Principles of Programming
Languages II

Wael Aboelsaadat
wael@cs.toronto.edu
http://www.cs.toronto.edu/~wael/324.html
Today

- Grammar

- Introduction to Functional Programming
Administrivia

- Office hour(s)

- Feedback
Grammar
Grammar: introduction

• Grammar:
  – A Grammar is a formalism that describes which sequence of terminals are meaningful in a PL. Mathematically, it is defined as a quadruple \((N, T, P, S)\) where:
    • \(N\) is the set of symbols called **Nonterminals**
    • \(T\) is the set of symbols called **Terminals**
    • \(P\) is the set of **productions**
    • \(S\ subsetof N\) is the nonterminal called the **starting symbol**
  – Example:
    \[ G = (N,T,P,S) \text{ where } N = \{S\} , T = \{a,b\}, \]
    \[ P = \{ S \rightarrow aS, S \rightarrow bS, S \rightarrow \} \]

• Production:
  – A **production** is a rule of the form \(X \rightarrow Y\) where \(X\) is a string of symbols (terminals or nonterminals) containing **at least one nonterminal**, and \(Y\) is a string of symbols (terminals or nonterminals)
Grammar: context free

- A *context free grammar* (CFG) is a grammar in which $|X| = 1$, i.e. X is a single nonterminal
  - LHS: 1 nonterminal
  - RHS: a sequence of terminals and nonterminals
  - E.g.
    - $S \Rightarrow ab$ (CFG)
    - $SA \Rightarrow ab$ (non CFG)

- CFG is sufficient to describe most of the constructs in programming languages

- Programming languages describable by CFG are recognizable by push down automata *(analogues to FSA with a stack)*
Grammar: backus Naur form

• Backus Naur Form (BNF) is a metalanguage for describing programming languages

• A BNF grammar is a context free grammar

• Notation:
  – Nonterminals are enclosed in angle brackets, i.e. “<“ and “>”
  – Uses “::=“ instead of “⇒” in productions
  – Productions having the same left hand side can be grouped together using the alteration symbol “|”
    • e.g. <S> ::= a <S> | b <S> |
  – Lists are described using recursive rules
    • e.g. <ident.list> ::= identifier | identifier, <ident.list>
Grammar: extended BNF

• Notation:
  – (... | ... | ...) Any one of the alterations
  – [...] Optional part
  – (...)* or {...} or [...] repeat zero or more times
  – (...)* or {...} or [...] repeat one or more times
  – "x" or 'x' terminal symbol
  – Unquoted words non-terminal symbol

• Example:
  – Using the above notation
    \[ <expression> ::= <expression> + <term> \]
    \[ | <expression> - <term> \]
    \[ | <term> \]

    could be written in the form of an iteration, as follows:
    \[ <expression> ::= <term> [ ( + | - ) <term> ]* \]
Grammar: BNF recursive rules

• **Left Recursive BNF Grammar:**
  – A BNF grammar rule is *left recursive* if its LHS appears at the left end of the RHS
    e.g. \(<\text{ident.list}> ::= <\text{ident.list}> , \text{identifier} | \text{identifier}\)
    e.g. \(A \rightarrow A x | y\) (parse yxx)

• **Right Recursive BNF Grammar:**
  – A BNF grammar rule is *right recursive* if its LHS appears at the right end of the RHS
    e.g. \(<\text{ident.list}> ::= \text{identifier} | \text{identifier}, <\text{ident.list}>\)
    e.g. \(A \rightarrow x A | y\) (parse xxy)

*The order of recursion has implications on the order of evaluation and associativity.*
• **Leftmost Derivation:**
  – In a *leftmost derivation*, the replaced nonterminal is *always* the leftmost nonterminal.

• **Rightmost Derivation:**
  – In a *rightmost derivation*, the replaced nonterminal is *always* the rightmost nonterminal.

• **Typically, language parsers either do leftmost or rightmost derivation on the grammar**
  – LR parser: left-to-right right most derivation
  – LL parser: left-to-right left most derivation
Grammar: definitions...

