

## Recap: Concrete and Abstract Syntax

- Every language  $X$  has one ***concrete syntax***
- Programmers using language  $X$  write programs using the concrete syntax
- To represent programs in language  $X$  for processing with language  $Y$ , we represent ***abstract syntax*** for  $X$  programs
- The representation is specific to  $X$  in  $Y$ , but there is more than one choice

'(+ 1 2)

(plus (number 1) (number 2))

## Recap: Concrete and Abstract Syntax

- Every language  $X$  has one ***concrete syntax***
- Programmers using language  $X$  write programs using the concrete syntax
- To represent programs in language  $X$  for processing with language  $Y$ , we represent ***abstract syntax*** for  $X$  programs
- The representation is specific to  $X$  in  $Y$ , but there is more than one choice
- Abstract syntax is ***abstract*** because it omits irrelevant details

(“irrelevant” depends on the analysis task)

# Concrete Syntax for the Book Language

$\langle \text{prog} \rangle ::= \langle \text{expr} \rangle$   
 $\langle \text{expr} \rangle ::= \langle \text{num} \rangle$   
 $\quad ::= \langle \text{id} \rangle$   
 $\quad ::= \langle \text{prim} \rangle ( \{ \langle \text{expr} \rangle \}^{*(,)} )$   
 $\langle \text{prim} \rangle ::= + \mid - \mid * \mid \text{add1} \mid \text{sub1}$

Example:

1

# Concrete Syntax for the Book Language

$\langle \text{prog} \rangle ::= \langle \text{expr} \rangle$   
 $\langle \text{expr} \rangle ::= \langle \text{num} \rangle$   
 $\quad ::= \langle \text{id} \rangle$   
 $\quad ::= \langle \text{prim} \rangle ( \{ \langle \text{expr} \rangle \}^{*(,)} )$   
 $\langle \text{prim} \rangle ::= + \mid - \mid * \mid \text{add1} \mid \text{sub1}$

Example:

**x**

# Concrete Syntax for the Book Language

$\langle \text{prog} \rangle ::= \langle \text{expr} \rangle$   
 $\langle \text{expr} \rangle ::= \langle \text{num} \rangle$   
 $\quad ::= \langle \text{id} \rangle$   
 $\quad ::= \langle \text{prim} \rangle ( \{ \langle \text{expr} \rangle \}^{*(,)} )$   
 $\langle \text{prim} \rangle ::= + \mid - \mid * \mid \text{add1} \mid \text{sub1}$

Example:

$+(1, 2)$

# Concrete Syntax for the Book Language

$\langle \text{prog} \rangle ::= \langle \text{expr} \rangle$   
 $\langle \text{expr} \rangle ::= \langle \text{num} \rangle$   
 $\quad ::= \langle \text{id} \rangle$   
 $\quad ::= \langle \text{prim} \rangle ( \{ \langle \text{expr} \rangle \}^{*(,)} )$   
 $\langle \text{prim} \rangle ::= + \mid - \mid * \mid \text{add1} \mid \text{sub1}$

Example:

$+(1, 2, 3)$

# Concrete Syntax for the Book Language

$\langle \text{prog} \rangle ::= \langle \text{expr} \rangle$   
 $\langle \text{expr} \rangle ::= \langle \text{num} \rangle$   
 $\quad ::= \langle \text{id} \rangle$   
 $\quad ::= \langle \text{prim} \rangle ( \{ \langle \text{expr} \rangle \}^{*(,)} )$   
 $\langle \text{prim} \rangle ::= + \mid - \mid * \mid \text{add1} \mid \text{sub1}$

Example:

**add1(1)**

# Concrete Syntax for the Book Language

$\langle \text{prog} \rangle ::= \langle \text{expr} \rangle$   
 $\langle \text{expr} \rangle ::= \langle \text{num} \rangle$   
 $\quad ::= \langle \text{id} \rangle$   
 $\quad ::= \langle \text{prim} \rangle ( \{ \langle \text{expr} \rangle \}^{*(,)} )$   
 $\langle \text{prim} \rangle ::= + \mid - \mid * \mid \text{add1} \mid \text{sub1}$

Example:

**add1(+ (2, x))**



# Representation for the Book Language

**<prog> ::= (a-program <expr>)**  
**<expr> ::= (lit-exp <num>)**  
**::= (var-exp <symbol>)**  
**::= (primapp-exp <prim> (list <expr><sup>\*</sup>))**  
**<prim> ::= (add-prim) | (subtract-prim)**  
**::= (mult-prim) | (inc-prim) | (decr-prim)**

Concrete:                   1

Abstract representation:

