Symbolic Reasoning

Dr. Neil T. Dantam

CSCI-498/598 RPM, Colorado School of Mines

Spring 2018

Outline

Rewrite Systems

Expressions

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

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:
  - $\big(\neg(\alpha \wedge \beta)\big) \rightsquigarrow \big((\neg\alpha \vee \neg\beta)\big)$
  - $\big(\neg(\alpha \vee \beta)\big) \rightsquigarrow \big((\neg\alpha \wedge \neg\beta)\big)$
- etc.

Progressively apply reductions until reaching desired expression.

Example: Algebra

Given: $3x + 1 = 10$

Find: x

Solution:

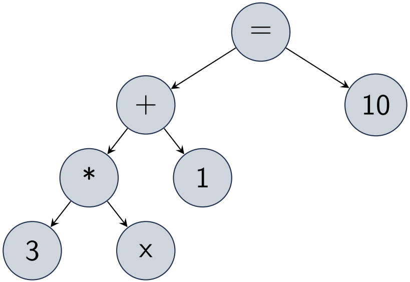
$$
\begin{array}{r|rcl}
\text{Initial} & 3x + 1 & = & 10 \\
-1 & 3x + 1 - 1 & = & 10 - 1 \\
\text{Simplify} & 3x & = & 9 \\
/3 & 3x/3 & = & 9/3 \\
\text{Simplify} & x & = & 3
\end{array}
$$

