Symbolic Reasoning

Sihui Li

CSCI-534, Colorado School of Mines

Spring 2022

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

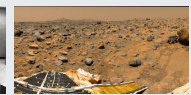
Algebra, Calculus



Computer Algebra



Symbolic Planning

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:
  - $a_0 x + a_1 x^2 + a_3 x^3$
  - $3x + 1 = 10$
- Propositional Logic:
  - $(p_1 \vee p_2) \wedge p_3$
  - $(p_1 \wedge p_2) \implies p_3$

Example (Reductions)

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

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

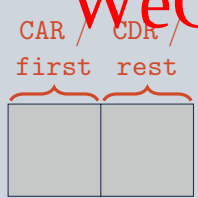
```
struct cons {
    void *first;
    struct cons *rest;
};
```

Example

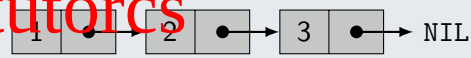
S-Expression

(1 2 3)

Diagram



Cons Cell Diagram



Abstract Syntax

Infix

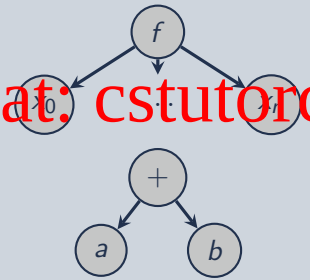
- Function / Operator
- Arguments / Operands

$f(x_0, \ldots, x_n)$

$a + b$

Abstract Syntax Tree

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



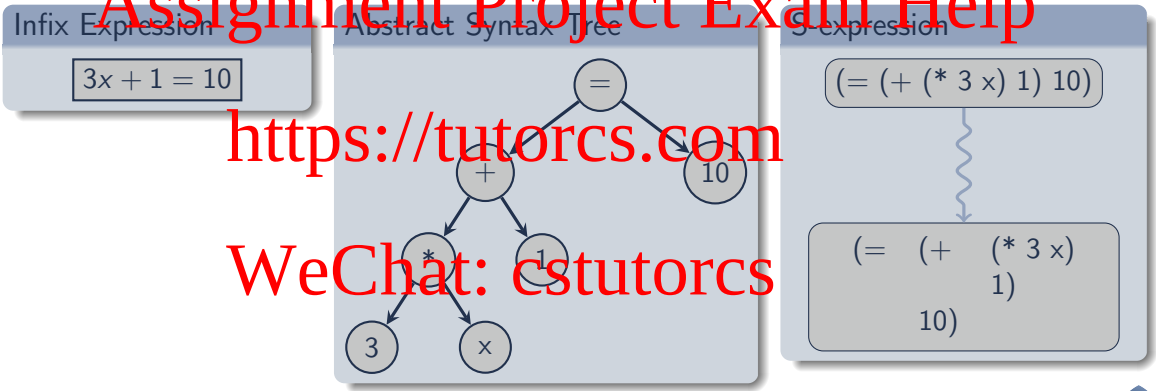
S-Expression

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

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

$(+\ a\ b)$

Example: S-Expression

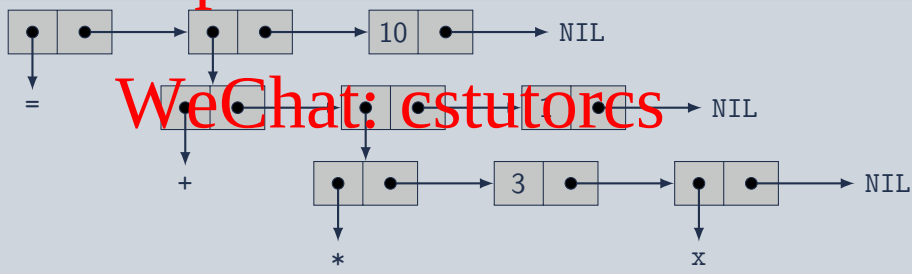


Example: Cell Diagram

S-Expression

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


Cell Diagram



List vs. Tree

List

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

Tree

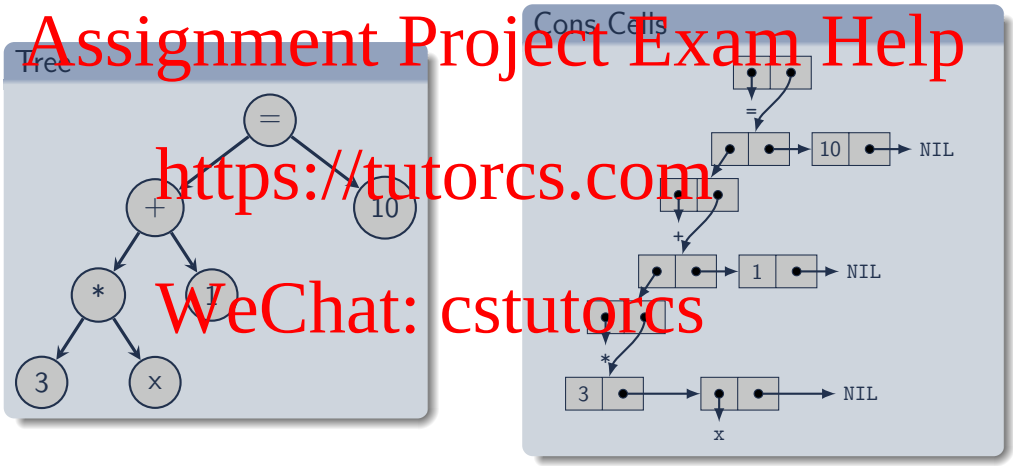
```
struct treenode {
    void *first;
    struct cons *children;
};

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


Same structure

Data Structure, Redux

$3x + 1 = 10$

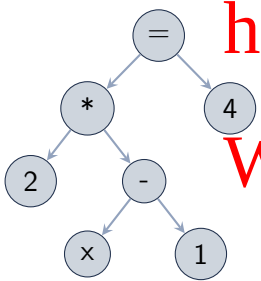


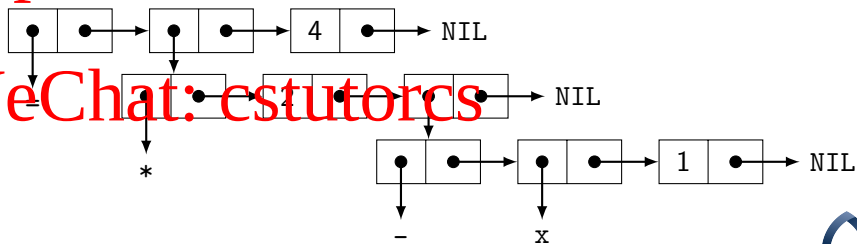
Exercise 1: S-Expression

$2(x\text{-}1) = 4$

$2(x - 1) = 4$

```
(= (* 2
      (- x 1))
   4)
```

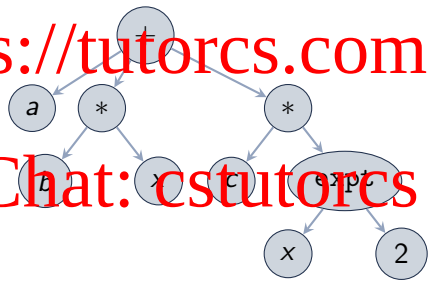





Exercise 2: S-Expression

$a + bx + cx^2$

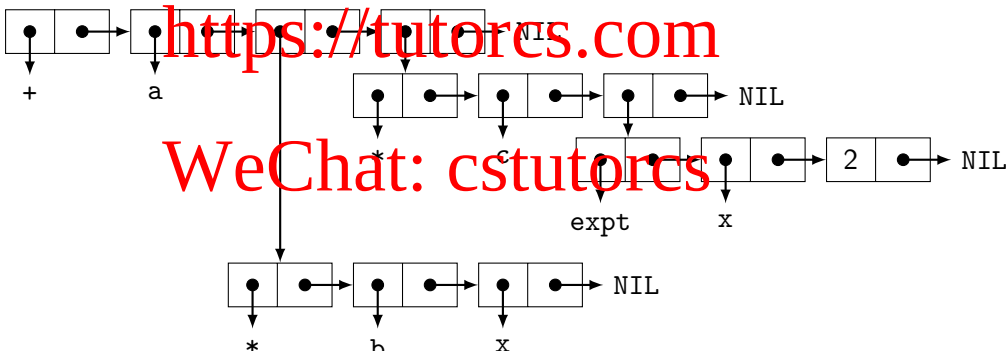
$a + bx + cx^2$



Exercise 2: S-Expression

$a + bx + cx^2$ – continued

