Symbolic Reasoning

Dr. Neil T. Dantam

CSCI-534, Colorado School of Mines

Spring 2020

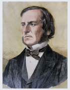
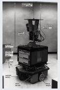
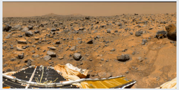
Introduction

What is Symbolic Reasoning?

Definition (Symbolic Reasoning)

Inference using **symbols**—abstract items which stand for something else—coupled with rules to rewrite or transform symbolic expressions.

Example

Algebra, Calculus	Computer Algebra	Symbolic Planning
  		 

Introduction

Overview and Outcomes

Overview

- Symbolic Reasoning: rewriting symbolic expressions
- Manual:
  - Algebra
  - Calculus
- Algorithmic:
  - computer algebra
  - symbolic planning

Outcomes

- Relate infix notations and symbolic data structures.
- Understand the *s-expression* notation.
- Apply recursion to s-expressions.
- Implement algorithms for symbolic reasoning.

Outline

Rewrite Systems

Definition (Rewrite System)

A well defined method for mathematical reasoning employing axioms and rules of inference or transformation. A **formal system** or **calculus**.

Example (Expressions)

- Arithmetic:
  - $a_0x + a_1x^2 + a_3x^3$
  - $3x + 1 = 10$
- Propositional Logic:
  - $(p_1 \vee p_2) \wedge p_3$
  - $(p_1 \wedge p_2) \implies p_3$

Example (Reductions)

- Distributive Properties:
  - $\left(x * (y + z)\right) \rightsquigarrow \left(xy + xz\right)$
  - $\left(\alpha \vee (\beta \wedge \gamma)\right) \rightsquigarrow \left((\alpha \vee \beta) \wedge (\alpha \vee \gamma)\right)$
- De Morgan's Laws:
  - $\left(\neg(\alpha \wedge \beta)\right) \rightsquigarrow \left((\neg\alpha \vee \neg\beta)\right)$
  - $\left(\neg(\alpha \vee \beta)\right) \rightsquigarrow \left((\neg\alpha \wedge \neg\beta)\right)$

Progressively apply reductions until reaching desired expression.

Example: Algebra

Given: $3x + 1 = 10$

Find: x

Solution:

Initial	$3x + 1$	$=$	10
-1	$3x + 1 - 1$	$=$	$10 - 1$
Simplify	$3x$	$=$	9
$/3$	$3x/3$	$=$	$9/3$
Simplify	x	$=$	3

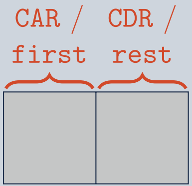
How would you write a program to solve for x?

The Cons Cell

Declaration

```
struct cons {
    void *first;
    struct cons *rest;
};
```

Diagram

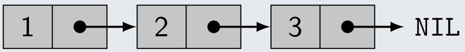


Example

S-Expression

(1 2 3)

Cons Cell Diagram



Abstract Syntax

Infix

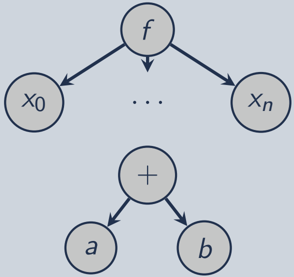
- Function / Operator
- Arguments / Operands

$$f(x_0, \ldots, x_n)$$

$$a + b$$

Abstract Syntax Tree

- **Root:** Function / Operator
- **Children:** Arguments / Operands



S-Expression

- **First:** Root, Function / Operator
- **Rest:** Children, Arguments / Operands

$$(f\ x_0\ \ldots\ x_n)$$

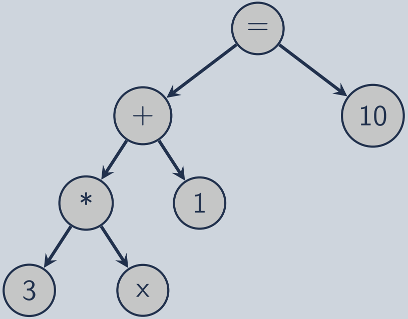
$$(+\ a\ b)$$

Example: S-Expression

Infix Expression

$3x + 1 = 10$

Abstract Syntax Tree



S-expression

```
(= (+ (* 3 x) 1) 10)
```

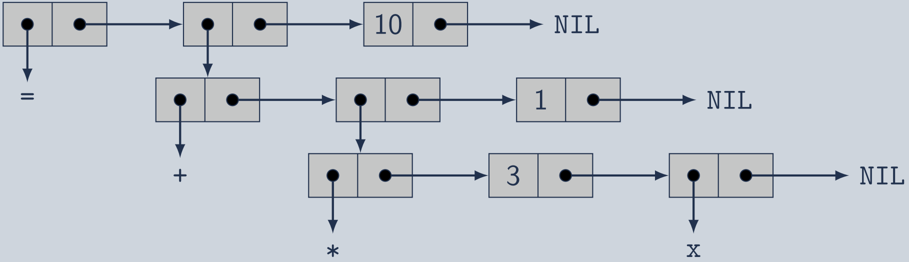
```
(=   (+    (* 3 x)
            1)
      10)
```

Example: Cell Diagram

S-Expression

```
(=   (+    (* 3 x)
           1)
     10)
```

Cell Diagram



List vs. Tree

List

```
struct cons {
    void *first;
    struct cons *rest;
};
```

Tree

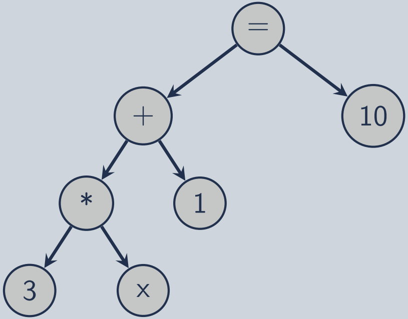
```
struct treenode {
    void *first;
    struct cons *children;
};

struct cons {
    void *first;
    struct cons *rest;
};
```