Outline

S-Expression

$3x + 1 = 10$

Abstract Syntax Tree



S-expression

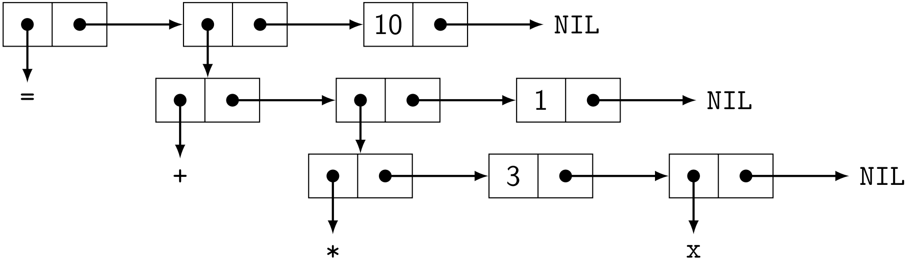
```
(= (+ (* 3 x) 1) 10)
```

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

Cell Diagram

$3x + 1 = 10$

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



List vs. Tree

List

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

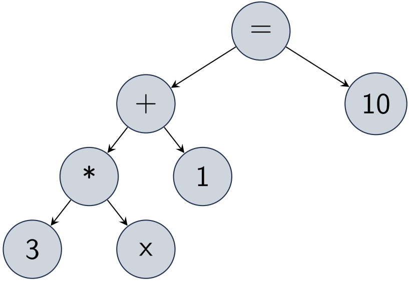
Tree

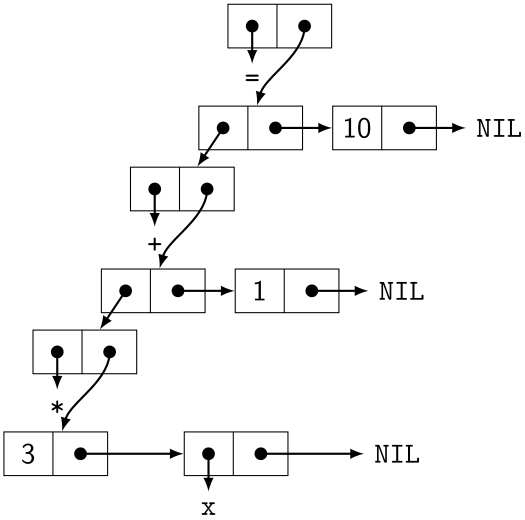
```
struct treenode {
    void *first;
    struct cons *children;
};

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

Data Structure, Redux

$3x + 1 = 10$





Exercise 1: S-Expression

2(x+1) = 4

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

Exercise 2: S-Expression

$a + bx + cx^2$

$$a + bx + cx^2$$

Example 2: S-Expression

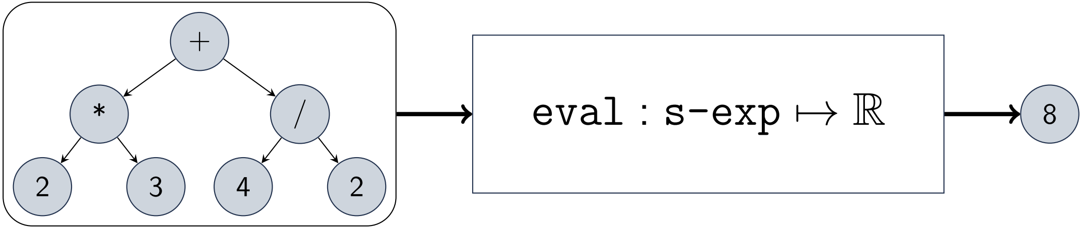
$a + bx + cx^2$ – continued

Outline

Rewrites

$$3x + 1 = 10 \overset{-1}{\rightsquigarrow} 3x + 1 - 1 = 10 - 1 \overset{\text{simplify}}{\rightsquigarrow} 3x = 9 \overset{/3}{\rightsquigarrow} 3x / 3 = 9 / 3 \overset{\text{simplify}}{\rightsquigarrow} x = 3$$

Evaluation Function



Recursive Evaluation Algorithm

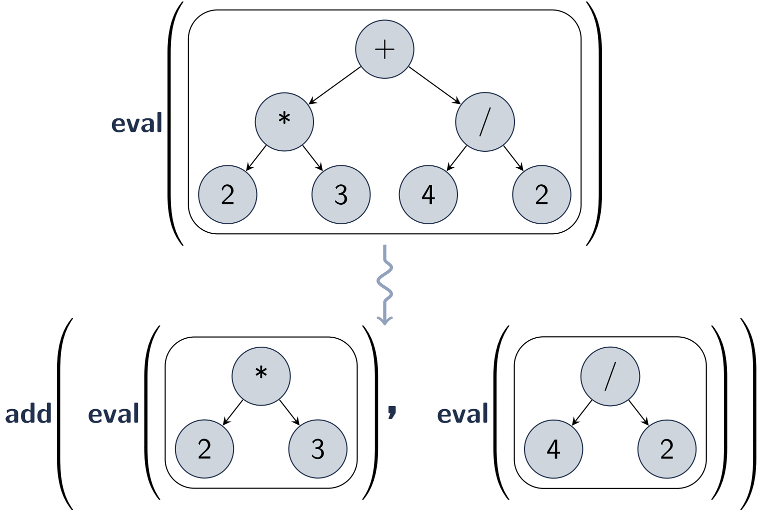
Base Case: If argument is a value: return the value

Recursive Case: Else (argument is an expression):

1. Recurse on arguments
2. Apply operator to results

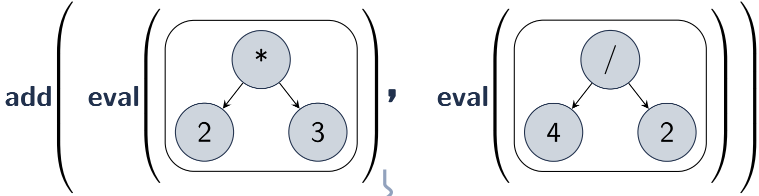
Example: Evaluation

2*3 + 4/2



Example: Evaluation

2*3 + 4/2 – continued



$$\rightsquigarrow$$

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

Example: Evaluation

2*3 + 4/2 – continued

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

$$\rightsquigarrow$$

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

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

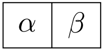