```
(+ a
   (* b x)
   (* c (expt x 2)))
```

Rewrites

Evaluation Function

Recursive Evaluation Algorithm

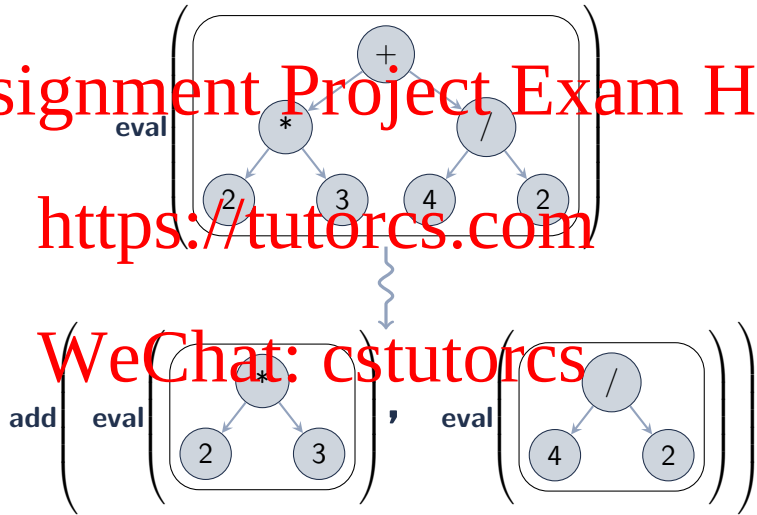
Base Case: If argument is a value: return the value

Recursive Case: Else (argument is an expression):

1. Recurse on arguments
2. Apply operator to results

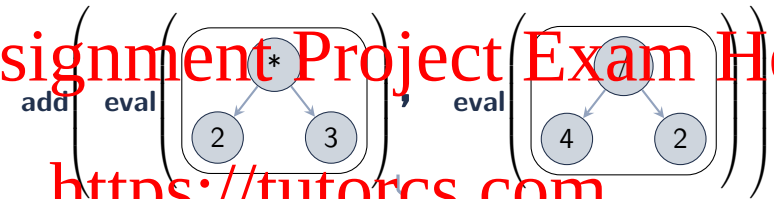
Example: Evaluation

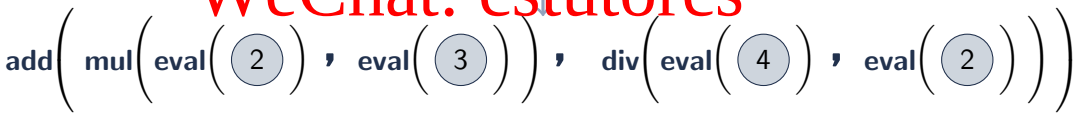
2*3 + 4/2



Example: Evaluation

2*3 + 4/2 – continued



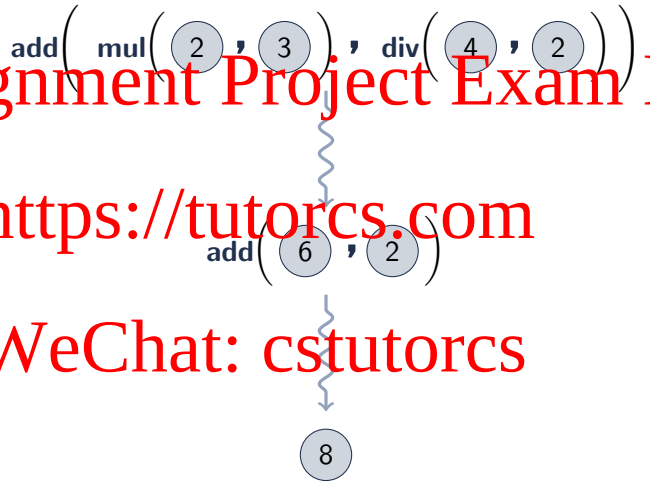


Example: Evaluation

2*3 + 4/2 – continued

Example: Evaluation

2*3 + 4/2 – continued



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

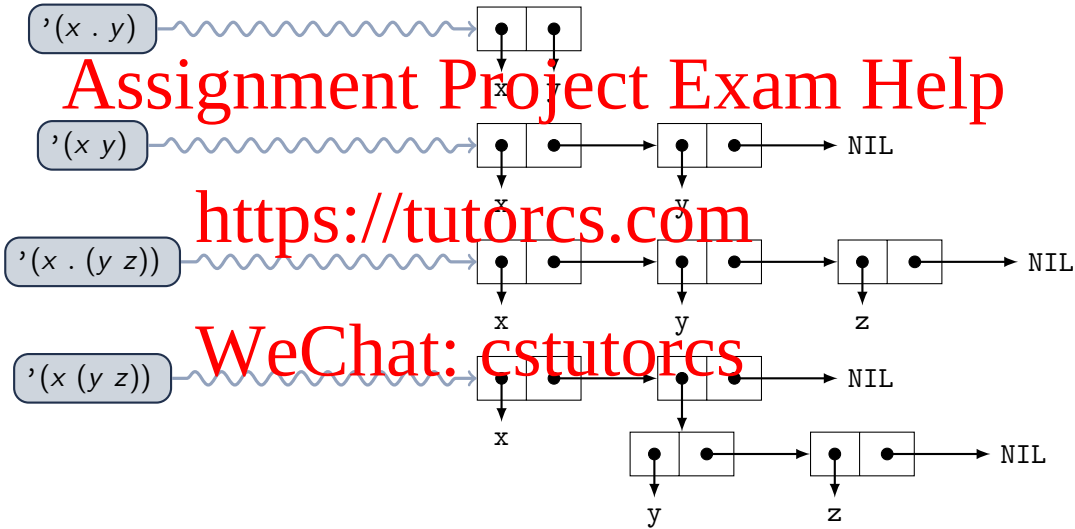
Returns the quoted s-expression:

'(`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

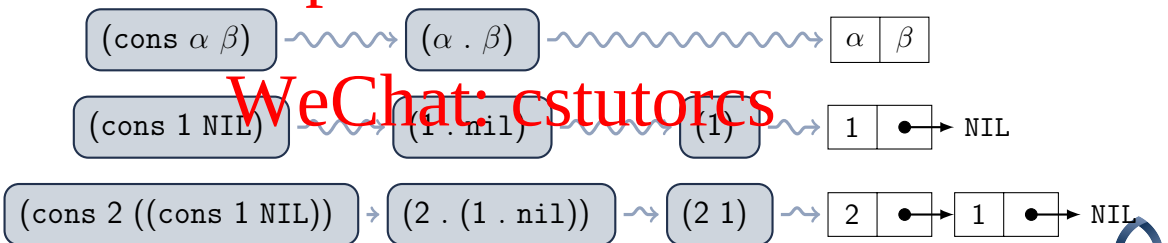
Creating Lists

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)



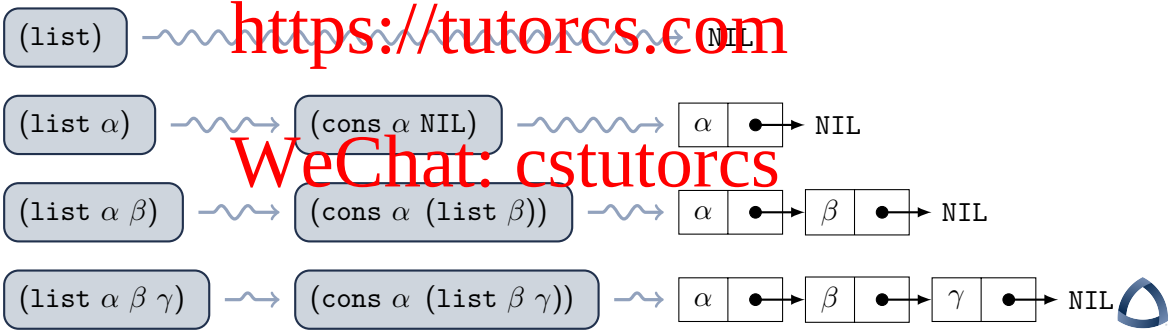
List Function

LIST

Return a list containing the supplied objects:

(list $a_0 \ldots a_n$)

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



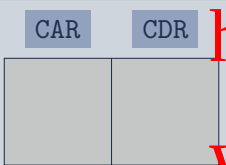
Exercise: List Construction

- (cons 'x 'y) $\rightsquigarrow$ (x . y)
- (cons 'x '(y z)) $\rightsquigarrow$ (x y z)
- (cons 'x (list y z)) $\rightsquigarrow$ (x y z)
- (list (+ 1 2 3)) $\rightsquigarrow$ (6)
- (list '(+ 1 2 3)) $\rightsquigarrow$ ((+ 1 2 3))
- (list '* (+ 2 2) '(- 2 2)) $\rightsquigarrow$ (list '* 4 '(- 2 2)) $\rightsquigarrow$ (* 4 (- 2 2))

List Access

CAR / CDR

Cons cell



CAR

Return the car of a cons cell:

`(car cell)`

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

`(car '(`α . β`))` $\rightsquigarrow$ α

CDR

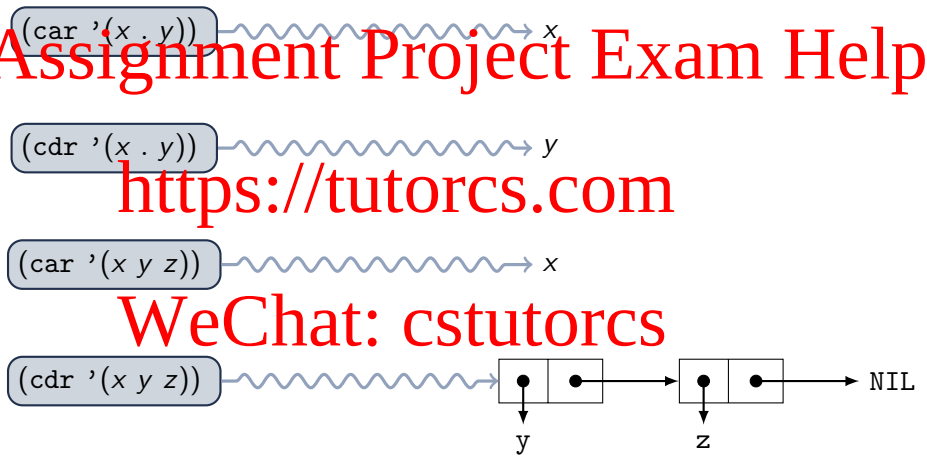
Return the cdr of a cons cell:

`(cdr cell)`

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

`(cdr '(`α . β`))` $\rightsquigarrow$ β

Example: CAR / CDR



List Template Syntax

Backquote (‘): Create a template

- ‘(x y) ⇝ (list ’x ’y) = ‘(x y)
- ‘(+ a (* b c)) ⇝ (list ’+ ’a (list ’* ’b ’c)) ⇝ (+ a (* b c))

Comma (,)

Evaluate and **insert**:

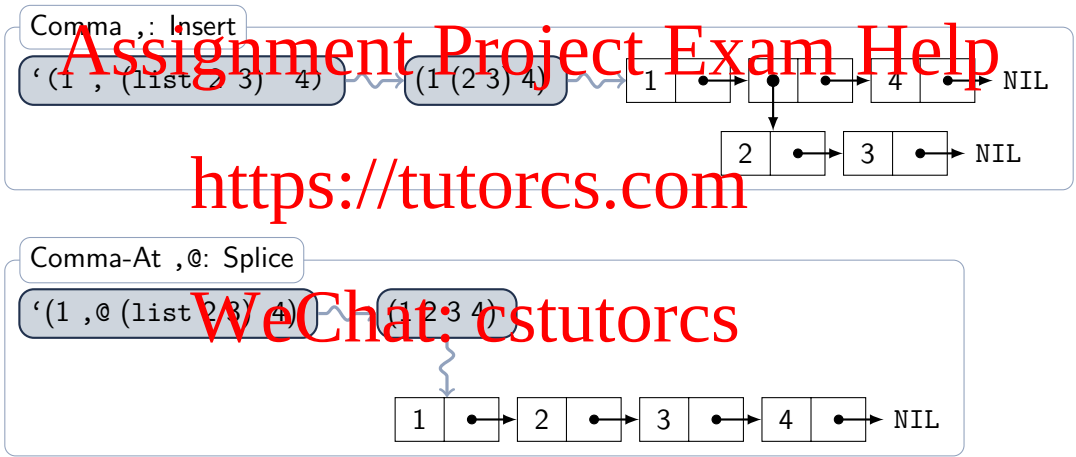
- ‘(α . . . ,y β . . .) ⇝ (list α . . . y β . . .)
  (y: evaluated)
- ‘(+ a ,(* 2 3))
  ⇝ (list ’+ ’a (* 2 3))
  ⇝ (+ a 6)

Comma-At (, @)

Evaluate and **splice**:

- ‘(α . . . ,@y β . . .) ⇝ (append α . . . y β . . .)
  (y: splice)
- ‘(+ a ,@ (list (* 2 3) (* 4 5)))
  ⇝ (append ’(+ a) (list (* 2 3) (* 4 5)))
  ⇝ (+ a 6 20)

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



Exercise: List Template Syntax

- `‘(1 2 ,(+ 3 4))` ⇝ `(1 2 7)`

- `‘(,1 ,2 (+ 3 4))` ⇝ `(1 2 (+ 3 4))`

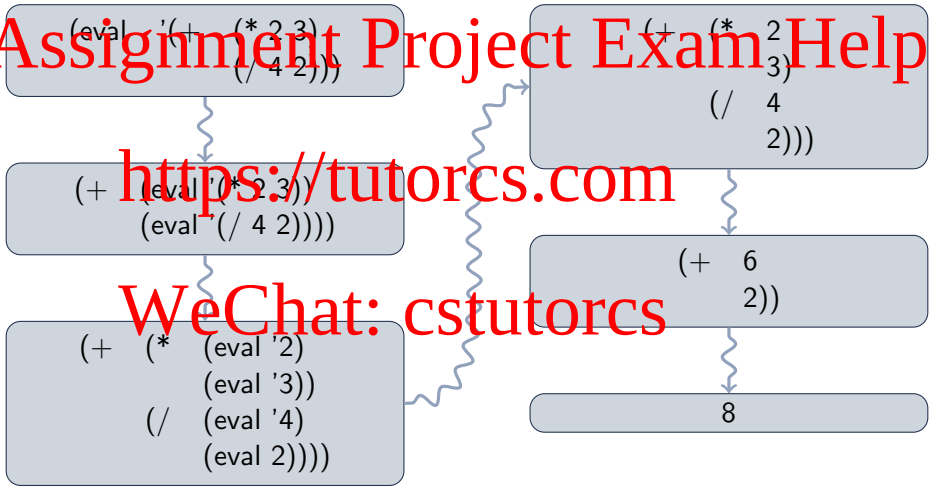
- `‘(+ 1 ,2 ,(+ 3 4))` ⇝ `(+ 1 2 7)`

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

Outline

Example: Evaluation via S-Expressions

2*3 + 4/2



Evaluation Algorithm

Procedure eval(e)