• Sentence
  – A finite sequence of terminals, constructed according to the rules of the grammar for that PL

• Sentential form
  – A finite sequence of terminals and non-terminals constructed according to the rules of the grammar for that PL

• Derivation

• Parse
Grammar: derivation

• Given the grammar

\[
\begin{align*}
\text{<letter>} &::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z \quad [1] \\
\text{<digit>} &::= 0|1|2|3|4|5|6|7|8|9 \quad [2] \\
\text{<identifier>} &::= \text{<letter>} | \text{<identifier>} \text{<letter>} | \text{<identifier>} \text{<digit>} \quad [3] \\
\text{<assign-stmt>} &::= \text{<identifier>} = 0 \quad [4]
\end{align*}
\]

• Can we generate \textit{x2=0} from these rules?

\[
\begin{align*}
\text{<assign-stmt>} \Rightarrow & \quad \text{<identifier>} = 0 \quad \text{(using 4)} \\
\Rightarrow & \quad \text{<identifier>} \text{<digit>} = 0 \quad \text{(using 3c)} \\
\Rightarrow & \quad \text{<letter>} \text{<digit>} = 0 \quad \text{(using 3a)} \\
\Rightarrow & \quad x \text{<digit>} = 0 \quad \text{(using 1)} \\
\Rightarrow & \quad x \quad 2 = 0 \quad \text{(using 2)}
\end{align*}
\]

• Yes! This is a \textit{derivation of a sentence} in the language described by the grammar above. Each sequence in this derivation is a \textit{sentential form}. At each step, the rule indicated is used to substitute the rhs of the rule for the leftmost \textit{non-terminal} in the sentential form.
Grammar: parsing

• Given the grammar

\[
\begin{align*}
\text{<letter>} &::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z \quad [1] \\
\text{<digit>} &::= 0|1|2|3|4|5|6|7|8|9 \quad [2] \\
\text{<identifier>} &::= \text{<letter>} | \text{<identifier>} \text{<letter>} | \text{<identifier>} \text{<digit>} \quad [3] \\
\text{<assign-stmt>} &::= \text{<identifier>} = 0 \quad [4]
\end{align*}
\]

• Can we recognize \(x^2 = 0\) from these rules?

\[
\begin{align*}
x^2=0 &\rightarrow \text{<letter>} \quad 2 = 0 \quad \text{(using 1)} \\
&\rightarrow \text{<identifier>} \quad 2 = 0 \quad \text{(using 3a)} \\
&\rightarrow \text{<identifier><digit>} = 0 \quad \text{(using 2)} \\
&\rightarrow \text{<identifier>} = 0 \quad \text{(using 3c)} \\
&\rightarrow \text{<assign-stmt>}
\end{align*}
\]

• This is a parse of the sentence \(x^2 = 0\)
Grammar: building parse trees

\[
x2=0 \rightarrow \text{<letter>} 2 = 0
\rightarrow \text{<identifier>} 2 = 0
\rightarrow \text{<identifier><digit>} = 0
\rightarrow \text{<identifier>} = 0
\rightarrow \text{<assign-stmt>}
\]

• A **parse tree** describes the hierarchical syntactic structure of the sentence based on a given language

• **In a parse tree**
  – Each internal node is a **non-terminal**, its children are the rhs of a rule for that non-terminal.
  – All leafs are **terminals**
Grammar: grammars are not unique

\[
\begin{align*}
\text{<letter>} &::= \text{a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z} \\
\text{<digit>} &::= 0|1|2|3|4|5|6|7|8|9 \\
\text{<id>} &::= \text{<letter>} | \text{<id> <letterordigit>} \\
\text{<assign-stmt>} &::= \text{<id> = 0} \\
\text{<letterordigit>} &::= \text{<letter>} | \text{<digit>}
\end{align*}
\]

• This grammar generates the same language (i.e. set of trees whose frontiers/leafs are the same), but has different parse trees than the previous grammar.