(+ (* 2 3)
(/ 4 2)))

$\longrightarrow$ $\texttt{eval} : \texttt{s-exp} \mapsto \mathbb{R}$ $\longrightarrow$ 8

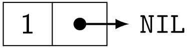
CONStruct

Creating Lists

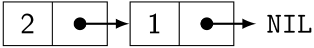
$(\texttt{cons}\ \alpha\ \beta) \rightsquigarrow (\alpha\ .\ \beta) \rightsquigarrow$



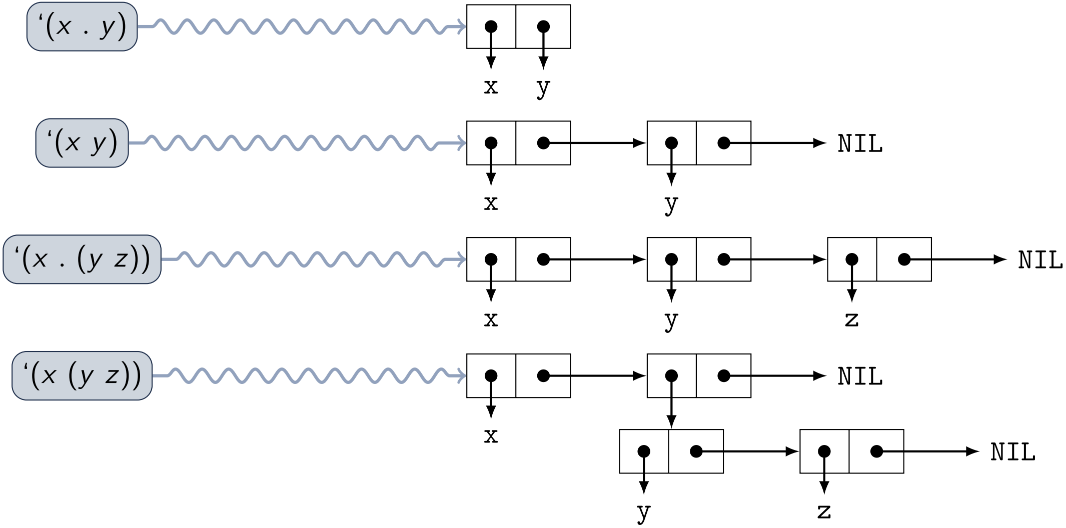
$(\texttt{cons 1 NIL}) \rightsquigarrow (1\ .\ \texttt{nil}) \rightsquigarrow (1) \rightsquigarrow$



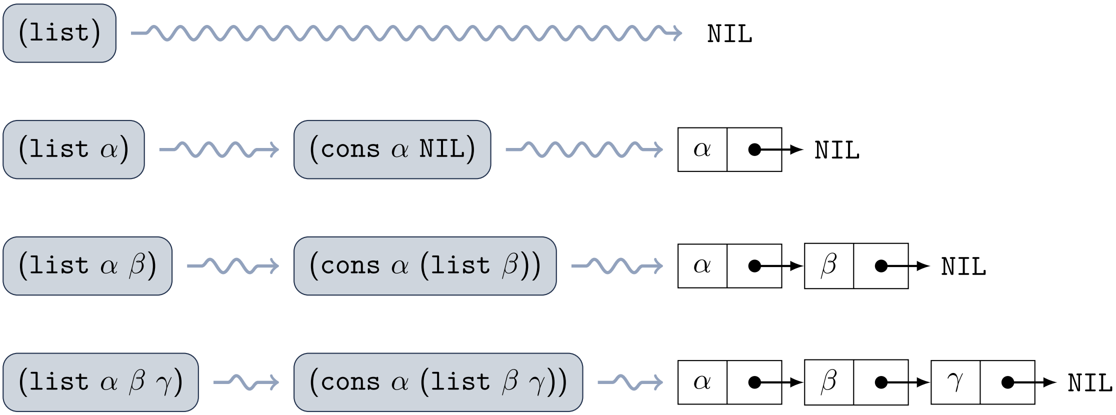
$(\texttt{cons}\ 2\ (\texttt{cons 1 NIL})) \rightarrow (2\ .\ (1\ .\ \texttt{nil})) \rightsquigarrow (2\ 1) \rightsquigarrow$



Dotted List Notation



List Function



S-Expression Quoting

Expressions vs. Execution

Execute: $(\texttt{fun}\ a\ b\ c) \rightsquigarrow$ return value of $\texttt{fun}$ called on arguments a, b, and c

Expression: $'(\texttt{fun}\ a\ b\ c) \rightsquigarrow$ The s-expression $(\texttt{fun}\ a\ b\ c)$

Examples:

- $(\texttt{list}\ 1\ 2\ 3) \rightsquigarrow (1\ 2\ 3)$
- $(\texttt{list}\ (+\ 1\ 2)\ 3) \rightsquigarrow (\texttt{list}\ 3\ 3) \rightsquigarrow (3\ 3)$
- $(\texttt{list}\ '(+\ 1\ 2)\ 3) \rightsquigarrow ((+\ 1\ 2)\ 3)$
- $(\texttt{list}\ '+\ 1\ (*\ 2\ 3)) \rightsquigarrow (\texttt{list}\ '+\ 1\ '6) \rightsquigarrow (+\ 1\ 6)$

Exercise: List Construction

- $(\texttt{cons}\ 'x\ 'y) \quad \rightsquigarrow$
- $(\texttt{cons}\ 'x\ '(y\ z)) \quad \rightsquigarrow$
- $(\texttt{cons}\ 'x\ (\texttt{list}\ 'y\ 'z)) \quad \rightsquigarrow$
- $(\texttt{list}\ (+\ 1\ 2\ 3)) \quad \rightsquigarrow$
- $(\texttt{list}\ '(+\ 1\ 2\ 3)) \quad \rightsquigarrow$
- $(\texttt{list}\ '*\ (+\ 2\ 2)\ '(-\ 2\ 2)) \quad \rightsquigarrow$
- $(\texttt{list}\ '+\ '(*\ a\ 2)\ (*\ 3\ 4)) \quad \rightsquigarrow$

Example: Evaluation via S-Expressions

$2*3 + 4/2$