```
1  if value?(e) then /* Argument is a value                    */
2      return e;
3  else /* Argument is an expression                           */
4      operator ← first(e) ;
5      arg-sexp ← rest(e) ;
6      arg-vals ← map (eval, arg-sexp);
7      switch operator do
8          case '+ do f ← +;
9          case '- do f ← −;
10         case '/ do f ← /;
11         case '* do f ← ∗;
12     return apply(f, arg-vals);
```

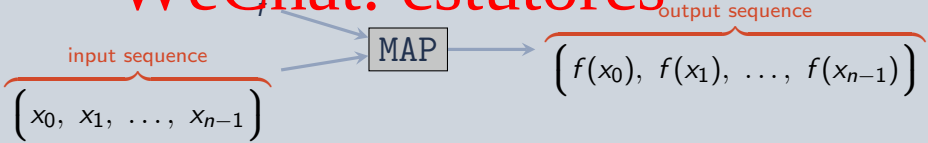
Map function

Definition (map)

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

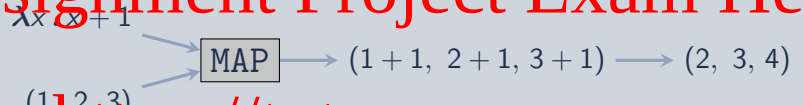
$$\texttt{map} \quad : \quad \underbrace{(\mathbb{X} \mapsto \mathbb{Y})}_{\text{function}} \quad \times \quad \underbrace{\mathbb{X}^n}_{\text{input sequence}} \quad \mapsto \quad \underbrace{\mathbb{Y}^n}_{\text{output sequence}}$$

Illustration

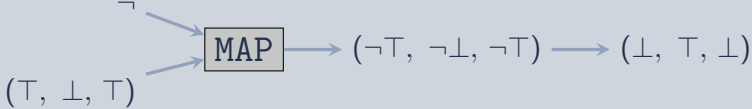


Example: Map

+1



¬



Algorithm: Map function

Functional Map

Procedure map(f,s)