Many grammars can correspond to 1 PL, but only 1 PL should correspond to any useful grammar!
Grammar: ambiguity

• Ambiguity
  – If there are 2 different derivations (or alternatively 2 parse trees) for the same sentence then the grammar is ambiguous

• There is no algorithm which can examine an arbitrary context-free grammar and tell if it is ambiguous or not
  – This is undecidable

• There is no algorithm which can examine two arbitrary context-free grammars and tell if they generate the same language
  – This is undecidable
Grammar: inherently ambiguous

- Sometimes we can remove an ambiguity from a grammar by restructuring the productions but it is not always possible.

- An inherently ambiguous language does not possess an unambiguous grammar.

- Consider $L = \{a^i b^j c^k \mid i = j \text{ or } j = k \text{ for } i,j,k \geq 1\}$ generated by the grammar:
  
  $$S \ ::= \ L \ C \ | \ A \ D$$
  $$L \ ::= \ a \ L \ b \ | \ ab$$
  $$C \ ::= \ c \ | \ cC$$
  $$D \ ::= \ bDc \ | \ bc$$
  $$A \ ::= \ a \ | \ aA$$

Try $a^3 b^3 c^3$ in $L$.

Problem is $L$ contains a non-context free language $a^n b^n c^n$ where $n \geq 1$. 
Grammar: sources of ambiguity

• **Operator Problem:** *associativity and precedence*
  – E.g.
    • Precedence of multiplication/subtraction/addition/…
  – Solution:
    • Change the grammar to reflect operator precedence
      \[X*Y-Z \text{ means } ((X*Y) - Z)\]

• **Obscure recursion**
  – E.g.
    • \(\text{exp} \rightarrow \text{exp} \text{ exp}\)
    • \(\text{A} \rightarrow \text{A B}\)
  – Solution:
    • ??

• **Substructure Problem:** *extent of a substructure*
  – E.g.
    • Dangling else
  – Solution:
    • Coming slides!
Yes, because the sentence $A = B - C - A$ has two different parse trees. The grammar does not force "normal" left-to-right evaluation of addition and subtraction.
Grammar: is it really a problem?

- The operation of addition is associative in mathematics. Hence
  \[ A + B + C \]
  can be performed as either
  \[ (A + B) + C \]
  or \[ A + (B + C) \].

The multiply operation is also associative.

Therefore one might say the previous ambiguous grammar would be satisfactory for addition and multiplication.

But would it?
Grammar: is this ambiguous?

\[
\begin{align*}
\texttt{< assign >} & : : = & \texttt{< identifier >} = \texttt{< expression >} & \{1\} \\
\texttt{< identifier >} & : : = & \texttt{A|B|C} & \{2\} \\
\texttt{< expression >} & : : = & \texttt{< expression > + < term >} & \{3\} \\
& & | \texttt{< expression > - < term >} & \{4\} \\
& & | \texttt{< term >} & \{5\} \\
\texttt{< term >} & : : = & \texttt{< term >*< factor >} | \texttt{< factor >} & \{6\} \\
\texttt{< factor >} & : : = & ( \texttt{< expression >}) | \texttt{< identifier >} & \{7\}
\end{align*}
\]

Tree for \( A = B + C \times A \)
Grammar: If-then-else

- Grammar:
  \[
  \langle \text{if stmt} \rangle ::= \text{if} \ \langle \text{logic expression} \rangle \ \text{then} \ \langle \text{stmt} \rangle \\
  \quad | \ \text{if} \ \langle \text{logic expression} \rangle \ \text{then} \ \langle \text{stmt} \rangle \ \text{else} \ \langle \text{stmt} \rangle \\
  \langle \text{stmt} \rangle ::= \langle \text{if stmt} \rangle | \ldots
  \]

- Consider:
  if (logic-expression) then
  if (logic-expression) then
    statement 1
  else
    statement 2
Grammar: If-then-else

- Grammar:
  \[
  \text{< if stmt > ::= if < logic expression > then < stmt >} \\
  \text{ | if < logic expression > then < stmt > else < stmt >} \\
  \text{< stmt > ::= < if stmt > | . . .}
  \]

- Example: if (x=0) then
  
  if (y = 0) then
   
  z := 1
  
  else
   
  w := 2

```
IF x = 0 THEN IF y = 0 THEN z := 1 ELSE w := 2;
```

Q: which tree is correct?
Grammar: substructure problem, how to solve ambiguity?