Same structure

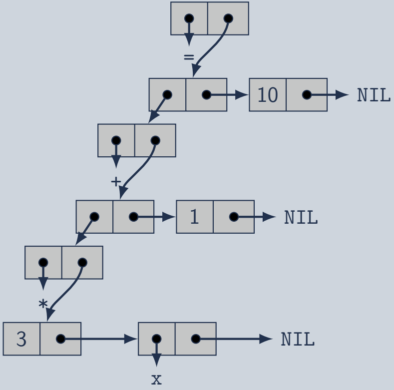
Data Structure, Redux

$3x + 1 = 10$

Tree



Cons Cells



Exercise 1: S-Expression

2(x-1) = 4

$$2(x-1) = 4$$

Exercise 2: S-Expression

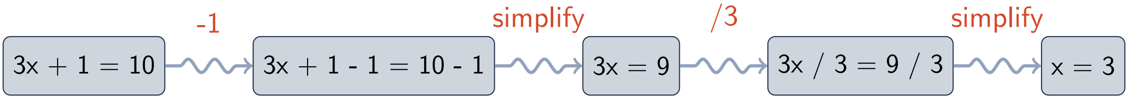
$a + bx + cx^2$

$$a + bx + cx^2$$

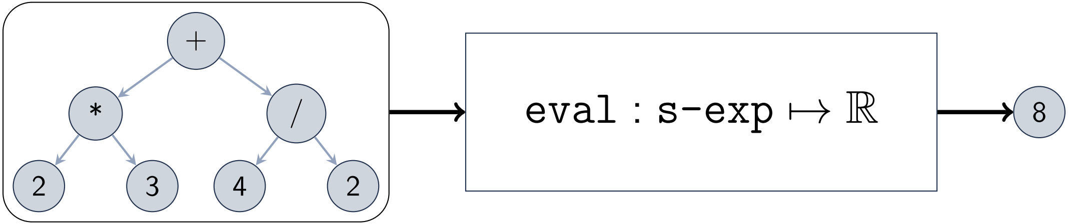
Exercise 2: S-Expression

$a + bx + cx^2$ – continued

Rewrites



Evaluation Function



Recursive Evaluation Algorithm

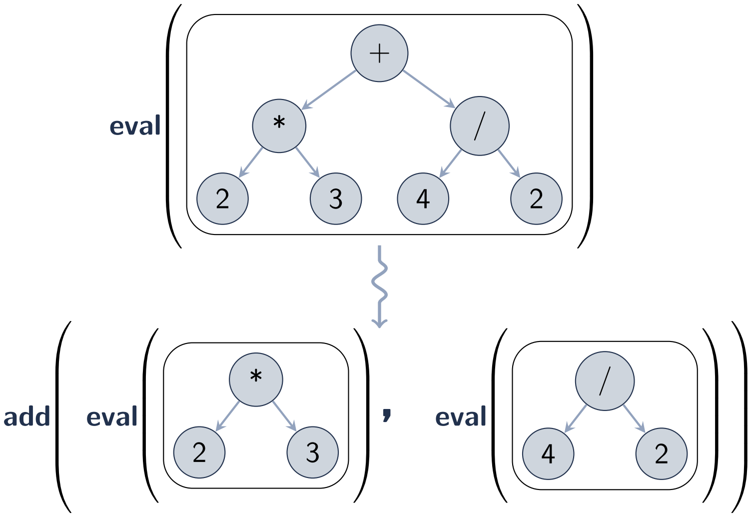
Base Case: If argument is a value: return the value

Recursive Case: Else (argument is an expression):

1. Recurse on arguments
2. Apply operator to results

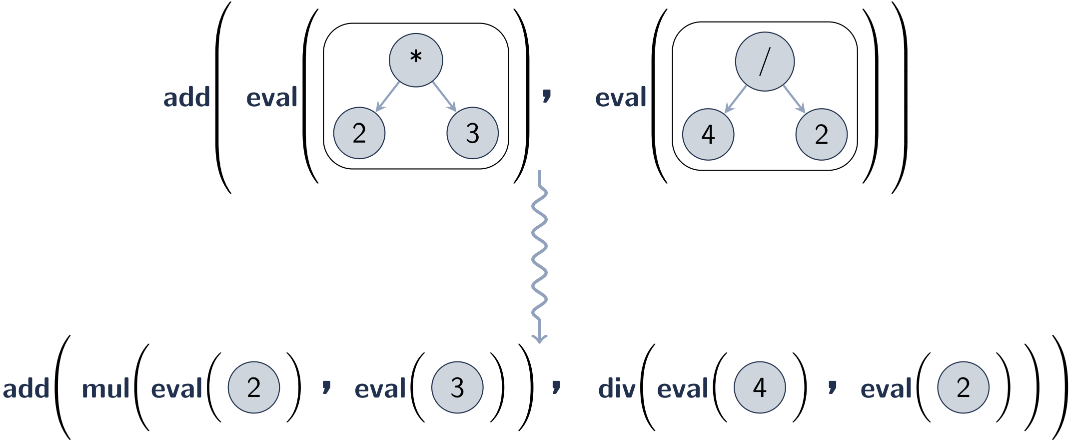
Example: Evaluation

2*3 + 4/2



Example: Evaluation

2*3 + 4/2 – continued



Example: Evaluation

2*3 + 4/2 – continued

$$\mathbf{add}\Big(\mathbf{mul}\big(\mathbf{eval}(2), \mathbf{eval}(3)\big), \mathbf{div}\big(\mathbf{eval}(4), \mathbf{eval}(2)\big)\Big)$$

$$\rightsquigarrow$$

$$\mathbf{add}\Big(\mathbf{mul}(2, 3), \mathbf{div}(4, 2)\Big)$$

Example: Evaluation

2*3 + 4/2 – continued

$$\mathbf{add}\left(\mathbf{mul}\left(2 , 3 \right) , \mathbf{div}\left(4 , 2 \right) \right)$$

$$\rightsquigarrow$$

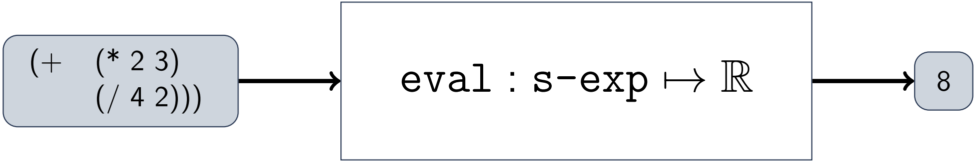
$$\mathbf{add}\left(6 , 2 \right)$$

$$\rightsquigarrow$$

$$8$$

Evaluation via S-Expressions

2*3 + 4/2



Outline

Evaluation and Quoting

Evaluation

Evaluating (executing) an expression and yielding its return value:

(`fun` a b c)
$\rightsquigarrow$ return value of `fun` applied to a, b, and c

Example

- (+ 1 2) $\rightsquigarrow$ 3
- 1 $\rightsquigarrow$ 1

Quoting

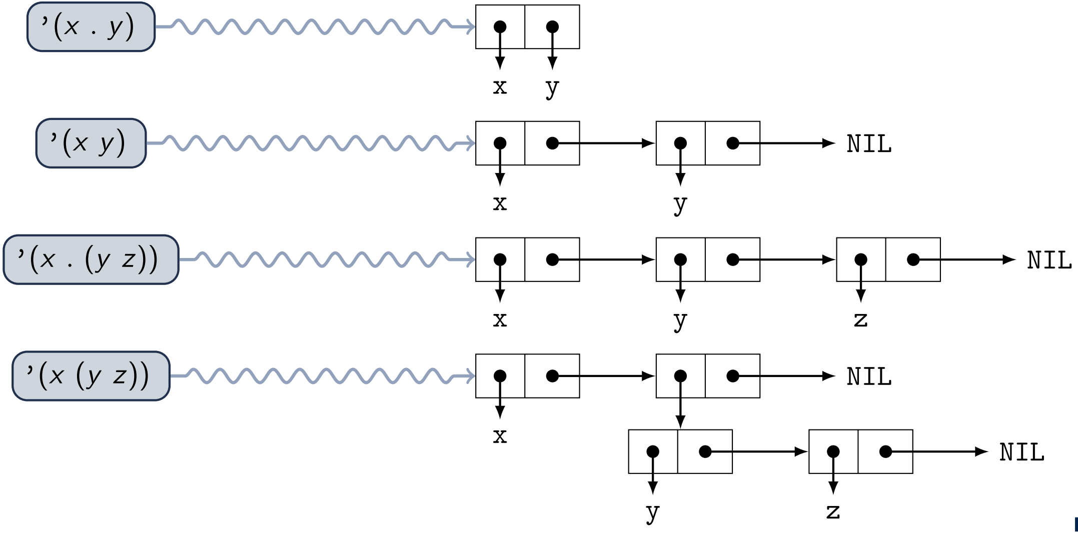
Returns the quoted s-expression:

'(`fun` a b c)
$\rightsquigarrow$ The s-expression: (`fun` a b c)

Example

- '(+ 1 2) $\rightsquigarrow$ (+ 1 2)
- '1 $\rightsquigarrow$ 1
- 'x $\rightsquigarrow$ x
- (quote x) $\rightsquigarrow$ x

Dotted List Notation



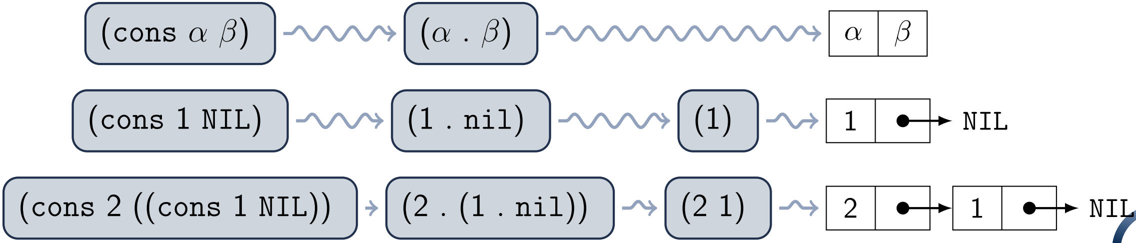
CONStruct

Creating Lists

CONS

Contruct a new cons cell:

(cons x y)
$\rightsquigarrow$ a fresh cons cell with x in the car (first) and y in the cdr (rest)



List Function

LIST

Return a list containing the supplied objects:

`(list` a_0 `...` a_n`)`

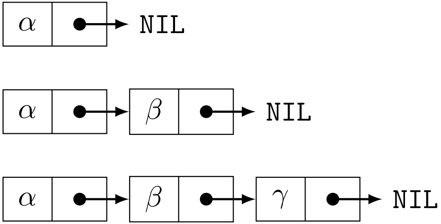
$\rightsquigarrow$ a list containing objects $a_0, \ldots, a_n$

`(list)` $\rightsquigarrow$ `NIL`

`(list` α`)` $\rightsquigarrow$ `(cons` α `NIL)` $\rightsquigarrow$

`(list` α β`)` $\rightsquigarrow$ `(cons` α `(list` β`))` $\rightsquigarrow$

`(list` α β γ`)` $\rightsquigarrow$ `(cons` α `(list` β γ`))` $\rightsquigarrow$



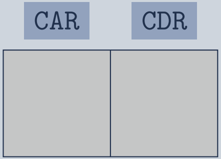
Exercise: List Construction

- (cons 'x 'y) $\rightsquigarrow$
- (cons 'x '(y z)) $\rightsquigarrow$
- (cons 'x (list 'y 'z)) $\rightsquigarrow$
- (list (+ 1 2 3)) $\rightsquigarrow$
- (list '(+ 1 2 3)) $\rightsquigarrow$
- (list '$*$ $(+\ 2\ 2)$ '$(-\ 2\ 2)$) $\rightsquigarrow$

List Access

CAR / CDR

Cons cell



CAR

Return the car of a cons cell:

`(car cell)`
$\rightsquigarrow$ the car (first) of `cell`

`(car '`$(\alpha\ .\ \beta)$`)` $\rightsquigarrow$ α

CDR

Return the cdr of a cons cell:

`(cdr cell)`
$\rightsquigarrow$ the cdr (rest) of `cell`

`(cdr '`$(\alpha\ .\ \beta)$`)` $\rightsquigarrow$ β

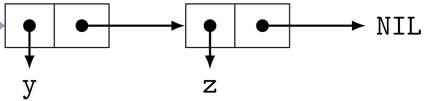
Example: CAR / CDR

`(car '(x . y))` ⇝ x

`(cdr '(x . y))` ⇝ y

`(car '(x y z))` ⇝ x

`(cdr '(x y z))` ⇝



List Template Syntax

Backquote (‘): Create a template

- ‘$(x_0 \ldots x_n)$ ⇝ `(list` ’x_0 … ’x_n`)`
- ‘(+ a (* b c)) ⇝ `(list` ’+ ’a `(list` ’$*$ ’b ’c`))` ⇝ $(+\ a\ (*\ b\ c))$

Comma (,)

Evaluate and **insert**:

- ‘$(\alpha \ldots\ ,y\ \beta \ldots)$ ⇝ $\left(\texttt{list}\ \alpha \ldots \overbrace{y}^{\text{evaluated}}\ \beta \ldots\right)$
- ‘$(+\ a\ ,(*\ 2\ 3))$
  ⇝ `(list` ’+ ’a $(*\ 2\ 3))$
  ⇝ $(+\ a\ 6)$

Comma-At (,@)

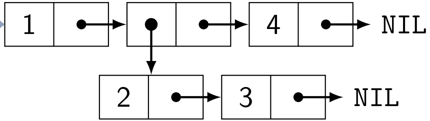
Evaluate and **splice**:

- ‘$(\alpha \ldots\ ,@y\ \beta \ldots)$ ⇝
  $(\text{append}\ \alpha \ldots \overbrace{y}^{\text{splice}}\ \beta \ldots)$
- ‘(+ a ,@ `(list` $(*\ 2\ 3)\ (*\ 4\ 5))$)
  ⇝ (append ’$(+\ a)$ `(list` $(*\ 2\ 3)\ (*\ 4\ 5)))$
  ⇝ $(+\ a\ 6\ 20)$

Comma (,) vs. Comma-At (,@)

Comma ,: Insert

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



Comma-At ,@: Splice

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

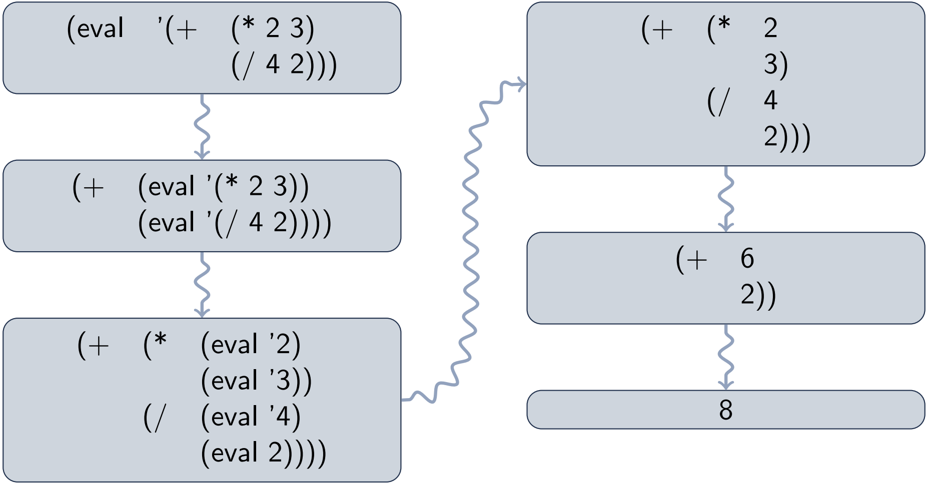
Exercise: List Template Syntax

- `` `(1 2 ,(+ 3 4)) `` ⇝
- `` `(,1 ,2 (+ 3 4)) `` ⇝
- `` `(+ 1 ,2 ,(+ 3 4)) `` ⇝
- `` `(1 2 ,@(list '+ '3 '4)) `` ⇝

Outline

Example: Evaluation via S-Expressions

2*3 + 4/2



Evaluation Algorithm

Procedure eval(e)

```
if value?(e) then /* Argument is a value */
    return e;
else /* Argument is an expression */
    operator ← first(e) ;
    arg-sexp ← rest(e) ;
    arg-vals ← map (eval, arg-sexp);
    switch operator do
        case '+ do f ← +;
        case '- do f ← −;
        case '/ do f ← /;
        case '* do f ← ∗;
    return apply(f, arg-vals);
```

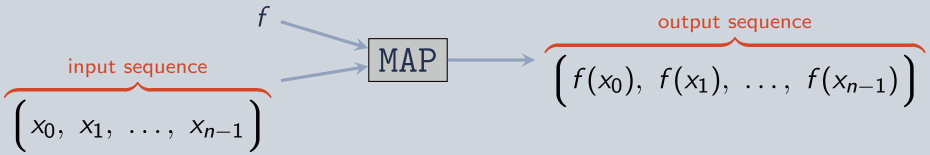
Map function

Definition (map)

Apply a function to every member of an input sequence, and collect the results into the output sequence.

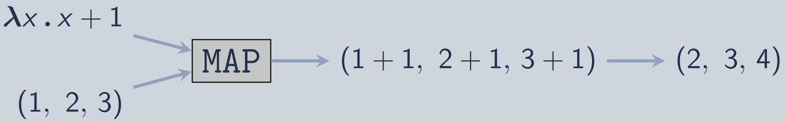
$$\texttt{map} : \underbrace{(\mathbb{X} \mapsto \mathbb{Y})}_{\text{function}} \times \underbrace{\mathbb{X}^n}_{\text{input sequence}} \mapsto \underbrace{\mathbb{Y}^n}_{\text{output sequence}}$$

Illustration

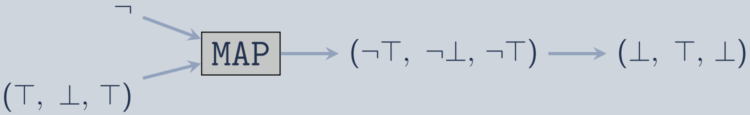


Example: Map

+1



¬



Algorithm: Map function

Functional Map

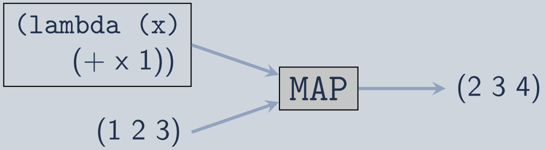
```
Procedure map(f,s)
if empty(s) then /* s is empty */
    return NIL
else /* s has members */
    return
      cons(f(first(s)), map(f, rest(s)));
```

Imperative Map

```
Procedure map(f,s)
n ← length(s);
Y ← make-sequence(n);

i ← 0;
while i < n do
    Y[i] ← f(s[i]);
    i ← i + 1;
return Y;
```

Example: Map

Example (Illustration)



Example (Lisp)

```
(map 'list                        ; result type
     (lambda (x) (+ 1 x))  ; function
     (list 1 2 3))               ; sequence
;; RESULT: (2 3 4)
```

Exercise 1: Evaluation

$$2*(1+2+3) - 5$$

Exercise 1: Evaluation

continued

Example: Partial Evaluation

Given:
- $f(x_0, x_1, x_2) = x_2(2x_0 + 3x_1 + x_2)$
- $a = 1$
- $b = 2$

Find: Simplification of $f(a, b, c)$

Solution:

initial	$c(2a + 3b + c)$
substitute	$c(2 * 1 + 3 * 2 + c)$
evaluate	$c(2 + 6 + c)$
evaluate	$c(8 + c)$
expand	$8c + c^2$

Partial Evaluation via S-Expressions

```
(*  (+  (* 2 a)
        (* 3 b)
        c)
    c)
```

```
(*  (+  (* 2 1)
        (* 3 2)
        c)
    c)
```

```
(*  (+  2
        6
        c)
    c)
```

```
(*  (+  8
         c)
     c)
```

```
(+  (* 8 c)
    (* c c))
```

Partial Evaluation Function

```
(*  (+  (* 2 a)
        (* 3 b)
        c)
    c)
```

```
(a . 1)
(b . 2)
```

$$\texttt{p-eval} : \texttt{s-exp} \times (\texttt{symbol} \times \mathbb{R})^* \mapsto \texttt{s-exp}$$

```
(+  (* 8 c)
    (* c c))
```