```
(eval '(+ (* 2 3)
          (/ 4 2)))
```

```
(+ (eval '(* 2 3))
   (eval '(/ 4 2))))
```

```
(+ (* (eval '2)
      (eval '3))
   (/ (eval '4)
      (eval 2))))
```

```
(+ (* 2
      3)
   (/ 4
      2)))
```

```
(+ 6
   2))
```

```
8
```

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

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

Recursive Implementation

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)));
```

Iterative Implementation

Procedure map(f,s)

```
n ← length(s);
Y ← make-sequence(n);
i ← 0;
while i < n do
    Y[i] = f(s[i]);
return Y;
```

Exercise 1: Evaluation

$$\boxed{2*(1+2+3)\ \text{-}\ 5}$$

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)
```

$\rightsquigarrow$

```
(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 first**(S);
    else return cons $('+, S)$;
else
    if $\emptyset = S$ **then return** n;
    else return cons $('+,$ **cons** $(n, S))$;

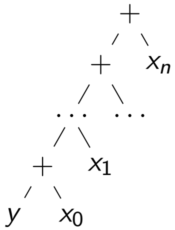
Fold-Left

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)

$i \leftarrow 0$;
while $i < |X|$ **do**
  $y \leftarrow f(y, X_i)$;
return y;

Recursive

Function fold-left(f,y,X)

if `empty?`(X) **then** **return** y ; `/* Base Case */`
else `/* Recursive Case */`
  $y' \leftarrow f(y, \textbf{first}(X))$;
  return fold-left $(f, y', \textbf{rest}(X))$;

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:	$\frac{d}{dt}(f(t) * g(t))$	$\leadsto$	$\frac{d}{dt}f(t) + \frac{d}{dt}g(t)$
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: $\sin' x \rightsquigarrow \cos x$

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

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

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

Differentiation Steps

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

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

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

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

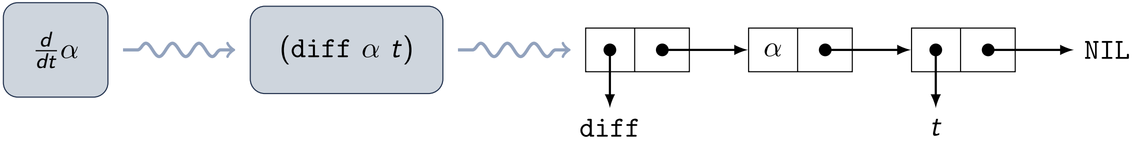
$$\left(-\sin t^2\right) * \frac{d}{dt} t^2$$