```
1 if empty(s) then /* s is empty                    */
2 |   return NIL
3 else /* s has members                             */
4 |   return
  |     cons(f(first(s)), map(f, rest(s)));
```

Imperative Map

Procedure map(f,s)

```
1 n ← length(s);
2 Y ← make-sequence(n);
3 i ← 0;
4 while i < n do
5 |   Y[i] ← f(s[i]);
6 |   i ← i + 1;
7 return Y;
```

Example: Map

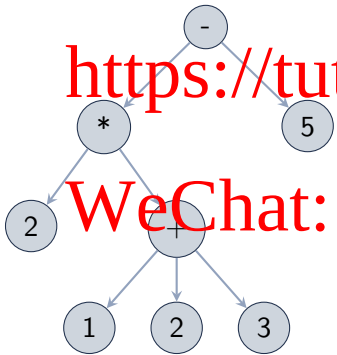
Example (Illustration)

Example (Lisp)

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

Exercise 1: Evaluation

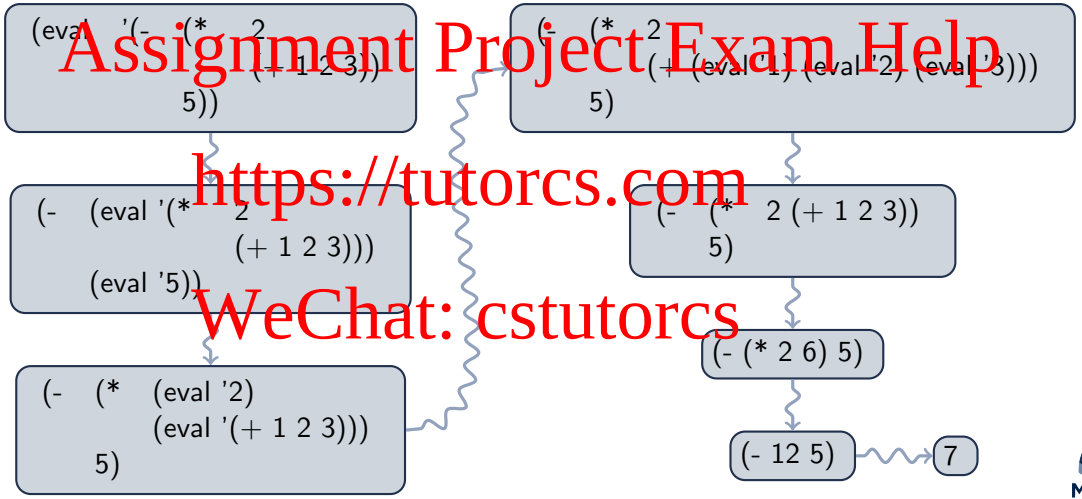
2*(1+2+3)- 5




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

Exercise 1: Evaluation

continued



Example: Partial Evaluation

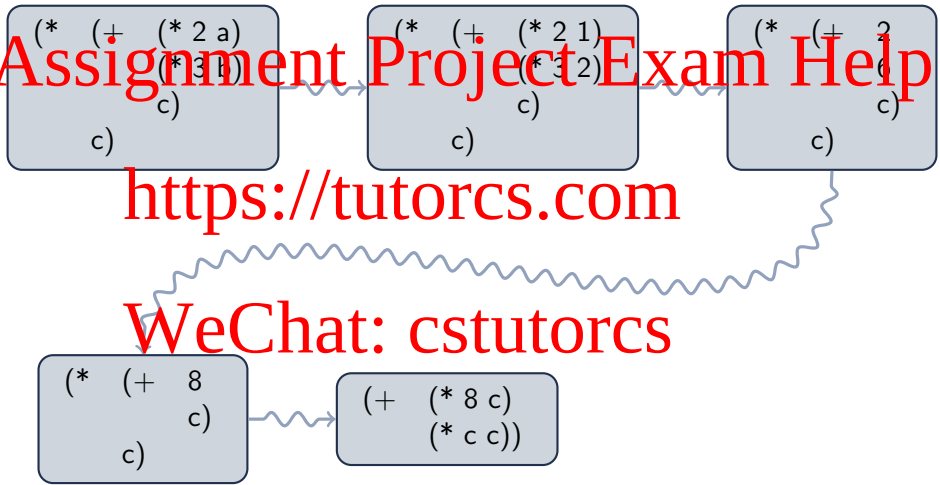
Given:
- $f(x_0, x_1, x_2) = x_2(2x_0 + 3x_1 + x_2)$
- $a = 1$
- $b = 2$

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

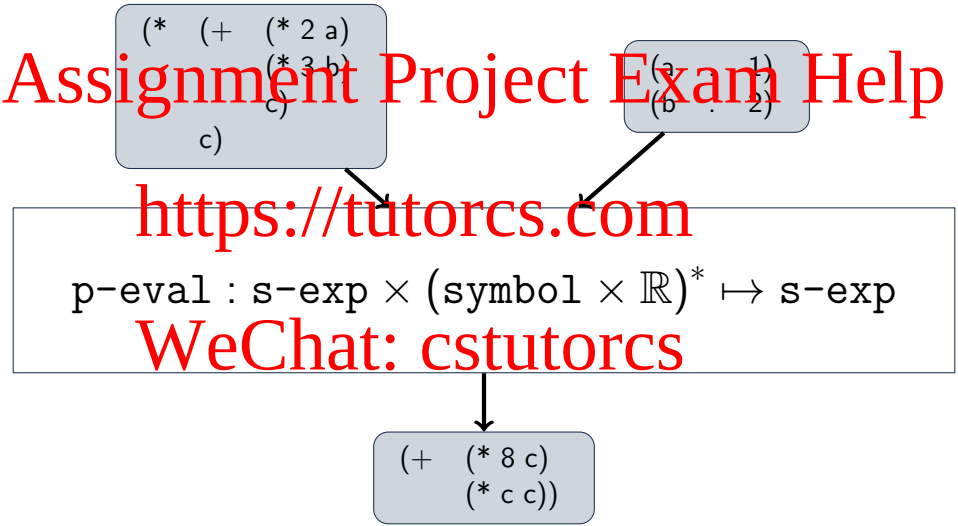
Solution:

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

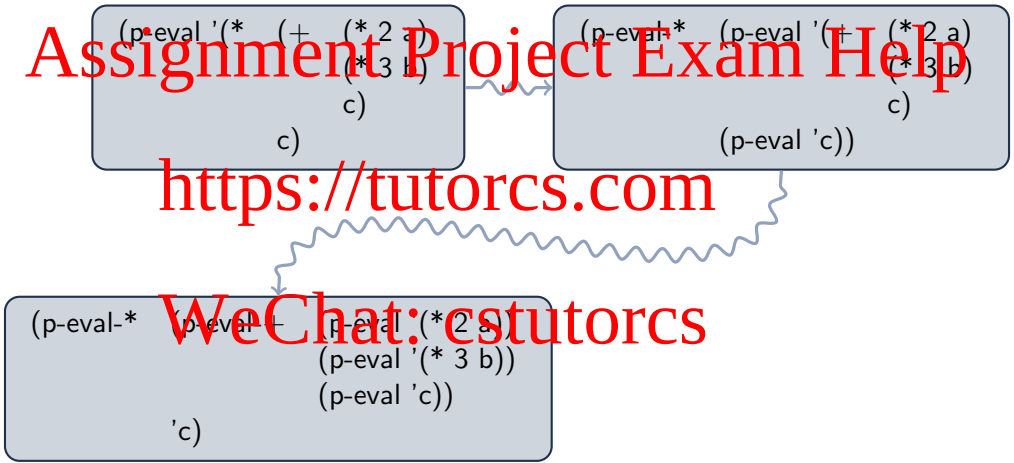
Partial Evaluation via S-Expressions



Partial Evaluation Function

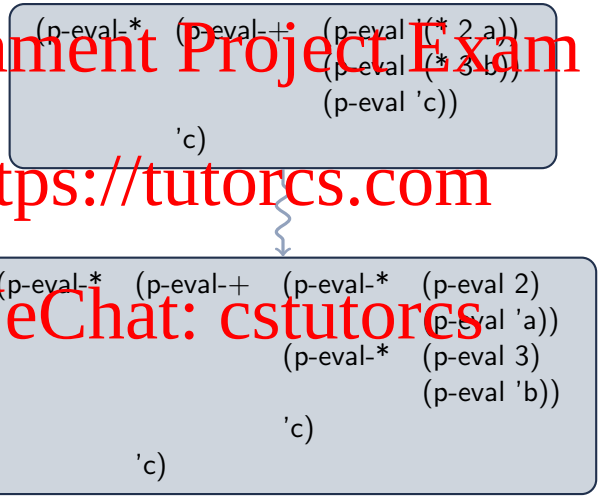


Recursive Partial Evaluation



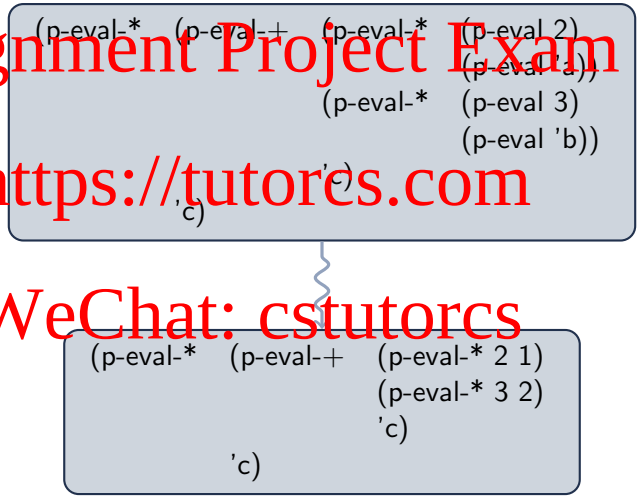
Recursive Partial Evaluation

continued



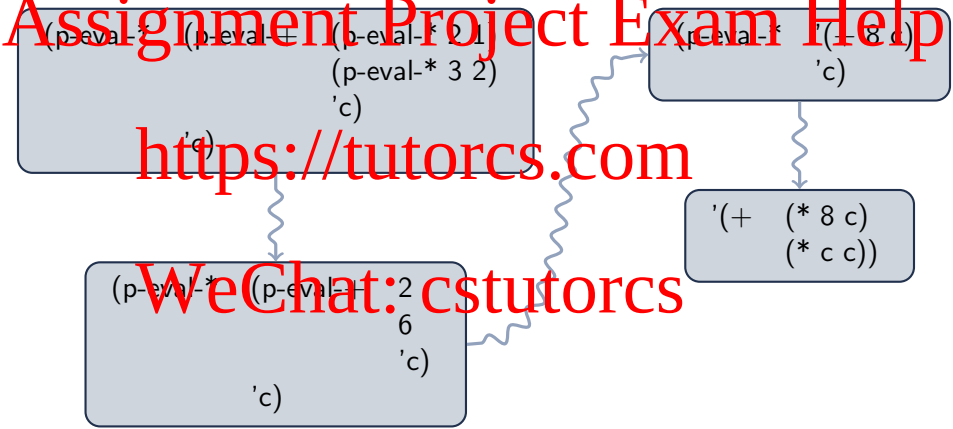
Recursive Partial Evaluation

continued



Recursive Partial Evaluation

continued



Algorithm: Partial Evaluation

Procedure p-eval(e,bindings)

