

→ (APPLY '(LAMBDA (x y) (EQ x y))
         '(A B)  nil )
nil

→ (APPLY '(LAMBDA (x y) (EQ x y))
         '(A B)  '((x . B)(y . B)) )
nil

→ (APPLY '(LAMBDA (x) (CONS (CAR x) w))
         '((A B C))
         '((x .  (D E F))(w . (G H I))) )
(A G H I)
```


Examples of APPLY (II)

```
→((LAMBDA (a b) (APPLY 'EQ (LIST a b) nil))
   t t)
t

→((LAMBDA (x) (APPLY (QUOTE (LAMBDA () (NULL nil)))
                      ()  '((x . T)) ))
   nil)
t

→((LAMBDA (x) (APPLY (QUOTE (LAMBDA () (ATOM x)))
                      ()  () ))
   nil)
t

→((LAMBDA (x) (APPLY (QUOTE (LAMBDA (y) (EQ x y)))
                      '(T)   '((x . T)) ))
   nil)
t

→((LAMBDA (x) (APPLY (FUNCTION (LAMBDA (y) (EQ x y)))
                      '(T)
                      '((x . T)) ))
   nil)
nil
```


Application of Apply
Object Oriented Programming

- "Top Level" Operations:

```
(Add 17 22)                  Integer Addition
(Add 22.3 -4.2E-1)           Real Addition
(Add (2 .  3) (9 .  -7))     Complex Addition
        for  [2 + 3i] + [9 - 7i]
...
```

Same "Add" operation,
but different (Low level) code involved.

Other datatypes – eg Matrix, Group, . . .
Other operations – eg Times, LessThan, . . .

Problem: System must determine appropriate Code,
dependent on DATA-Types of Args

Solution: . . .

DataType with Associated Operations

Integer:

```
Addition    (LAMBDA (x y) (PLUS x y))
Less-Than   (LAMBDA (x y) (LessP x y))
...
```

Real:

```
Addition    (LAMBDA (x y) (+ x y))
Less-Than   (LAMBDA (x y) (< x y))
...
```

Complex-Num:

```
Addition    (LAMBDA (x y) (CONS (+ (CAR x) (CAR y))
                                (+ (CDR x) (CDR y))))
Less-Than   (LAMBDA (x y) (AND  (< (CAR x) (CAR y))
                                (< (CDR x) (CDR y))))
...
```

Matrix:

```
Addition    (LAMBDA (x y) ...)
Less-Than   (LAMBDA (x y) ...)
...
```

. . .

Object Oriented Programming (con't)

- Code for `add`:

```
(DEFUN  add (x y)
        (APPLY  (Find-Addition-Method x y)
                (LIST x y) nil))
```

- `Find-Addition-Method`
  (1) Determine "data type" of args
  ["Real" for args `22.3`, `-15.2`]

  (2) Find method for that operation
  for that data types
  (using default, inheritance, . . .)
  ["`(LAMBDA (x y) (+ x y))`" for Real Addition]

- Similar function for `Times`, `LessThan`, . . .

`APPLY` in General

- `Apply` is defined in FranzLisp
  (Context $\neq$ alist)

- It can take (only) 2 args:
  Function
  List of arguments
  (Context taken to be `nil`)

- Also `Funcall`:
  Like `Apply`, but takes $n + 1$ args:
  First is function;
  $i + 1^{st}$ is i^{th} arg to function.

Examples of `Apply`, `Funcall`

```
→ (apply '+ (3 5))
8

→ (funcall '+ 3 5)
8

→ (apply 'car '((a b c)))
a

→ (apply 'car '(a b c))
a

→ (apply '(lambda (x) (cadr x))  '((a b c)))
b

→ (funcall '(lambda (x) (cadr x))  '(a b c))
b

→ (apply '(lambda (x y) (eq x y))  '(a b))
nil

→ (funcall '(lambda (x y) (eq x y))  'a 'b)
nil
```