Abstract Syntax for Arithmetic

Expressions like

$$\Big(\big((3+x)*(y*(y+2))\big)+(4*4)\Big)$$

What to do with expressions?

- **Evaluate** them, given values for the variables.
  (If $x = 2$ and $y = 3$, the expression above has value 91.)
- "**Simplify**" them, for example,

$$\Big(\big((3+x)*(y*(y+2))\big)+\mathbf{16}\Big)$$

- **Print** them, perhaps in **prefix** format
- Take symbolic **derivative**, say wrt x.
- **Make them!**

```
(value-of '(((3 + x) * (y * (y + 2))) + (4 * 4))
          '((x 2) (y 3)))
    ⟹ 91

(print-in-prefix '(((3 + x) * (y * (y + 2))) + (4 * 4)))

    ⟹ (+ (* (+ 3 x) (* y (+ y 2))) (* 4 4))

(derivative '(((3 + x) * (y * (y + 2))) + (4 * 4)) 'x)

    ⟹ (y * (y + 2))
```

For today,

Four Kinds of Abstract Expressions

An ***Expr*** is either a

Constant, ***Variable***, ***Sum***, ***Product***

Constructors	Selectors
`make-constant`: *Sch-Num* $\rightarrow$ *Constant*	`constant-value`: *Sch-Num* $\rightarrow$ *Constant*
`make-variable`: *Symbol* $\rightarrow$ *Variable*	`variable-name`: *Variable* $\rightarrow$ *Symbol*
`make-sum`: *Expr* $\times$ *Expr* $\rightarrow$ *Sum*	`addend`: *Sum* $\rightarrow$ *Expr* `augend`: *Sum* $\rightarrow$ *Expr*
`make-product`: *Expr* $\times$ *Expr* $\rightarrow$ *Product*	`multiplier`: *Product* $\rightarrow$ *Expr* `multiplicand`: *Product* $\rightarrow$ *Expr*

Contract:

Selectors give back what Constructors put in

`(addend (make-sum` $\langle \mathrm{expr}_1 \rangle$ $\langle \mathrm{expr}_2 \rangle$`))` $\Longrightarrow$ $\langle \mathrm{expr}_1 \rangle$

`(multiplicand (make-product` $\langle \mathrm{expr}_1 \rangle$ $\langle \mathrm{expr}_2 \rangle$`))` $\Longrightarrow$ $\langle \mathrm{expr}_2 \rangle$

`(constant-value (make-constant 3.2))` $\Longrightarrow$ `3.2`

•

•

•

Also have predicates

`constant?  variable?  sum?  product?`

of type $Expr \rightarrow Bool$.

Contract:

`(constant? (make-constant` $\langle$num$\rangle$`))` $\Longrightarrow$ `#t`

`(constant? (make-variable 'a))` $\Longrightarrow$ `#f`

`(sum? (make-sum` $\langle \mathrm{expr}_1 \rangle$ $\langle \mathrm{expr}_2 \rangle$`))` $\Longrightarrow$ `#t`

`(sum? (make-product` $\langle \mathrm{expr}_1 \rangle$ $\langle \mathrm{expr}_2 \rangle$`))` $\Longrightarrow$ `#f`

•
•
•

A *Variable*

- Has a **printable name**

Constructor	Selector
`make-variable`: *Symbol* $\rightarrow$ *Variable*	`variable-name`: *Variable* $\rightarrow$ *Symbol*

- May have a **numerical value** which can be looked up in an **Environment**.

An Arithmetic *Environment*

Associates number values with variables.