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

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

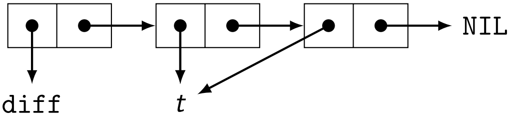
Just apply rewrite rules

Differentiation via Symbolic Expressions

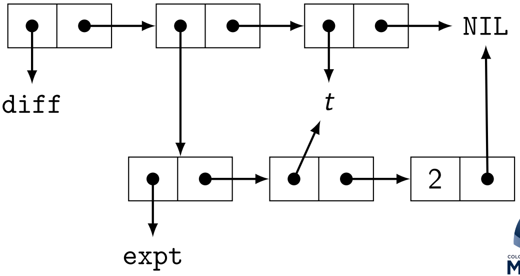


Example: Differentiation S-exps

$\frac{d}{dt}t$ ⇝ `(diff t t)` ⇝



$\frac{d}{dt}t^2$ ⇝ `(diff (expt t 2) t)` ⇝



Exercise: Differentiation S-exps

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

Differential Calculus

S-expression Rewrite Rules

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

Constant:	$(\mathtt{diff}\ k\ t)$	$\rightsquigarrow$	0
Variable:	$(\mathtt{diff}\ t\ t)$	$\rightsquigarrow$	1
Constant Power:	$(\mathtt{diff}\ (\mathtt{expt}\ t\ k)\ t)$	$\rightsquigarrow$	$(*\ k\ (\mathtt{expt}\ t\ (-\ k\ 1)))$
Addition:	$(\mathtt{diff}\ (+\ (f\ t)\ (g\ t))\ t)$	$\rightsquigarrow$	$(+\ (\mathtt{diff}\ (f\ t)\ t)\ (\mathtt{diff}\ (g\ t)\ t))$
Multiplication:	$(\mathtt{diff}\ (+\ (f\ t)\ (g\ t))\ t)$	$\rightsquigarrow$	$(+\ (*\ (\mathtt{diff}\ (f\ t)\ t)\ (g\ t))$ $(*\ (f\ t)\ (\mathtt{diff}\ (g\ t)\ t)))$
Chain Rule:	$(\mathtt{diff}\ (f\ (g\ t)))$	$\rightsquigarrow$	$(*\ (\mathtt{deriv}\ f\ (g\ t))$ $(\mathtt{diff}\ (g\ t)\ t))$

Exercise: Differential Calculus

S-expression Rewrite Rules

Subtraction:

$$\frac{d}{dt}\left(f(t) - g(t)\right) \quad \rightsquigarrow \quad \frac{d}{dt} f(t) - \frac{d}{dt} g(t)$$

Division:

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

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)
```

$\rightsquigarrow$ (diff (f (g t))) $\rightsquigarrow$ (∗ ((deriv f (g t))) (diff (g t)))

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

$\rightsquigarrow$ (deriv cos x) $\rightsquigarrow$ (− (sin x))

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

$\rightsquigarrow$ (diff (expt t k)) $\rightsquigarrow$ (∗ k (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)))
```

List Template Syntax

Backquote (‘): Create a template

- ‘$(x_0 \; \ldots \; x_n) \quad \rightsquigarrow \quad$ (list $'x_0 \; \ldots \; 'x_n$)
- `‘(+ a (* b c))` $\rightsquigarrow$ `(list '+ 'a (list '* b c))` $\rightsquigarrow$ `(+ a (* b c))`

Comma (,): Evaluate next element and insert into list

- ‘$(\alpha \ldots \; ,y \; \beta \ldots) \quad \rightsquigarrow \quad$ (list $\alpha \ldots \overbrace{y}^{\text{evaluated}} \beta \ldots$)
- `‘(+ a ,(* 2 3))` $\rightsquigarrow$ `(list '+ 'a ,(* 2 3))` $\rightsquigarrow$ `(+ a 6)`

Comma-At (,@): Evaluate next element and splice into list

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

Exercise: List Template Syntax

- `` `(1 2 ,(+ 3 4)) `` $\rightsquigarrow$

- `` `(,1 ,2 (+ 3 4)) `` $\rightsquigarrow$

- `` `(+ 1 ,2 ,(+ 3 4)) `` $\rightsquigarrow$

- `` `(1 2 ,@(list '+ '3 '4)) `` $\rightsquigarrow$

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)

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

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

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)

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

$a_0 \leftarrow$ **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 $`(* \;\; ,\overbrace{(\texttt{deriv}\ f\ a_0)}^{f'(g(v))} \;\; ,\overbrace{(\texttt{d-eval}\ a_0\ v)}^{\frac{d}{dv} g(v)})$;

Deriv Function

$f'(g(t))$

'cos

(expt t 2)

$$\texttt{deriv} : \overbrace{\texttt{symbol}}^{f} \times \overbrace{\texttt{s-exp}}^{g(t)} \mapsto \overbrace{\texttt{s-exp}}^{f'(g(t))}$$

(- (sin (expt t 2)))