**(a-program (lit-exp 1))**

# Representation for the Book Language

**<prog> ::= (a-program <expr>)**  
**<expr> ::= (lit-exp <num>)**  
**::= (var-exp <symbol>)**  
**::= (primapp-exp <prim> (list <expr><sup>\*</sup>))**  
**<prim> ::= (add-prim) | (subtract-prim)**  
**::= (mult-prim) | (inc-prim) | (decr-prim)**

Concrete:                   **x**

Abstract representation:

**(a-program (var-exp 'x))**

# Representation for the Book Language

**<prog> ::= (a-program <expr>)**  
**<expr> ::= (lit-exp <num>)**  
**::= (var-exp <symbol>)**  
**::= (primapp-exp <prim> (list <expr><sup>\*</sup>))**  
**<prim> ::= (add-prim) | (subtract-prim)**  
**::= (mult-prim) | (inc-prim) | (decr-prim)**

Concrete:               +(1, 2)

Abstract representation:

**(a-program**  
  **(primapp-exp (add-prim) (list (lit-exp 1) (lit-exp 2))))**

## Representation for the Book Language

**<prog>** ::= (**a-program** **<expr>**)  
**<expr>** ::= (**lit-exp** **<num>**)  
          ::= (**var-exp** **<symbol>**)  
          ::= (**primapp-exp** **<prim>** (**list** **<expr>**<sup>\*</sup>))  
**<prim>** ::= (**add-prim**) | (**subtract-prim**)  
          ::= (**mult-prim**) | (**inc-prim**) | (**decr-prim**)

But the connection between concrete and abstract/representation examples is only in our heads right now...

# Parsing

- Converting concrete syntax to abstract syntax is the job of a ***parser***
- Parsing is a deep topic with a long history...
- ... that we will ignore almost entirely
- The EoPL extensions to Scheme include a parser generator called ***SLLGEN***

(see parser example in DrScheme)

# Ways of Evaluating

- So far:

$$*(+(3, 4), -(2,1)) \rightarrow *(7, -(2,1)) \rightarrow *(7,1) \rightarrow 7$$

- Alternative:

$$\frac{+(3,4) = 7 \quad \quad -(2,1) = 1}{*(+(3,4), -(2,1)) = 7}$$

In other words, to evaluate an expression, first evaluate the sub-expressions, then combine their values

=> a recursive **eval-expression** function

## eval-expression

(implementation in DrScheme)

- Note: evaluating an identifier is an error for now

# Add Conditionals

- Concrete:

`<expr> ::= if <expr> then <expr> else <expr>`

- Abstract:

`<expr> ::= (if-exp <expr> <expr> <expr>)`

(update implementation in DrScheme)



# Add Local Bindings

- Concrete:

$\langle \text{expr} \rangle ::= \text{let } \{ \langle \text{id} \rangle = \langle \text{expr} \rangle \}^* \text{ in } \langle \text{expr} \rangle$

- Abstract:

$\langle \text{expr} \rangle ::= (\text{let-exp } (\text{list } \langle \text{symbol} \rangle^*) (\text{list } \langle \text{expr} \rangle^*) \langle \text{expr} \rangle)$

Evaluating an identifier isn't an error anymore... but how does `eval-expression` know the value of the identifier?

## Evaluating Let

- One possibility: for **let-exp** expressions, **eval-expression** could call **substitute** on the body
- Another possibility: **eval-expression** can perform the substitution lazily, as it goes
  - **eval-expression** now takes two arguments: an expression and a set of lazy substitutions
  - the set of lazy substitutions is called an ***environment***

# Environments

Implement environments as an ADT with three operations:

- (**empty-env**) : creates an empty environment; i.e., no substitutions
- (**extend-env** *<env>* (**list** *<symbol>*<sup>\*</sup>) (**list** *<val>*<sup>\*</sup>)) : creates a new environment that has the substitutions of *<env>*, plus (or instead of) the substitution of each *<symbol>* with *<val>*
- (**apply-env** *<env>* *<symbol>*) : extracts the substitution of *<symbol>* from *<env>*

# Environment Examples

```
(let ([s (extend-env '(x) '(1) (empty-env))])  
  (apply-env s 'x))
```

→→ 1

## Environment Examples

```
(let ([s (extend-env '(x y z) '(1 2 3) (empty-env))])
```

```
  (apply-env s 'y)
```

```
→→ 2
```

## Environment Examples

```
(let ([s (extend-env '(x y z) '(1 2 3) (empty-env))])  
  (let ([t (extend-env '(a y) '(5 6) s)])  
    (apply-env t 'a)
```

→→ 5

## Environment Examples

```
(let ([s (extend-env '(x y z) '(1 2 3) (empty-env))])  
  (let ([t (extend-env '(a y) '(5 6) s)])  
    (apply-env t 'y)
```

→→ 6

# Environment Examples

**(apply-env (empty-env) 'x)**

**→→ error**



# Implementing Let

(update implementation in DrScheme)