Recursive Partial Evaluation

```
(p-eval '(*  (+  (* 2 a)
                 (* 3 b)
                 c)
             c)
```

```
(p-eval-*  (p-eval '(+  (* 2 a)
                        (* 3 b)
                        c)
           (p-eval 'c))
```

```
(p-eval-*  (p-eval-+  (p-eval '(* 2 a))
                      (p-eval '(* 3 b))
                      (p-eval 'c))
           'c)
```

Recursive Partial Evaluation

continued

```
(p-eval-*  (p-eval-+  (p-eval '(* 2 a))
                      (p-eval '(* 3 b))
                      (p-eval 'c))
           'c)
```

↓

```
(p-eval-*  (p-eval-+  (p-eval-*  (p-eval 2)
                                 (p-eval 'a))
                      (p-eval-*  (p-eval 3)
                                 (p-eval 'b))
                      'c)
           'c)
```

Recursive Partial Evaluation

continued

```
(p-eval-*   (p-eval-+   (p-eval-*   (p-eval 2)
                                    (p-eval 'a))
                        (p-eval-*   (p-eval 3)
                                    (p-eval 'b))
                        'c)
            'c)
```

⇝

```
(p-eval-*   (p-eval-+   (p-eval-* 2 1)
                        (p-eval-* 3 2)
                        'c)
            'c)
```

Recursive Partial Evaluation

continued

```
(p-eval-*   (p-eval-+   (p-eval-* 2 1)
                        (p-eval-* 3 2)
                        'c)
            'c)
```

```
(p-eval-*   (p-eval-+   2
                        6
                        'c)
            'c)
```

```
(p-eval-*   '(+ 8 c)
            'c)
```

```
'(+   (* 8 c)
      (* c c))
```

Algorithm: Partial Evaluation

Procedure p-eval(e,bindings)

```
if number?(e) then
    return e;
else if symbol?(e) then
    if bindings[e] then return bindings[e] ;
    else return e ;
else
    y ← map (p-eval, rest(e));
    switch first(e) do
        case '+ do f ← p-eval-+;
        case '* do f ← p-eval-*;
        ...
    return apply (f, y);
```

Algorithm: Partial Evaluation

Continued – Addition

Algebraic Properties

Commutative: $\left(\alpha + \beta\right) \rightsquigarrow \left(\beta + \alpha\right)$

Associative: $\left((\alpha + \beta) + \gamma\right) \rightsquigarrow \left(\alpha + (\beta + \gamma)\right)$

Identity: $\left(\alpha + 0\right) \rightsquigarrow \left(\alpha\right)$

Procedure p-eval-+$(E \ldots)$

$N \leftarrow \{e \in E \mid \texttt{number?}(e)\}$;
$n \leftarrow$ **fold-left** $(+, 0, N)$;
$S \leftarrow \{e \in E \mid \neg\texttt{number?}(e)\}$;
if $0 = n$ **then**
  if $\emptyset = S$ **then return** 0;
  else if $1 = |S|$ **then return** $\texttt{first}(S)$;
  else return $\texttt{cons}\,('+,\ S)$;
else
  if $\emptyset = S$ **then return** n;
  else return $\texttt{cons}\,('+,\ \texttt{cons}\,(n,\ S))$;

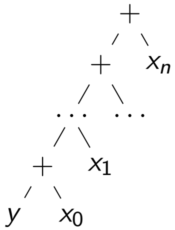
Fold-Left

Definition (fold-left)

Apply a binary function to every member of a sequence and the result of the previous call, starting from the left-most (initial) element.

$$\text{fold-left} : \underbrace{(\mathbb{Y} \times \mathbb{X} \mapsto \mathbb{Y})}_{\text{function}} \times \underbrace{\mathbb{Y}}_{\text{init.}} \times \underbrace{\mathbb{X}^n}_{\text{sequence}} \mapsto \underbrace{\mathbb{Y}}_{\text{result}}$$

Function Application



Fold-left Pseudocode

Procedural

Function fold-left(f,y,X)

```
i ← 0;
while i < |X| do
    y ← f(y, X_i) ;
return y;
```

Recursive

Function fold-left(f,y,X)

```
if empty?(X) then return y ; /* Base Case */
else /* Recursive Case */
    y' ← f (y, first(X));
    return fold-left(f, y', rest(X));
```

Exercise: Partial Evaluation

Given
- $a = 3$
- $b = 5$
- $c = 7$
- $e = \frac{a}{1+b+c} - d$

Find: Recursively simplify e

Solution:

Exercise: Partial Evaluation

continued – 1

Exercise: Partial Evaluation

continued – 2

Exercise: Partial Evaluation

continued – continued 3

Derivative

$$\frac{d\,f(t)}{dt} = \frac{\text{change in } f(t)}{\text{change in } t}$$

$$= \frac{\Delta f(t)}{\Delta t}$$

$$= \lim_{h \to 0} \frac{f(t+h) - f(t)}{h}$$

Differential Calculus

Rewrite Rules

Constant:	$\frac{d}{dt}k$	$\leadsto$	0
Variable:	$\frac{d}{dt}t$	$\leadsto$	1
Constant Power (var):	$\frac{d}{dt}t^k$	$\leadsto$	$k * t^{k-1}$
Constant Power (fun):	$\frac{d}{dt}f(t)^k$	$\leadsto$	$k * (f(t))^{k-1} * \frac{d}{dt}f(t)$
Addition:	$\frac{d}{dt}(f(t) + g(t))$	$\leadsto$	$\frac{d}{dt}f(t) + \frac{d}{dt}g(t)$
Subtraction:	$\frac{d}{dt}(f(t) - g(t))$	$\leadsto$	$\frac{d}{dt}f(t) - \frac{d}{dt}g(t)$
Multiplication:	$\frac{d}{dt}(f(t) * g(t))$	$\leadsto$	$\left(\frac{d}{dt}f(t)\right)g(t) + f(t)\left(\frac{d}{dt}g(t)\right)$
Division:	$\frac{d}{dt}\left(\frac{f(t)}{g(t)}\right)$	$\leadsto$	$\frac{\frac{d}{dt}f(t)}{g(t)} - \frac{f(t) * \frac{d}{dt}g(t)}{(g(t))^2}$
Chain Rule:	$\frac{d}{dt}f(g(t))$	$\leadsto$	$f'(g(t)) * \frac{d}{dt}g(t)$