```
1  if number?(e) then
2      return e;
3  else if symbol?(e) then
4      if bindings[e] then return bindings[e] ;
5      else return e;
6  else
7      y ← map(p-eval, rest(e));
8      switch first(e) do
9          case '+ do f ← p-eval-+;
10         case '* do f ← p-eval-*;
11         ...
12     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, e)

```
1  N ← {e ∈ E | number?(e)};
2  n ← fold-left(+, 0, N);
3  S ← {e ∈ E | ¬number?(e)};
4  if 0 ≠ n then
5      if ∅ = S then return 0;
6      else if 1 = |S| then return
         first(S);
7      else return cons('+, S);
8  else
9      if ∅ = S then return n;
10     else return
         cons('+, cons(n, S));
```

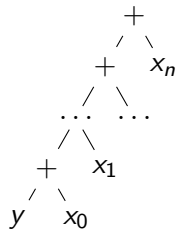
Fold-Left

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)

1 $i \leftarrow 0$;
2 **while** $i < |X|$ **do**
3 $\quad y \leftarrow f(y, X_i)$;
4 **return** y;

Recursive

Function fold-left(f, y, X)

1 **if** `empty?`(X) **then return** y ; `/* Base Case */`
2 **else** `/* Recursive Case */`
3 $\quad y' \leftarrow f(y,$ `first`$(X))$;
4 $\quad$ **return** fold-left$(f, y',$ `rest`$(X))$;