- **Use block structure to enclose if statement** *(e.g. Algol60)*
  
  - E.g. 
    
    ```plaintext
    if x = 0 then
      begin
        if y = 0 then
          z:= 1
        end
      end
    else
      w := 2
    end
    ```

- **Use statement begin/end markers** *(e.g. Algol68)*
  
  - E.g. 
    
    ```plaintext
    if x = 0 then
      if y = 0 then
        z := 1
      fi
    else
      w := 2
    fi
    ```

- **Change the if statement grammar to disallow parse tree 2; that is, always associate an else with the closet if** *(e.g. Pascal)*

  ```plaintext
  if x = 0 then
    if y = 0 then
      z := 1
    fi
  else
    w := 2
  fi
  ```
Grammar: If-then-else, Pascal solution

Start ::= Stmt
Stmt ::= Stmt1 | Stmt2
Stmt1 ::= IF LogExp THEN Stmt1 ELSE Stmt1 \| Astmt
Stmt2 ::= IF LogExp THEN Stmt1 IF LogExp THEN Stmt1 ELSE Stmt2
Astmt ::= Id := Digit
Digit ::= 0|1|2|3|4|5|6|7|8|9
LogExp ::= Id = 0
Id ::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z

Note: only if statements with IF..THEN..ELSE are allowed after the THEN clause of an IF-THEN-ELSE statement.

IF x = 0 THEN IF y = 0 THEN z := 1 ELSE w := 2;

In the new grammar there is only 1 parse tree!
Functional Programming Languages
FPL
FPL: why?

• “Can programming be liberated from the Von Neumann style?”
  
  *John Backus*

• **Problems with Imperative programming languages:**
  – Von Neumann bottle neck (i.e. fetching words across bus)
  – Assignment
  – Side-effects
  – State-based transformation

• **FPL alternative:**
  – Goal? mimic mathematical functions to the greatest extent possible
  – How? Use $\lambda$ calculus for the computation
  – The program is a mathematical function

• **FPL-based solutions:**
  – Hardware: Symbolics Machine, TI Explorer…
  – Software: Lisp, Scheme, ML, Haskell, Miranda …
FPL: mathematical vs. imperative functions

• Recall: how do imperative functions work?
  – Specify a sequence of operations on values in memory to produce a value
  – Evaluation is controlled by sequencing and iteration

• Why are mathematical functions different?
  – The value is defined and not produced
  – Evaluation order is controlled by recursion and conditional expressions

• Example:
  – Write a procedure to implement the following function \(f(x) = x \times x / 3\)
    • Imperative: procedure float foo( var int x )
      begin
        int product;
        float quotient;
        product := x \times x;
        quotient := product/ 3;
        return quotient;
      end;
    • Functional: ??
FPL: desiderata

• **A program consists of:**
  – Function definitions
  – Function calls
  – There is no other structure.

• **Control flow:**
  – Recursion and function application is the only way to achieve repetition

• **No assignment**
  – Values are bound to values only through parameter association

• **No side effects**
  – A function may not change its parameters
  – A function cannot do input or output

• **No variables**
  – No variable declaration
  – No explicit typing
FPL: desiderata cont’d

• **Implicit memory management:**
  – no new or free
  – Program unaware of underlying memory structure

• **Referential transparency:**
  – Execution of a function always produce the same result when given the same parameters
  – Implication: all variables in a function body must be local to that function; why?