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)
```

$\leadsto$ `d-eval-chain`

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

$\leadsto$ (deriv cos x) $\leadsto$ ($-$ (sin x))

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

$\leadsto$ `d-eval-expt`

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

$\leadsto$ $\frac{d}{dt} t = 1$

```
`(* (- (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

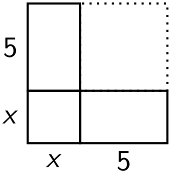
Exercise 2: Symbolic Differentiation Recursion Trace

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

Outline

Historical Note: Algebra

"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 موسی خوارزمی
"Algoritmi"
C.E 780-850

Modern Notation

$$
\begin{aligned}
x^2 + 10x &= 39 \\
x^2 + 10x + 25 &= 39 + 25 \\
(x+5)^2 &= 64 \\
x + 5 &= 8 \\
x &= 3
\end{aligned}
$$

Sapir-Whorf Hypothesis

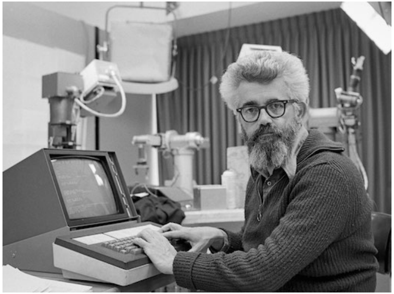
Language determines / constraints thought.

Edward Sapir

Benjamin Lee Whorf

Appropriate language/notation/abstraction makes math easier.

S-Expressions and Programming



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

"Math:"

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

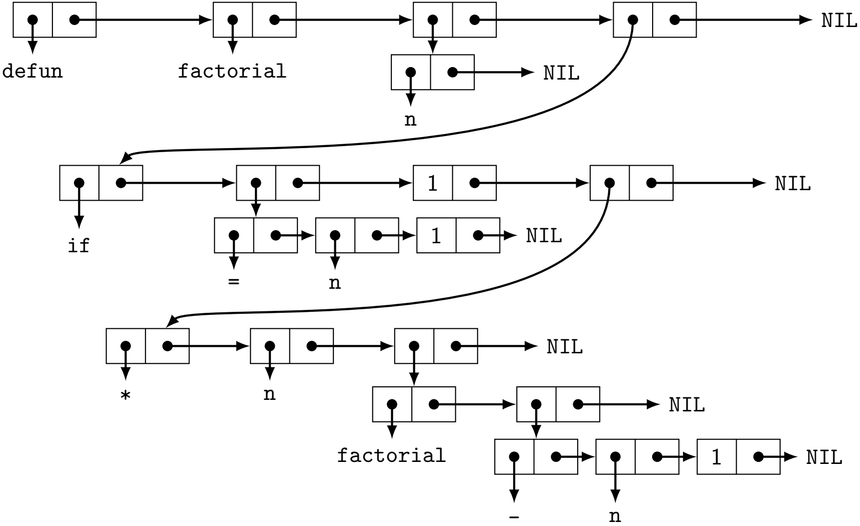
M-expression:

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

S-expression:

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

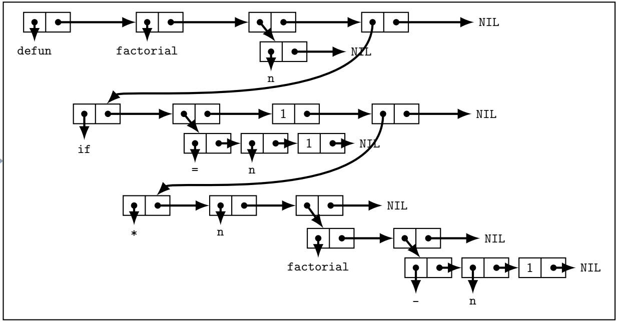
Factorial Cell Diagram



Homoiconic

Code is Data

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



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

Lisp

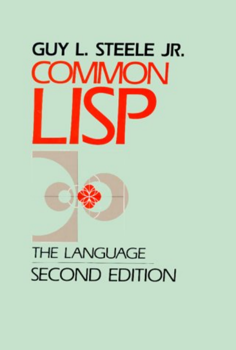
"LISt Processor"

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

Rewrite Systems

Expressions

Reductions

- Evaluation as Reduction
- Partial Evaluation
- Differentiation

Notation and Programming