Derivatives of Common Functions

Sine:	$\sin' x$	$\leadsto$	$\cos x$
Cosine:	$\cos' x$	$\leadsto$	$-\sin x$
Natural Logarithm:	$\ln' x$	$\leadsto$	$\frac{1}{x}$
Exponential:	$\exp' x$	$\leadsto$	$\exp x$

Differentiation Steps

$$\frac{d}{dt}\left(\cos t^2\right)$$

$$\frac{d}{dt} f(g(t)) \rightsquigarrow f'(g(t)) * \frac{d}{dt} g(t)$$

$$\left(\cos' t^2\right) * \frac{d}{dt} t^2$$

$$\frac{d}{dx} cos(x) \rightsquigarrow -\sin x$$

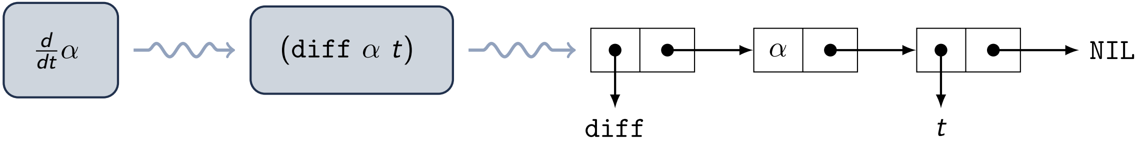
$$\left(-\sin t^2\right) * \frac{d}{dt} t^2$$

$$\frac{d}{dt} t^k \rightsquigarrow k t^{k-1}$$

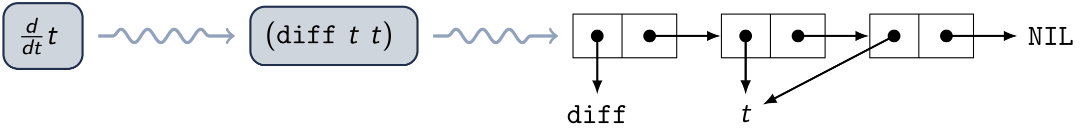
$$\left(-\sin t^2\right) * 2 * t$$

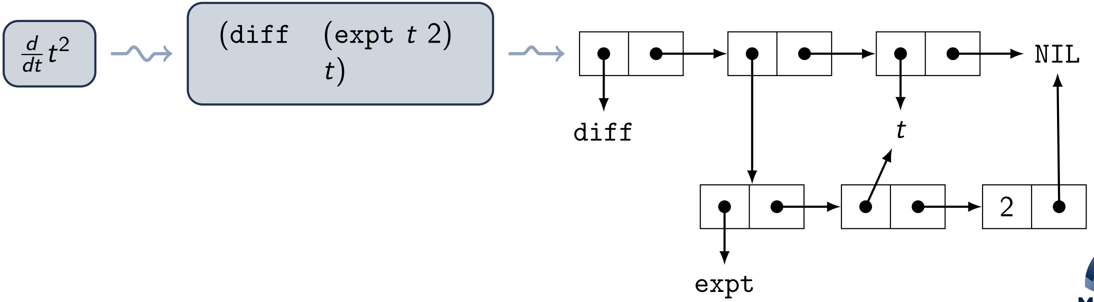
Just apply rewrite rules

Differentiation via Symbolic Expressions



Example: Differentiation S-exps





Exercise: Differentiation S-exps

$$\frac{d}{dt}\frac{\sin t}{\cos t}$$

Differential Calculus

S-expression Rewrite Rules

$$\frac{d}{dt} f(t) \rightsquigarrow (\texttt{diff}\ (f\ t)\ t)$$

Constant: $(\texttt{diff}\ k\ t) \rightsquigarrow 0$

Variable: $(\texttt{diff}\ t\ t) \rightsquigarrow 1$

Constant Power: $(\texttt{diff}\ (\texttt{expt}\ t\ k)\ t) \rightsquigarrow (*\ k\ (\texttt{expt}\ t\ (-\ k\ 1)))$

Addition: $(\texttt{diff}\ (+\ (f\ t)\ (g\ t))\ t) \rightsquigarrow (+\ (\texttt{diff}\ (f\ t)\ t)\ (\texttt{diff}\ (g\ t)\ t))$

Multiplication: $(\texttt{diff}\ (+\ (f\ t)\ (g\ t))\ t) \rightsquigarrow (+\ (*\ (\texttt{diff}\ (f\ t)\ t)\ (g\ t))\ (*\ (f\ t)\ (\texttt{diff}\ (g\ t)\ t)))$

Chain Rule: $(\texttt{diff}\ (f\ (g\ t))t) \rightsquigarrow (*\ (\texttt{deriv}\ f\ (g\ t))\ (\texttt{diff}\ (g\ t)\ t))$

Exercise: Differential Calculus

S-expression Rewrite Rules

Subtraction: $\frac{d}{dt}(f(t) - g(t)) \rightsquigarrow \frac{d}{dt}f(t) - \frac{d}{dt}g(t)$

Division: $\frac{d}{dt}\left(\frac{f(t)}{g(t)}\right) \rightsquigarrow \frac{\frac{d}{dt}f(t)}{g(t)} - \frac{f(t)*\frac{d}{dt}g(t)}{(g(t))^2}$

Derivatives of Common Functions

S-expressions

$$f'(x) \leadsto (\texttt{deriv}\ f\ x)$$

Sine:	$(\texttt{deriv sin}\ \alpha)$	$\leadsto$	$(\texttt{cos}\ \alpha)$
Cosine:	$(\texttt{deriv cos}\ \alpha)$	$\leadsto$	$(-\ (\texttt{sin}\ \alpha))$
Natural Logarithm:	$(\texttt{deriv ln}\ \alpha)$	$\leadsto$	$(/\ 1\ \alpha)$
Exponential:	$(\texttt{deriv exp}\ \alpha)$	$\leadsto$	$(\texttt{exp}\ \alpha)$

S-expression Differentiation Steps

```
(diff   (cos (expt t 2))
        t)
```

$(\mathtt{diff}\ (f\ (g\ t))) \rightsquigarrow (*\ ((\mathtt{deriv}\ f\ (g\ t)))\ (\mathtt{diff}\ (g\ t)))$

```
(*   (deriv cos (expt t 2))
     (diff (expt t 2) t))
```

$(\mathtt{deriv}\ \mathtt{cos}\ x) \rightsquigarrow (-\ (\mathtt{sin}\ x))$

```
(*   (- (sin (expt t 2)))
     (diff (expt t 2) t))
```

$(\mathtt{diff}\ (\mathtt{expt}\ t\ k)) \rightsquigarrow (*\ k\ (\mathtt{expt}\ t\ (-\ k\ 1)))$

```
(*   (- (sin (expt t 2)))
     (* 2 t)))
```