A **binding** is a list consisting of a symbol and a number, for example,

`(x 2)` or `(y 3)`.

$$\texttt{make-environment} : \mathit{List}(\text{binding}) \rightarrow \mathit{Environment}$$
$$\texttt{lookup} : \mathit{Variable} \times \mathit{Environment} \rightarrow (\mathit{Sch\text{-}Num} \cup \{\texttt{\#f}\})$$

Contract (by example):

```
(define env (make-environment '((x 2) (y 3))))
```

`(lookup (make-variable 'y) env)` $\Longrightarrow$ `3`

`(lookup (make-variable 'z) env)` $\Longrightarrow$ `#f`

Arithmetic Evaluation of Abstract Expressions

```
(define (ar-eval expr env)
  (cond ((constant? expr) (constant-value expr))
        ((variable? expr) (lookup expr env))
        ((sum? expr)
         (+ (ar-eval (addend expr) env)
            (ar-eval (augend expr) env)))
        ((product? expr)
         (* (ar-eval (multiplier expr) env)
            (ar-eval (multiplicand expr) env)))
        (else (error "bad expression" expr))))
```

Printing Abstract Expressions

The Scheme Approach: **convert the expression** into a **value**, usually a list or tree, which **Scheme's built-in printer will display** in the desired format.

For example,

```
(define expr (make-sum (make-product (make-constant 9)
                                     (make-variable 'x))
                       (make-sum (make-constant 4.1)
                                 (make-variable 'y))))
```

```
  (expr->prefix-list expr) ⟹ (+ (* 9 x) (+ 4.1 y))
   (expr->infix-list expr) ⟹ ((9 * x) + (4.1 + y))
 (expr->infix-string expr) ⟹ "((9 * x) + (4.1 + y))"
```

output representations:
two lists and a string

`expr->prefix-list` : $Expr \rightarrow \mathrm{Tree}(Sch\text{-}Num \cup Symbol)$

```
(define (expr->prefix-list expr)
  (cond ((constant? expr) (constant-value expr))
        ((variable? expr) (variable-name expr))
        ((sum? expr)
         (list '+           ; put operator symbol at the beginning
               (expr->prefix-list (addend expr))
               (expr->prefix-list (augend expr))))
        ((product? expr)
         (list '*
               (expr->prefix-list (multiplier expr))
               (expr->prefix-list (multiplicand expr))))
        (else (error "bad expression" expr))))
```

`expr->infix-string` : $Expr \to String$

```
(define (expr->infix-string expr)
  (cond ((constant? expr)
         (number->string (constant-value expr)))
        ((variable? expr)
         (symbol->string (variable-name expr)))
        ((sum? expr)
         (string-append
          "("             ; put in our own parens
          (expr->infix-string (addend expr))
          " + "           ; put operator string in the middle
          (expr->infix-string (augend expr))
          ")" ))
        ((product? expr)
             ⋮
             )))
```

User Interface

User describes expressions with (quoted) infix notation. The procedure `import-infix` converts user description into abstract expression:

$$\texttt{import-infix} : \mathrm{Tree}(\mathit{Sch\text{-}Num} \cup \mathit{Symbol}) \to \mathit{Expr}$$

```
(value-of '(((3 + x) * (y * (y + 2))) + (4 * 4))
          '((x 2) (y 3)))
```

$\Longrightarrow$ 97

```
(define (value-of user-expr bindings)
  (ar-eval (import-infix user-expr)
           (make-environment bindings)))
```

```
(define (import-infix user-expr)
  ((cond ((number? user-expr) (make-constant user-expr))
         ((symbol? user-expr) (make-variable user-expr))
         ((and (list? user-expr)
               (= (length user-expr) 3))
          (let ((left-exp  (car user-expr))
                (op        (cadr user-expr))
                (right-exp (caddr user-expr)))
            ((if (eq? op '+) make-sum make-product)
             (import-infix left-exp)
             (import-infix right-exp))))
         (else (error "ill-formed user expression" user-expr)))))
```

```
(expr->infix-list (import-infix '(3 + (x * y))))
    ⟹                          (3 + (x * y))
```

Our System

User level

`value-of` `print-in-prefix` `derivative` ...

Operations on abstract *Expr*

`import-infix` `ar-eval` `expr->prefix-list` ...

Implementation of abstract *Expr*

`sum?` `make-product` `make-environment` ...