• **Functions can be:**
  – Passed as an argument
  – Returned from a function
  – Represented by a data structure and that data structure can then be evaluated

• **Is this possible?**
**FPL: λ calculus**

- **What is a Calculus?**
  - A method of analysis or calculation using a special symbolic notation
  - E.g.
    - differential calculus and integral calculus (mathematical calculus/calculus)
    - Bi-calculus

- **Defined by Alonzo Church, a logician, in 1930s as a computational theory of recursive functions**

- **λ calculus is equivalent in computational power to Turing machines**

- **Recall: what’s a Turing machine?**
  - Turing machines are abstract machines that emphasizes computation as a series of state transitions driven by symbols on an input tape (which leads naturally to an imperative style of programming based on assignment)

- **How is λ calculus different?**
  - λ calculus emphasizes typed expressions and functions (which naturally leads to a functional style of programming).
  - No state transitions.
FPL: \( \lambda \) calculus – cont’d

- \( \lambda \) calculus is a formal system for defining recursive functions and their properties
  - Expressions in this notation are called \( \lambda \) expressions
  - Every \( \lambda \) expression denotes a function that is "out there"
  - A \( \lambda \) expression consists of 3 kinds of terms:
    - Variables: \( x,y,z \) etc
      - We use \( V \) for arbitrary variables
    - Abstractions: \( \lambda V.E \)
      - where \( V \) is some variable and \( E \) is another \( \lambda \)-term
    - Applications: \( (E_1 E_2) \)
      - where \( E_1 \) and \( E_2 \) are \( \lambda \) terms
      - applications are sometimes called combinations
**FPL: λ calculus & functional forms**

- **A higher-order function (functional form) is one that:**
  - Takes functions as parameters
  - Yields a function as a result
  - E.g.:
    
    \[
    f(x) = x + 2, \quad g(x) = 3 \times x
    \]

    Then, \( h(x) = f(g(x)) \) or \( h(x) = (3 \times x) + 2 \)

    \( h(x) \) is called a higher-order function

- **Types of Functional Forms:**
  - **Construction form:**
    - E.g. \( g(x) = x \times x, \ h(x) = 2 \times x, \ i(x) = x / 2 \)
    - \([g,h,i] (4) = (16,8,2)\)
  - **Apply-to-all form:**
    - E.g. \( h(x) = x \times x \)
    - \( y(h, (2,3,4)) = (4,9,16) \)
Can we represent the class of Turing computable functions?
   - Yes, we can represent:
     • Booleans and conditional functions
     • Numerals and arithmetic functions
     • Data structures, such as ordered pairs, lists, etc…
     • Recursion

Doing so however is syntactically tedious;
   - Actual programming have a lot of syntactic sugar (gives the programmer an alternative way of coding that is more succinct or more like some familiar notation)
   - \( \lambda \) calculus is more suitable as an abstract model of a programming language rather than a practical programming language

Both Turing machines and \( \lambda \) calculus are idealized, mathematical models of computation
Scheme: introduction

• History:

  - Lisp
    - 1950’s
    - John McCarthy
  
  - Scheme
    - 1975
    - Guy Steele
    - Gerald Sussman
  
  - Common Lisp
    - standardized PL

  - dynamic scoping
  
  - lexical scoping
  
  - functions as first class values
  
  - continuations

• **Scheme is based on lambda calculus:** that is the meaning of all syntactic programming constructs in the language are defined in terms of mathematical functions

• A Scheme program consists of **function definitions and calls.** There is no other structure.

• Automatic garbage collection.

• A variable assumes the type of the value that is bound to them at run-time. So, the type of a variable changes dynamically during execution.
Scheme: basic data types

Expression
  Atom
    Symbol
      'Rosebud
    Number
      3
      7/13
      23.45
  Sequence
    List
      (3 4 5 6)
      (+ 12 7 32 78)
      (bob (has (arms 2) (legs 2)))
    String
      "Rosebud"
Scheme: expressions

• An expression in Scheme has the form \((E_1 \ E_2 \ E_3 \ldots \ E_n)\)
  – \(E_1\) evaluates to an operator
  – \(E_2\) through \(E_n\) are evaluated as operands

• Examples:
  – (+ a b c) ; (a + b + c)
  – (+ 1 (* 2 3) 4 5) ; (1 + (2 * 3) + 4 + 5)
  – (+ (- 6 3) (/ 10 2) 2 (* 2 3)) ; 16
  – (<= (- 5 3) (+ 2 (* 3 3)) 14) ; ?
  – (max (+ 2 3) (abs –4) (remainder 12 5)) ; ?