Symbolic Differentiation Function

$$\frac{d}{dx}\alpha$$

(cos (expt *t* 2))   *t*

$$\texttt{d-eval} : \overbrace{\texttt{s-exp}}^{\alpha} \times \overbrace{\texttt{symbol}}^{x} \mapsto \overbrace{\texttt{s-exp}}^{\frac{d}{dx}\alpha}$$

```
(*   (- (sin (expt t 2)))
     (* 2 t)))
```

Symbolic Differentiation Algorithm

Procedure d-eval(e,v)

if `constant?`(e) **then return** 0; // $\frac{d}{dv}k \rightsquigarrow 0$

else if $v = e$ **then return** 1; // $\frac{d}{dv}v \rightsquigarrow 1$

else

  $f \leftarrow$ `first`(e);

  if $+ = f$ **then return** `d-eval-+`(e,v); // $\frac{d}{dt}(f(t) + g(t))$

  else if $* = f$ **then return** `d-eval-*`(e,v); // $\frac{d}{dt}(f(t) * g(t))$

  else if (`expt` $= f$) $\wedge$ `constant?`(**third** (e)) **then**// $\frac{d}{dt}f^k(t) \rightsquigarrow kf^{k-1}(t)(\frac{d}{dt}f(t))$

    return `d-eval-expt`(e,v);

  ...

  else if $1 = |$`rest`$(e)|$ **then**// $\frac{d}{dt}f(g(t)) = f'(g(t))\frac{d}{dt}g(t)$

    return `d-eval-chain`(e,v);

  else **error**("Unhandled expression") ;

Symbolic Differentiation Algorithm

d-eval-+

Procedure d-eval-+(e, v)

```
/* d/dv (f(t) + g(t)) ⇝ d/dv f(t) + d/dv g(t)   */
return cons('+, map(d-eval, rest(e)))
```

Symbolic Differentiation Algorithm

d-eval-*

Procedure d-eval-*(e,v)

/* $\frac{d}{dv}(f(v) * g(v)) \rightsquigarrow \left(\frac{d}{dv} f(v)\right) * g(v) + f(v) * \left(\frac{d}{dv} g(v)\right)$ */

$a \leftarrow$ `rest`(e);

if $0 = |a|$ **then** **return** 0 ;

else if $1 = |a|$ **then** **return** `d-eval(first(`a`),`v`)` ;

else if $2 = |a|$ **then**

  $a_0 \leftarrow$ `first(`a`)` ; // $f(t)$

  $a_1 \leftarrow$ **second** (a) ; // $g(t)$

  return `(+ $\underbrace{(*\ ,(\text{d-eval}\ a_0\ v)\ ,a_1)}_{\frac{d}{dv}f(v)*g(v)}$ $\underbrace{(*\ ,a_0\ ,(\text{d-eval}\ a_1\ v))}_{\frac{d}{dv}g(v)*f(v)}$);

else // n-ary multiply: $(*\ a\ \beta_0\ \ldots\ \beta_n) \rightsquigarrow (*\ a\ (*\ \beta_0\ \ldots\ \beta_n))$

  return `d-eval-*(first(`a`), cons('*, rest(`a`)))`;

Symbolic Differentiation Algorithm

`d-eval-expt`

Procedure d-eval-expt(e,v)

/* $\frac{d}{dv} f^k(v) \leadsto k * (f(v))^{k-1} * \left(\frac{d}{dv} f(v)\right)$ */

$a_0 \leftarrow$ **second** (e);

$k \leftarrow$ **third** (e);

return $`(*\quad k \quad \overbrace{(\texttt{expt}\ a_0\ (-\ k\ 1))}^{(f(v))^{k-1}} \quad \overbrace{(\texttt{d-eval}\ a_0\ v)}^{\frac{d}{dt} f(v)})$

Symbolic Differentiation Algorithm

`d-eval-chain`

Procedure d-eval-chain(e, v)

/* $\frac{d}{dv} f(g(v)) \rightsquigarrow f'(g(v)) * \frac{d}{dv} g(v)$ */

$f \leftarrow$ `first`(e);

$a_0 \leftarrow$ **second** (e) ; // $g(v)$

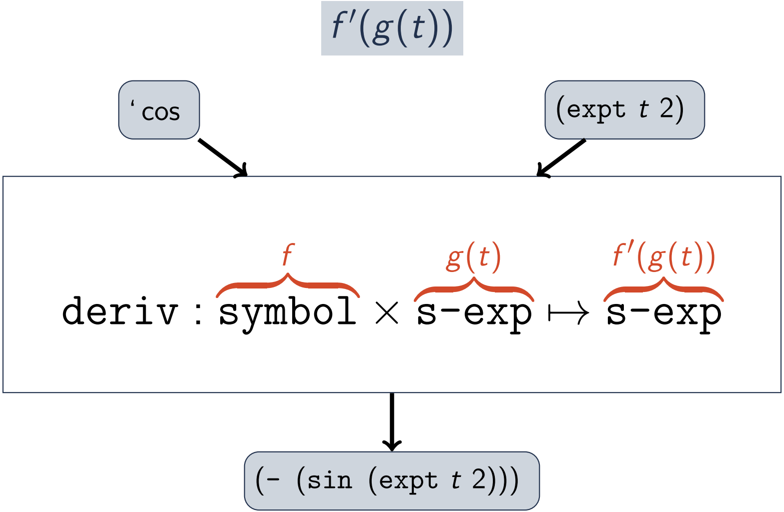
if `constant?`(a_0) **then**

  return 0;

else

  return \`$(*$,$\underbrace{(\texttt{deriv}\ f\ a_0)}_{f'(g(v))}$,$\underbrace{(\texttt{d-eval}\ a_0\ v)}_{\frac{d}{dv} g(v)}$$))$;

Deriv Function



Symbolic Differentiation Algorithm

deriv

Procedure deriv(f,a)

```
switch f do
    case 'sin do  return `(cos , a) ; // sin' a = cos a
    case 'cos do  return `(- (sin , a)) ; // cos' a = − sin a
    case 'ln do  return `(/ 1 , a) ; // ln' a = 1/a
    case 'exp do  return `(exp , a) ; // exp a = exp a
    ...
/* Else:                                                   */
error( "Unhandled function" )
```

Example 0: Symbolic Differentiation Recursion Trace

```
(d-eval '(cos (expt t 2))
        't)
```

⇝ `d-eval-chain`

```
`(* ,(deriv ,cos ,(expt t 2))
    ,(d-eval '(expt t 2) 't))
```

⇝ `(deriv cos` x`)` ⇝ `(- (sin` x`))`

```
`(* (- (sin (expt t 2)))
    ,(d-eval '(expt t 2) 't))
```

