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

\[
'(+ 1 2) \\
(\text{plus (number 1) (number 2)})
\]

Concrete Syntax for the Book Language

\[
\begin{align*}
<\text{prog}> &::= <\text{expr}> \\
<\text{expr}> &::= <\text{num}> \\
&::= <\text{id}> \\
&::= <\text{prim}> ( \{ <\text{expr}> \}^{*} ) \\
<\text{prim}> &::= + | - | * | \text{add1} | \text{sub1}
\end{align*}
\]

Example:

1

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

\[
\begin{align*}
<\text{prog}> &::= <\text{expr}> \\
<\text{expr}> &::= <\text{num}> \\
&::= <\text{id}> \\
&::= <\text{prim}> ( \{ <\text{expr}> \}^{*} ) \\
<\text{prim}> &::= + | - | * | \text{add1} | \text{sub1}
\end{align*}
\]

Example:

\( x \)
Concrete Syntax for the Book Language

<prog> ::= <expr>
<expr> ::= <num>
   ::= <id>
   ::= <prim> ( { <expr> }* )
<prim> ::= + | - | * | add1 | sub1

Example:

+(1, 2)

Concrete Syntax for the Book Language

<prog> ::= <expr>
<expr> ::= <num>
   ::= <id>
   ::= <prim> ( { <expr> }* )
<prim> ::= + | - | * | add1 | sub1

Example:

add1(1)

Concrete Syntax for the Book Language

<prog> ::= <expr>
<expr> ::= <num>
   ::= <id>
   ::= <prim> ( { <expr> }* )
<prim> ::= + | - | * | add1 | sub1

Example:

add1((2, x))

Concrete Syntax for the Book Language

<prog> ::= <expr>
<expr> ::= <num>
   ::= <id>
   ::= <prim> ( { <expr> }* )
<prim> ::= + | - | * | add1 | sub1

Example:

(+(1, 2))
Representation for the Book Language

\[ \texttt{<prog>} ::= \texttt{(a-program <expr>)} \]
\[ \texttt{<expr>} ::= \texttt{(lit-exp <num>)} \]
\[ \texttt{<expr>} ::= \texttt{(var-exp <symbol>)} \]
\[ \texttt{<expr>} ::= \texttt{(primapp-exp <prim> (list <expr>*)}) \]
\[ \texttt{<prim>} ::= \texttt{(add-prim) | (subtract-prim)} \]
\[ \texttt{<prim>} ::= \texttt{(mult-prim) | (inc-prim) | (decr-prim)} \]

Concrete: 1

Abstract representation:
\[
\texttt{(a-program \texttt{(lit-exp 1)})}
\]

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:
  
  \[
  \ast(+(3, 4), -(2,1)) \rightarrow \ast(7, -(2,1)) \rightarrow \ast(7,1) \rightarrow 7
  \]

- Alternative:
  \[
  \begin{align*}
  +(3,4) &= 7 \\
  -(2,1) &= 1 \\
  \ast(+(3,4), -(2,1)) &= 7
  \end{align*}
  \]

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:
  \[
  <expr> ::= \text{if} <expr> \text{then} <expr> \text{else} <expr>
  \]

- Abstract:
  \[
  <expr> ::= (\text{if-exp} <expr> <expr> <expr>)
  \]

  (update implementation in DrScheme)
Add Local Bindings

- Concrete:
  \[
  \text{<expr>} ::= \text{let } \{ \text{id} = \text{<expr>} \}^* \text{ in } \text{<expr>}
  \]

- Abstract:
  \[
  \text{<expr>} ::= (\text{let-exp } (\text{list } \text{<symbol>}^*) (\text{list } \text{<expr>}^*) \text{<expr>})
  \]

Evaluating an identifier isn’t an error anymore... but how does \text{eval-expression} know the value of the identifier?

Evaluating Let

- One possibility: for \text{let-exp} expressions, \text{eval-expression} could call \text{substitute} on the body

- Another possibility: \text{eval-expression} can perform the substitution lazily, as it goes
  - \text{eval-expression} now takes two arguments: an expression and a set of lazy substitutions
  - the set of lazy substitutions is called an \textit{environment}

Environments

Implement environments as an ADT with three operations:

- \text{(empty-env)} : creates an empty environment; i.e., no substitutions

- \text{(extend-env } \text{<env>} (\text{list } \text{<symbol>}^*) (\text{list } \text{<val>}^*)) : creates a new environment that has the substitutions of \text{<env>}, plus (or instead of) the substitution of each \text{<symbol>} with \text{<val>}

- \text{(apply-env } \text{<env>} \text{<symbol>}) : extracts the substitution of \text{<symbol>} from \text{<env>}

Environment Examples

\[
(\text{let } ([s (\text{extend-env } '(x) '(1) \text{ (empty-env))}) \text{ (apply-env } s 'x))
\]
\[
\rightarrow \rightarrow 1
\]
Environment Examples

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

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

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

(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

34-37
Implementing Let

(update implementation in DrScheme)