• Postfix v.s. Infix:
  – Scheme expressions use prefix notation while imperative languages use infix notation, which is better?
Scheme: expressions cont’d

• **Literal Expressions:**
  – quote operator
  – E.g.: \( \text{\texttt{\textbf{}} (quote (1 2 3))} \); same as '(1 2 3)
  \((1 2 3)\)
  \(\text{\texttt{\textbf{}} (quote (* 4 5))} \); same as '(* 4 5)
  \((* 4 5)\)

• **define**
  – Introduces a binding of an identifier to some object, which is either a value or an expression
  – E.g.: \( \text{\texttt{\textbf{}} (define size 2)} \)
    \(\text{\texttt{\textbf{}} (define sum ' (+ 1 3 4 5))} \)
  \[=> (\text{\textbf{}} * 2 size)\]
  \[=> (\text{\textbf{}} * 2 2)\]
  \[=> 4\]
  \[=> (\text{\textbf{}} define sum ' (+ 1 3 4 5))\]
  \[=> (\text{\textbf{}} define sum ' (+ 1 3 4 5))\]
  \[=> (\text{\textbf{}} (sum))\]
  \[=> (\text{\textbf{}} (define sum ' (+ 1 3 4 5)))\]
  \[=> (\text{\textbf{}} (sum))\]
  \[=> \text{???}\]
Scheme: evaluating expressions

• **eval FORM**
  – Evaluate ⇔ compute/fetch value of an expression
  – Form ⇔ an expression to be evaluated
  – Rules:
    • A number evaluates to itself
      $76 \Rightarrow 76$
    • A variable evaluates to its value
      `(define x 54) \Rightarrow x = 54$
    • A quoted symbol evaluates to the symbol itself:
      `'z \Rightarrow z`
    • A string evaluates to itself
      `"trondheim" \Rightarrow "trondheim"`
    • A single quoted list evaluates to a simple list of symbols
      `'(+ 2 3) \Rightarrow (+ 2 3)`
    • An unquoted list evaluates to a function call
      `(+ 2 3) \Rightarrow 5`
      `(a b c) \Rightarrow ERROR: attempt to call an undeclared function ‘a`

• **define**
  – E.g.:
    `]=> (define sum '(+ 1 3 4 5))`
    `]=> (sum)`
    `syntax error…`
    `]=> (eval sum (repl/environment (nearest-repl)))`
    `13`
Scheme: expression evaluation order

- Scheme follows a depth-first applicative evaluation order

- Example: \((f1 (f2 a (f3 b)) c 8 (f2 3 2))\)
Scheme: lists

- A list is denoted by a collection of items enclosed in parentheses

- The empty list is denoted ()

- Improper list: a list that does not end with an empty list.

- Lists can be nested
  - Eg. (a (b c) (d) )

- Note: Lists should be quoted when fed to the interpreter, otherwise the interpreter will try to apply the first item in the list to the other items
  - E.g.
    
    ```scheme
    ]=> (2 4 6 8)
    error: procedure application: expected procedure, given: 2;
      arguments were: 4 6 8
    ]=> (quote (2 4 6 8)) ; or ' (2 4 6 8)
    (2 4 6 8)
    ```
Scheme: lists cont’d

• Constructing Lists:
  – *(cons arg1 arg2)*
    • The second argument to *cons* must be a list
    • E.g.
      (cons 'peanut '(butter and jelly)) ; (peanut butter and jelly)
      (cons '(banana and) '(peanut butter and jelly)) ; ((banana and) peanut butter and jelly)

  – *(append arg1 arg2)*
    • Returns the list formed by joining the elements of a and b together.
    • Precondition: arg1 and arg2 must be lists

  – *(list arg1 arg2 ... arg_n)*
    • E.g.
      (list 2 4 6 8 10)
**Scheme: lists cont’d**

- **Internal implementation**
  - Linked list storage management used
  - Head: first member of the list.
  - Tail: everything else other than the head
Scheme: lists cont’d