Exercise: Partial Evaluation

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

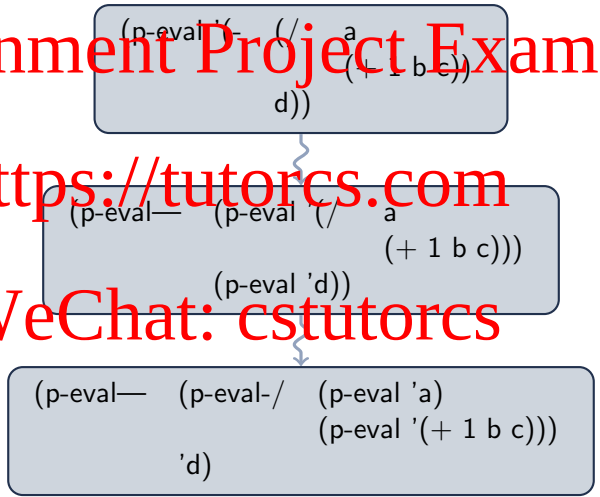
Find: Recursively simplify e

Solution:

```
(p-eval '(- (/ a
               (+ 1 b c))
            d))
```

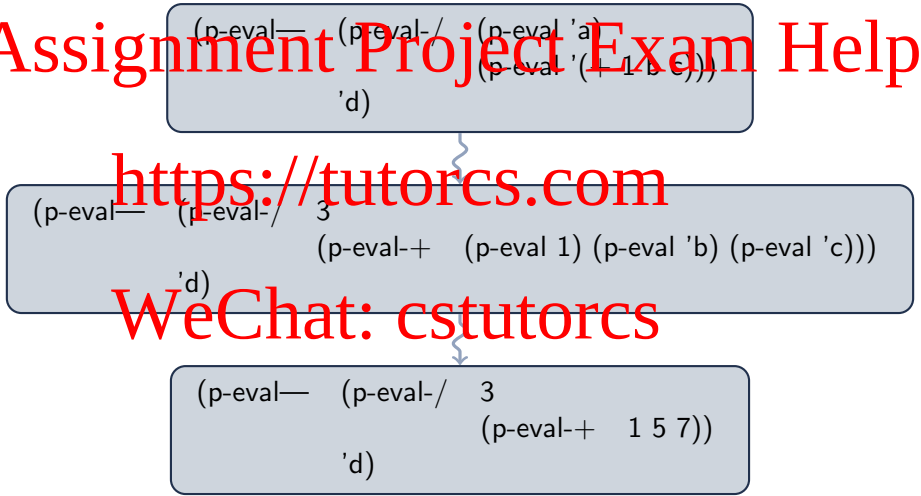
Exercise: Partial Evaluation

continued – 1



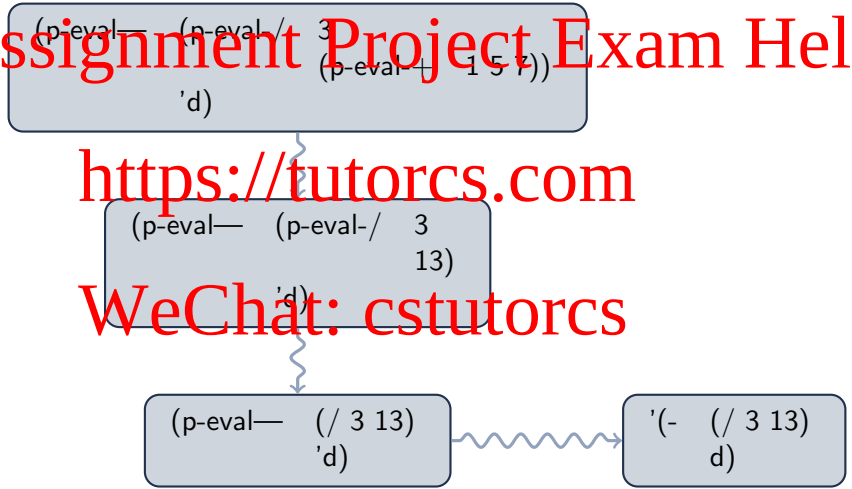
Exercise: Partial Evaluation

continued – 2



Exercise: Partial Evaluation

continued – continued 3



Derivative

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

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

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

Differential Calculus

Rewrite Rules

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

Derivatives of Common Functions

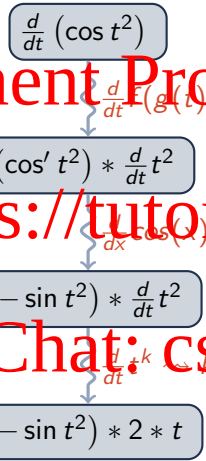
Sine: $\sin' x \rightsquigarrow \cos x$

Cosine: $\cos' x \rightsquigarrow -\sin x$

Natural Logarithm: $\ln' x \rightsquigarrow \frac{1}{x}$

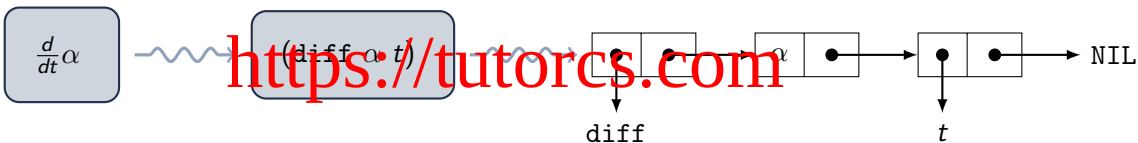
Exponential: $\exp' x \rightsquigarrow \exp x$

Differentiation Steps

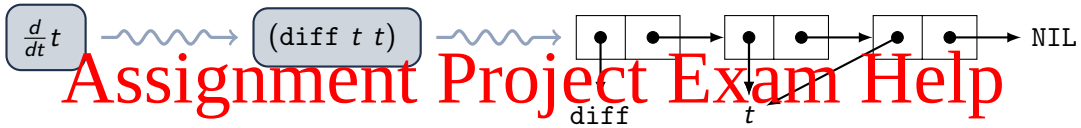


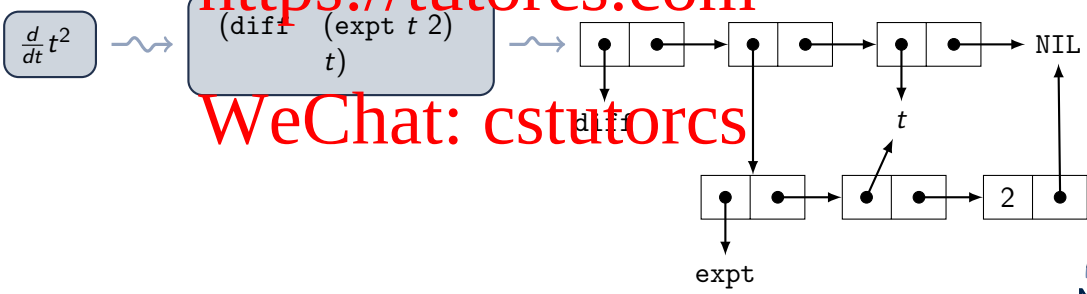
Just apply rewrite rules

Differentiation via Symbolic Expressions

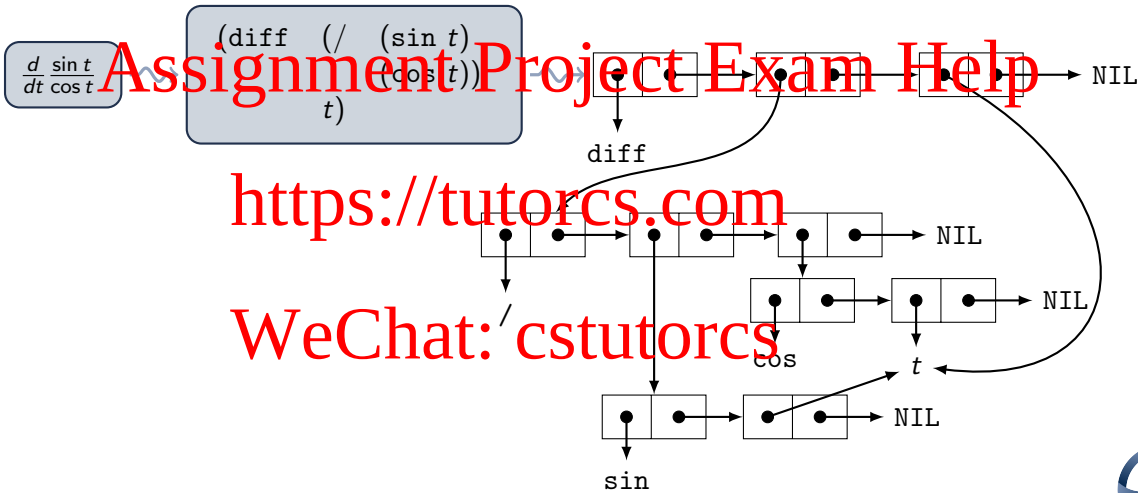


Example: Differentiation S-exps





Exercise: Differentiation S-exps



Differential Calculus

S-expression Rewrite Rules

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

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

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

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

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

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

$$\begin{array}{l}\texttt{(+ (* (diff (} f\ t\texttt{) } t\texttt{) (} g\ t\texttt{))}\\ \texttt{\quad (* (} f\ t\texttt{) (diff (} g\ t\texttt{) } t\texttt{)))}\end{array}$$

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

$$\begin{array}{l}\texttt{(* (deriv } f\ \texttt{(} g\ t\texttt{))}\\ \texttt{\quad (diff (} g\ t\texttt{) } t\texttt{))}\end{array}$$

Exercise: Differential Calculus

S-expression Rewrite Rules

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

$$(\texttt{diff}\ (-\ (f\ t)\ (g\ t))\ t) \rightsquigarrow (-\ (\texttt{diff}\ (f\ t)\ t)\ (\texttt{diff}\ (g\ t)\ t))$$

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


```
(diff (/ (f t) (g t)) t)  ⇝  (- (/ (diff (f t) t)
                                   (g t))
                                (/ (* (f t) (diff (g t) t))
                                   (expt (g t) 2)))
```