⇝ `d-eval-expt`

```
`(* (- (sin (expt t 2)))
    (* 2 (expt t 1) ,(d-eval 't 't)))
```

⇝ $\frac{d}{dt} t = 1$

```
`(* (- (sin (expt t 2)))
    (* 2 (expt t 1) 1))
```

Exercise 1: Symbolic Differentiation Recursion Trace

$\frac{d}{dt} \sin^2 t$

Exercise 2: Symbolic Differentiation Recursion Trace

$\frac{d}{dx}\left(\ln x + a * x^2\right)$

Exercise 2: Symbolic Differentiation Recursion Trace

$\frac{d}{dx}\left(\ln x + a * x^2\right)$ – continued 1

Exercise 2: Symbolic Differentiation Recursion Trace

$\frac{d}{dx}\left(\ln x + a * x^2\right)$ – continued 2

Exercise 2: Symbolic Differentiation Recursion Trace

$\frac{d}{dx}\left(\ln x + a * x^2\right)$ – continued 3

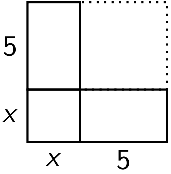
Exercise 2: Symbolic Differentiation Recursion Trace

$\frac{d}{dx}\left(\ln x + a * x^2\right)$ – continued 4

Outline

Historical Note: Algebra Notations

"The first quadrate, which is the square, and the two quandrangle sides, which are the ten roots, make together 39."



Muhammad ibn Musa al-Khwarizmi
محمد بن موسى خوارزمی
"Algoritmi"
CE 780-850

Modern Notation

$$x^2 + 10x = 39$$
$$x^2 + 10x + 25 = 39 + 25$$
$$(x+5)^2 = 64$$
$$x + 5 = 8$$
$$x = 3$$

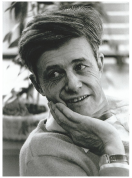
Sapir-Whorf Hypothesis

Language determines thought.

Edward Sapir

Benjamin Lee Whorf

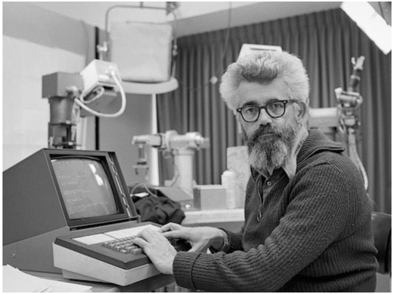
“Notation as a tool of thought.”



Kenneth Iverson

Appropriate abstractions make math/programming easier.

S-Expressions and Programming



McCarthy, John.
"Recursive Functions
of Symbolic Expressions
and Their Computation by Machine,
Part I"

"Math:"

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n * (n-1)! & \text{if } n \neq 0 \end{cases}$$

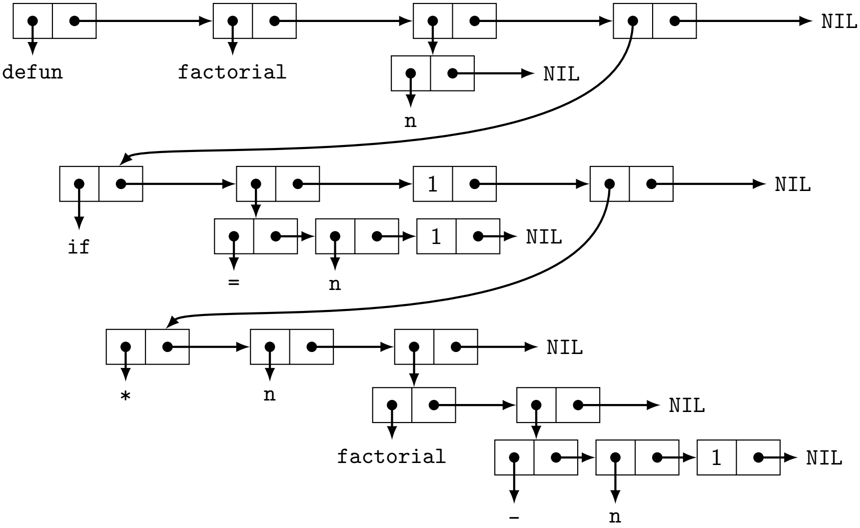
M-expression:

$$n! = (n = 0 \rightarrow 1, \quad T \rightarrow n \cdot (n-1)!)$$

S-expression:

```
(defun factorial (n)
  (if (= n 0)
      1
      (* n (factorial (- n 1)))))
```

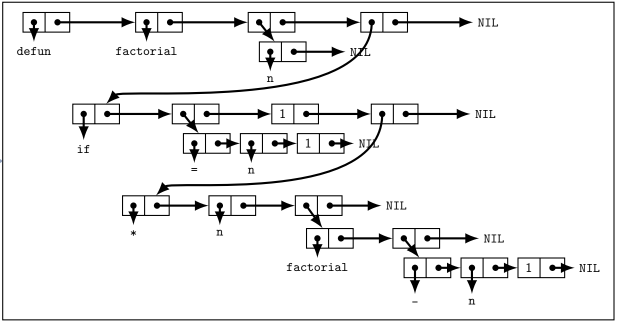
Factorial Cell Diagram



Homoiconic

Code is Data

```
(defun factorial (n)
  (if (= n 0)
      1
      (* n (factorial (- n 1)))))
```

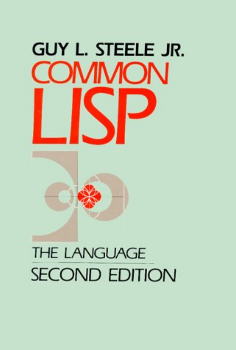


"data processing" $\Longleftrightarrow$ *"code processing"*

Lisp

“LISt Processor”

- 1960: John McCarthy. *Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I.*
- 1961: Tim Hart and Mike Levin. *The New Compiler.* MIT AI Memo 39.
- 1975: Gerald Sussman and Guy Steele, Jr. *Scheme: An Interpreter for Extended Lambda Calculus*. MIT AI Memo 349.
- 1994: ANSI Common Lisp Standard



Common Lisp Implementations

Use SBCL!

Name	Compiler	License	URL
Steel Bank Common Lisp	Good	Public Domain	`http://sbcl.org/`
Clozure Common Lisp	Fair	Apache	`https://ccl.clozure.com/`
Embeddable Common Lisp	Fair	LGPL	`https://common-lisp.net/project/ecl/`
CLISP	Bytecode	GPL	`http://clisp.org/`
LispWorks	Good	Commercial	`http://www.lispworks.com/`
Allegro Common Lisp	Good	Commercial	`https://franz.com`

Summary