• **Useful Operations:**
  – (reverse list) ; reverse the order of the elements in list
  – (member element list);
  – (car list) ; return head of the list , pronounced car
  – (cdr list) ; return tail of the list , pronounced coulder
  – (cadr list) ; eqv to (car (cdr list)) , pronounced cahder
  – (cdar list) ; eqv to (cdr (car list)) , pronounced couldaher
  – (caar list) ; eqv to (car (car list)) , pronounced cahar
  – (cddr list) ; eqv to (cdr (cdr list)) , pronounced coulduhder
  – (cadar list) ; eqv to (car (cdr (car list))) , pronounced cahdaher
  – (caadr list) ; eqv to (car (cadr list)) , pronounced cahader
  – (cddddr list) ; eqv to (cdr (cddr list)) , pronounced couldduhduhder
  – (cadadr list) ; eqv to ….  

• **Example:**

\[
\begin{align*}
\text{(car '((a b c)) is a} \\
\text{(car '((a) b (c d))) is (a)} \\
\text{(cdr '((a b c)) is (b c)} \\
\text{(cdr '((a) b (c d))) is (b (c d))}
\end{align*}
\]

\[
\begin{align*}
((a) b (c d))
\end{align*}
\]

\[
\begin{align*}
a & \quad () \\
b & \quad b \\
c & \quad ((a) b (c d)) \\
d & \quad (c d) \\
& \quad () \\
& \quad ()
\end{align*}
\]
Scheme: lists cont’d

- **Examples:**

  ```scheme
  => (define lista '(1 2 5 67 3 2 5 88))
  => (define fruits '(apple pear orange banana))
  => (define colors '(red blue green yellow orange))
  => (define prices '((banana 0.98) (orange 0.33) (lemon 0.20)))
  
  => (car lista)
  1

  => (car colors)
  red

  => (cdr lista)
  (2 5 67 3 2 5 88)

  => (cdr colors)
  (blue green yellow orange)

  => (cadr fruits)
  pear

  => (cadr colors)
  blue

  => (caddr fruits)
  orange

  => (caddr colors)
  blue

  => (cddr fruits)
  (pear)

  => (cddr colors)
  (yellow orange)

  => (cddddr fruits)
  (banana)

  => (car prices)
  (banana 0.98)

  => (caar prices)
  banana

  => (cadar prices)
  0.98
  ```
Scheme: read-evaluate-print Cycle

• **Read input from user:**
  – A function definition or abstraction
  – A function evaluation

• **Evaluate input:**
  – Store function definition (e1 e2 e3 … ek)
  – Evaluate e1 to obtain a function
  – Evaluate e2, … , ek to values
  – Execute function body using values from previous step as formal parameter values
  – Return value of function

• **Print return value**
Scheme: MIT interpreter

1 ]=> (+ 8 3 5 18 9)
;Value: 41

1 ]=> (define increment (lambda (n) (+ n 1)))
;Value: increment

1 ]=> (increment 21)
;Value: 22

1 ]=> (load "incr")
;Loading "incr.scm" -- done
;Value: increment-list

1 ]=> (increment-list (1 32 7))
The object 1 is not applicable.
;To continue, call RESTART with an option number:
; (RESTART 2) => Specify a procedure to use in its place.
; (RESTART 1) => Return to read-eval-print level 1.

2 error> (restart 1)
;Abort!

1 ]=> (increment-list '(1 32 7))
;Value 1: (2 33 8)

1 ]=> (trace increment-list)
;Unspecified return value

1 ]=> (increment-list '(1 32 7))
[Entering #[compound-procedure 2 increment-list]
 Args: (1 32 7)]
[Entering #[compound-procedure 2 increment-list]
 Args: (32 7)]
[Entering #[compound-procedure 2 increment-list]
 Args: (7)]
[Entering #[compound-procedure 2 increment-list]
 Args: ()]
[...]
1 ]=> (increment-list '(1 32 7))
[Entering #[compound-procedure 2 increment-list]
 Args: (7)]
[Entering #[compound-procedure 2 increment-list]
 Args: ()]
[...]
1 ]=> (exit)

Kill Scheme (y or n)? Yes
Happy Happy Joy Joy.
werewolf 2%