Achievements of Our Design

Implementation independent (as long as the contract is met). In fact, we defined the operations on abstract expressions without knowing what the implementation was.

- Can optimize implementations of environments and lookup without changing anything else.

User level definitions are **independent of abstract operation** definitions.

- Can optimize abstract operations without changing user-level definitions.
- To support another user notation, need only define **one** new `import-notation` procedure.
- To support another output notation, need only define **one** new `expr->output` procedure.

Suppose an **expression will be printed many times**. Save time by **saving the printable representation** instead of recomputing it.

Save where? **Inside the expression!**
—the beginning of *Object-Oriented Programming*

```
(define (expr->prefix-list expr)
  (car expr))

(define (make-constant num)
  (cons num num))

(define constant-value cdr)
```

```
(define (make-sum expr1 expr2)
  (let ((pref-list (list '+
                         (expr->prefix-list expr1)
                         (expr->prefix-list expr2))))
    (cons pref-list
          (list '+ expr1 expr2))))

(define addend caddr)

(define augend cadddr)
```

Gains

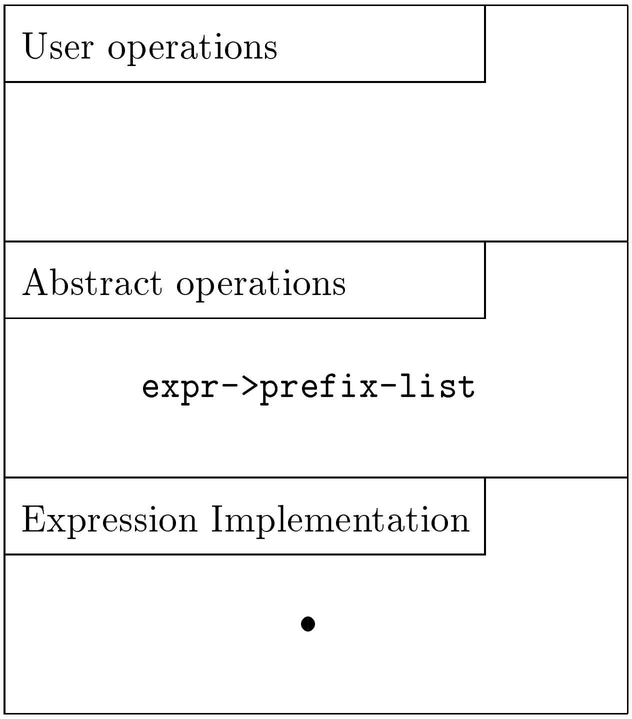
Faster to compute `expr->prefix-list`: only one `car`.

Costs

- **Space to store the prefix-list.**
- **More Time to make expressions**
  (wasted when `expr->prefix-list` is not applied to an expression).

Costs

- Definition of `expr->prefix-list` now **depends on implementation**.



Extend this idea to `ar-eval`, where it really counts:

Build into an expression ⟨expr⟩ an `ar-eval-at` procedure such that

$$\texttt{(ar-eval}\ \langle\text{expr}\rangle\ \texttt{env)} = \texttt{(ar-eval-at env)}$$

This wins because `ar-eval-at` can be customized for ⟨expr⟩, while `ar-eval` has to parse ⟨expr⟩ at every evaluation.

```
(define (ar-eval expr env)
  (let ((ar-eval-at (car expr)))
    (ar-eval-at env)))

(define (make-constant num)
  (cons (lambda (env) num) num))

(define constant-value cdr)
```

```
(define (make-sum expr1 expr2)
  (cons (lambda (env)
          (+ ((car expr1) env)        ;; ar-eval-at “knows” to add
             ((car expr2) env)))
        (list '+ expr1 expr2)))

(define addend caddr)

(define augend cadddr)
```

By ***partially*** **evaluating** an expression without knowing the environment, we have avoided recomputing the partial values at each full evaluation when an environment is given.