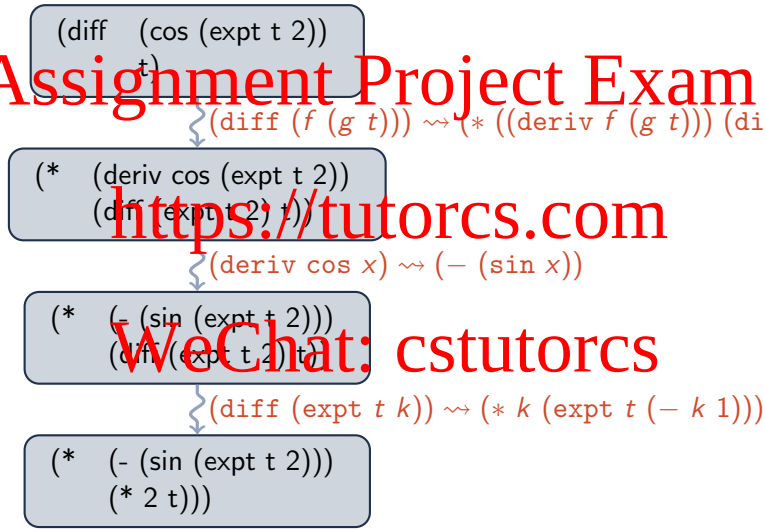
Derivatives of Common Functions

S-expressions

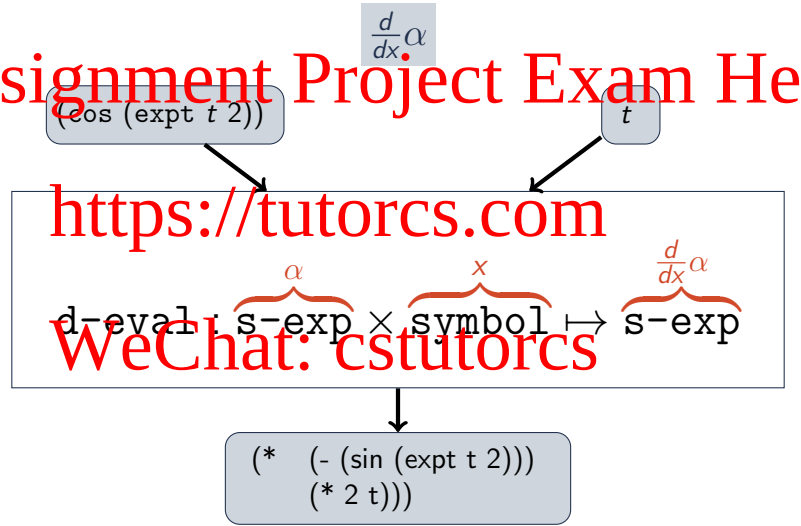
$$f'(x) \rightsquigarrow (\texttt{deriv}\ f\ x)$$

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

S-expression Differentiation Steps



Symbolic Differentiation Function



Symbolic Differentiation Algorithm

Procedure d-eval(e,v)

```
1  if constant?(e) then return 0; // d/dv k ⇝ 0
2  else if v = e then return 1; // d/dv v ⇝ 1
3  else
4      f ← first(e);
5      if + = f then return d-eval-+(e,v); // d/dt (f(t) + g(t))
6      else if ∗ = f then return d-eval-∗(e,v); // d/dt (f(t) ∗ g(t))
7      else if (expt = f) ∧ constant?(third(e)) then// d/dt f^k(t) ⇝ k f^{k−1}(t)(d/dt f(t))
8          return d-eval-expt(e,v);
9      ...
10     else if 1 = |rest(e)| then// d/dt f(g(t)) = f′(g(t)) d/dt g(t)
11         return d-eval-chain(e,v);
12     else error("Unhandled expression");
```

Symbolic Differentiation Algorithm

`d-eval-+`

Procedure d-eval-+(e,v)

/* $\frac{d}{dv}(f(t)+g(t)) \rightsquigarrow \frac{d}{dv}f(t)+\frac{d}{dv}g(t)$ */

1 **return** `cons`$('+,$ **map**$($`d-eval`$,$`rest`$(e)))$

Symbolic Differentiation Algorithm

`d-eval-*`

Procedure d-eval-*(e,v)

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

```
a ← rest(e);
if 0 = |a| then return 0 ;
else if 1 = |a| then return d-eval(first(a),v) ;
else if 2 = |a| then
    a0 ← first(a) ; // f(t)
    a1 ← second(a) ; // g(t)
    return ‘(+ (* ,(d-eval a0 v) ,a1) (* ,a0 ,(d-eval a1 v)));
else // n-ary multiply: (* a β0 ... βn) ⇝ (* a (* β0 ... βn))
    return d-eval-*(first(a), cons(′*, rest(a)));
```

(In the return on the `2 = |a|` branch, the first product is annotated $\frac{d}{dv} f(v) * g(v)$ and the second $\frac{d}{dv} g(v) * f(v)$.)

Symbolic Differentiation Algorithm

`d-eval-expt`

Procedure d-eval-expt(e, v)

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

1 $a_0 \leftarrow$ **second** (e);

2 $k \leftarrow$ **third** (e);

3 **return** `'(* k (expt a0 (- k 1)) (d-eval a v))`

Symbolic Differentiation Algorithm

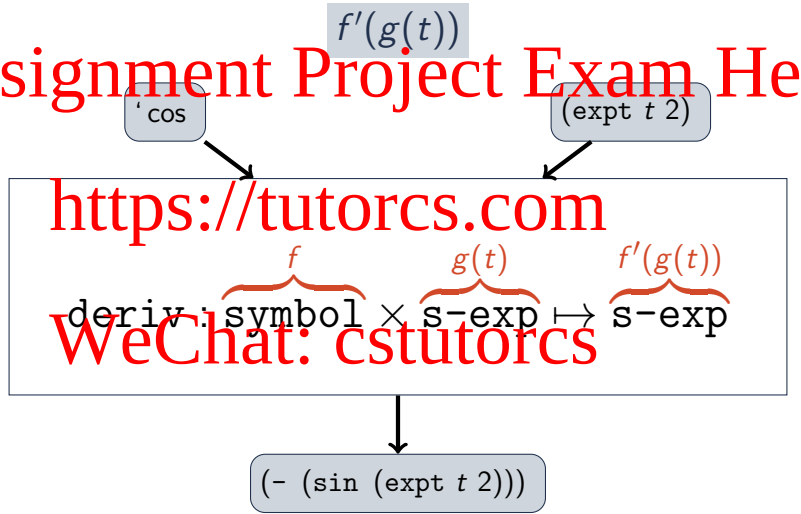
`d-eval-chain`

Procedure d-eval-chain(e,v)

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

1 $f \leftarrow$ `first`(e);
2 $a_0 \leftarrow$ **second**(e); {f, g, v}
3 **if** `constant?`(a_0) **then**
4 | **return** 0;
5 **else**
| $f'(g(v))$ $\frac{d}{dv} g(v)$
6 | **return** '(* , (deriv f a_0) , (d-eval a_0 v));

Deriv Function



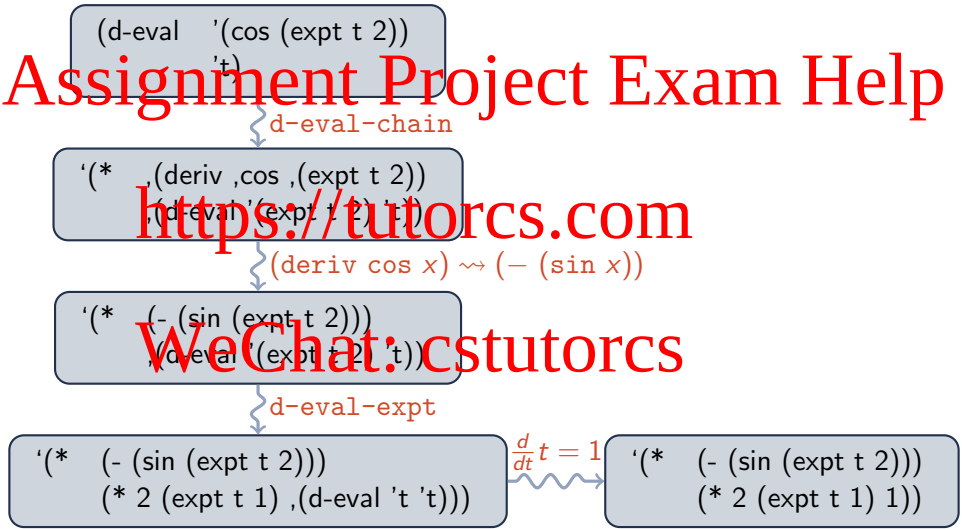
Symbolic Differentiation Algorithm

`deriv`

Procedure deriv(f,a)

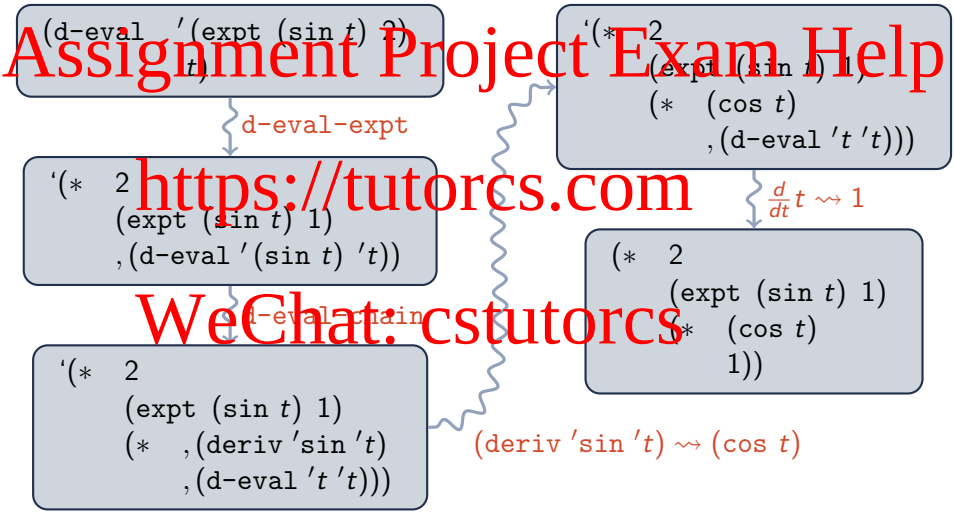
```
1 switch f do
2     case 'sin do return '(cos , a) ; // sin' a = cos a
3     case 'cos do return '(- ('sin , a)) ; // cos' a = - sin a
4     case 'ln do return '(/ 1 , a) ; // ln' a = 1/a
5     case 'exp do return '(exp , a) ; // exp a = exp a
6     ...
  /* Else:                                                    */
7 error("Unhandled function")
```

Example 0: Symbolic Differentiation Recursion Trace



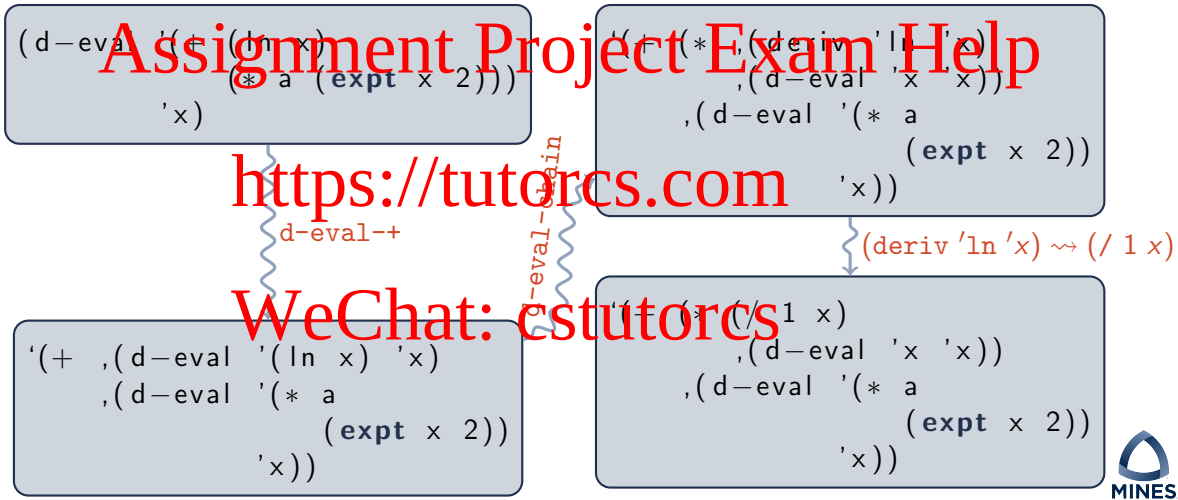
Exercise 1: Symbolic Differentiation Recursion Trace

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



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

```
'(+ (/ 1 x)
    ,(d-eval 'x 'x))
   ,(d-eval '(* a
                (expt x 2))
            'x))
```

$\frac{d}{dx}x \rightsquigarrow 1$

```
'(+ (* (/ 1 x)
       1)
    ,(d-eval '(* a
                 (expt x 2))
             'x))
```

d-eval-*

```
(+ (* (/ 1 x)
      1)
   (+ (* ,(d-eval 'a 'x)
         (expt x 2))
      (* a
         ,(d-eval '(expt x 2)
                  'x))))
```

Exercise 2: Symbolic Differentiation Recursion Trace

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

$\frac{d}{dx} K \rightsquigarrow 0$

```
'(+ (* (/ 1 x)
       1)
    (+ (* ,(d-eval 'a 'x)
          (expt x 2))
       (* a
          ,(d-eval '(expt x 2)
                   'x))))
```

```
(+ (* (/ 1 x)
      1)
   (+ (* 0 (expt x 2))
      (* a
         ,(d-eval '(expt x 2)
                  'x))))
```

Exercise 2: Symbolic Differentiation Recursion Trace

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

d-eval-expt

```
'(+ (* (/ 1 x)
       1)
    (+ (* 0 (expt x 2))
       (* a
          (* 2
             (expt x 1)
             ,(d-eval ,x
                      ,x)))))
```

Exercise 2: Symbolic Differentiation Recursion Trace

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

$\frac{d}{dx} x \rightsquigarrow 1$

```
'(+ (* (/ 1 x)
       1)
    (+ (* 0 (expt x 2))
       (* a
          (* 2
             (expt x 1)
             ,(d-eval ,x
                      ,x)))))
```

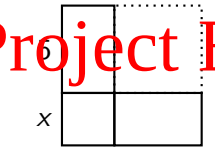


```
'(+ (* (/ 1 x)
       1)
    (+ (* 0 (expt x 2))
       (* a
          (* 2
             (expt x 1)
             1))))
```

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."

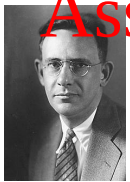


Muhammad ibn Musa al-Khwarizmi
محمد بن موسى الخوارزمي
"Algoritmi"
CE 780-850

Modern Notation

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

Sapir-Whorf Hypothesis



Edward Sapir

"Language determines thought."

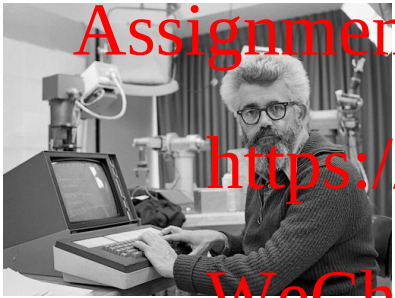
Benjamin Lee Whorf

"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:"

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

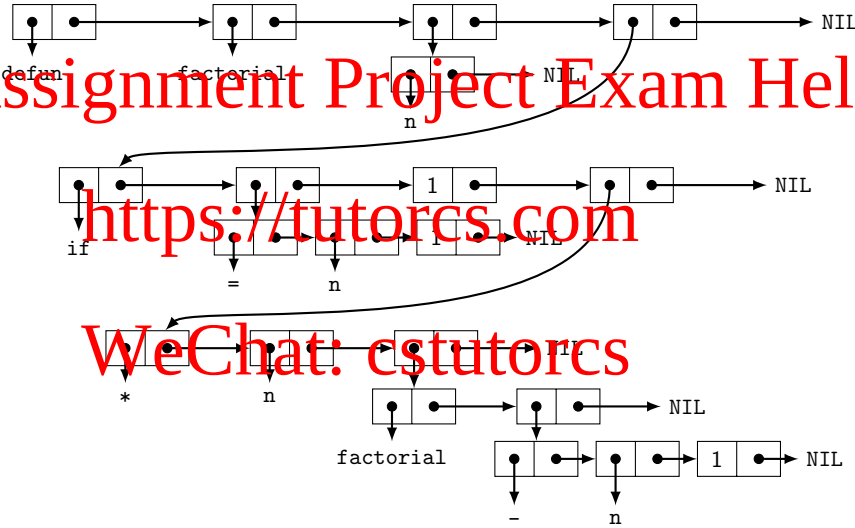
M-expression:

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

S-expression:

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

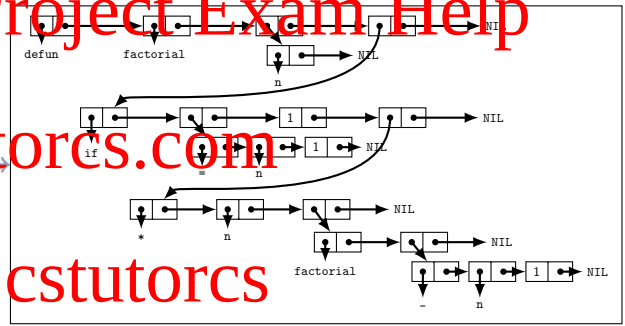
Factorial Cell Diagram



Homoiconic

Code is Data


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



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

Lisp

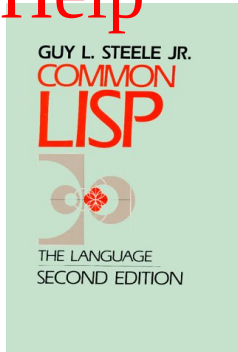
"LISt Processor"

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