CMPUT 325 - Lisp Basics

Dr. B. Price & Dr. R. Greiner

16th September 2004
Lisp History

- History: “LIS<sub>t</sub> Processing” Specified by McCarthy in 1958, but still in use.

- Inspired by Alonzo Church’s abstract theory of computations: lambda calculus in 1930’s
- Second high-level language after Fortran. First Language to support:
  - structured IF_THEN_ELSE_ENDIF
  - dynamic typing of variables, recursion
- Dialects of Lisp: Pure Lisp, Franz Lisp, MacLisp, InterLisp, Common Lisp (now largely standardized on Common Lisp)
- Supports functional, procedural, object-oriented and generic programming
Lisp History


- Inspired by Alonzo Church’s abstract theory of computations: “lambda calculus” in 1930’s
Lisp History

- History: “LIS*t Processing” Specified by McCarthy in 1958, but still in use.
- Inspired by Alonzo Church’s abstract theory of computations: “lambda calculus” in 1930’s
- Second high-level language after Fortran. First Language to support:
  - structured IF_THEN_ELSE_ENDIF
  - dynamic typing of variables, recursion
Lisp History

- **History:** “**List Processing**” Specified by McCarthy in 1958, but still in use.

- Inspired by Alonzo Church’s abstract theory of computations: “lambda calculus” in 1930’s

- Second high-level language after Fortran. First Language to support:
  - structured IF_THEN_ELSE_ENDIF
  - dynamic typing of variables, recursion

- Dialects of Lisp: Pure Lisp, Franz Lisp, MacLisp, InterLisp, Common Lisp (now largely standardized on Common Lisp)
Lisp History

- History: “LIST Processing” Specified by McCarthy in 1958, but still in use.
- Inspired by Alonzo Church’s abstract theory of computations: “lambda calculus” in 1930’s
- Second high-level language after Fortran. First Language to support:
  - structured IF_THENELSE_ENDIF
  - dynamic typing of variables, recursion
- Dialects of Lisp: Pure Lisp, Franz Lisp, MacLisp, InterLisp, Common Lisp (now largely standardized on Common Lisp)
- Supports functional, procedural, object-oriented and generic programming
Lisp is Interactive

ohaton: > gcl
GCL (GNU Common Lisp) Version(2.2)
Licensed under GNU Public Library License ...
Lisp is Interactive

ohaton: > gcl
GCL (GNU Common Lisp) Version(2.2)
Licensed under GNU Public Library License ...
> (+ 11 23)
Lisp is Interactive

ohaton: > gcl
GCL (GNU Common Lisp) Version(2.2)
Licensed under GNU Public Library License ...
> (+ 11 23)
34
Lisp is Interactive

ohaton: > gcl
GCL (GNU Common Lisp) Version(2.2)
Licensed under GNU Public Library License ...
> (+ 11 23)
34
> (SETF x (* 3 4)) {note: procedural!}
Lisp is Interactive

ohaton: > gcl
GCL (GNU Common Lisp) Version(2.2)
Licensed under GNU Public Library License ...
> (+ 11 23)
34
> (SETF x (* 3 4))  {note: procedural!}
12
Lisp is Interactive

ohaton: > gcl
GCL (GNU Common Lisp) Version(2.2)
Licensed under GNU Public Library License ...
> (+ 11 23)
34
> (SETF x (* 3 4)) {note: procedural!}
12
> (+ (* x 2) 5)
Lisp is Interactive

ohaton: > gcl
GCL (GNU Common Lisp) Version(2.2)
Licensed under GNU Public Library License ...
> (+ 11 23)
34
> (SETF x (* 3 4)) {note: procedural!}
12
> (+ (* x 2) 5)
29
Lisp is Interactive

ohaton: > gcl
GCL (GNU Common Lisp) Version(2.2)
Licensed under GNU Public Library License ...
> (+ 11 23)
34
> (SETF x (* 3 4)) {note: procedural!}
12
> (+ (* x 2) 5)
29
> (DEFUN sq (y) (* y y))
Lisp is Interactive

ohaton: > gcl
GCL (GNU Common Lisp) Version(2.2)
Licensed under GNU Public Library License ... 
> (+ 11 23)
34
> (SETF x (* 3 4))   {note: procedural!}
12
> (+ (* x 2) 5)
29
> (DEFUN sq (y) (* y y))
sq
Lisp is Interactive

ohaton: > gcl
GCL (GNU Common Lisp) Version(2.2)
Licensed under GNU Public Library License ...
> (+ 11 23)
34
> (SETF x (* 3 4))  {note: procedural!}
12
> (+ (* x 2) 5)
29
> (DEFUN sq (y) (* y y))
sq
> (sq x)
Lisp is Interactive

ohaton: > gcl
GCL (GNU Common Lisp) Version(2.2)
Licensed under GNU Public Library License ...
> (+ 11 23)
34
> (SETF x (* 3 4)) {note: procedural!}
12
> (+ (* x 2) 5)
29
> (DEFUN sq (y) (* y y))
 sq
> (sq x)
144
Lisp is Interactive

ohaton: > gcl
GCL (GNU Common Lisp) Version(2.2)
Licensed under GNU Public Library License ...
> (+ 11 23)
 34
> (SETF x (* 3 4)) {note: procedural!}
 12
> (+ (* x 2) 5)
 29
> (DEFUN sq (y) (* y y))
  sq
> (sq x)
  144
> (EXIT)
Lecture Note Notation

- Sitting at the Lisp interpreter, things look like this:
  
  \[ \text{> (+ 3 4)} \]
  
  \[7\]

- For compactness in the lecture notes, we write:

  \[(+ 3 4) \rightarrow 7\]
Controlling Evaluation in LISP

- By default, LISP attempts to evaluate the expressions you enter
Controlling Evaluation in LISP

- By default, LISP attempts to evaluate the expressions you enter.
- To enter a constant that should not be evaluated, preface it with a quote.
Controlling Evaluation in LISP

- By default, LISP attempts to evaluate the expressions you enter.

- To enter a constant that should not be evaluated, preface it with a quote.

  \[ y \rightarrow \]
Controlling Evaluation in LISP

- By default, LISP attempts to evaluate the expressions you enter.
- To enter a constant that should not be evaluated, preface it with a quote.

\[ y \rightarrow "Error Undefined!" \]
Controlling Evaluation in LISP

- By default, LISP attempts to evaluate the expressions you enter.

- To enter a constant that should not be evaluated, preface it with a quote.

\[ y \rightarrow "\text{Error Undefined!}" \]
\[ \text{'Y} \rightarrow \]
Controlling Evaluation in LISP

- By default, LISP attempts to evaluate the expressions you enter.

- To enter a constant that should not be evaluated, preface it with a quote.

  \[ y \rightarrow "Error Undefined!" \]
  \[ 'Y \rightarrow Y \]
Controlling Evaluation in LISP

- By default, LISP attempts to evaluate the expressions you enter.

- To enter a constant that should not be evaluated, preface it with a quote:

  \[ y \rightarrow "\text{Error Undefined!}" \]
  \[ 'Y \rightarrow Y \]
  \[ (+ 1 2) \rightarrow \]
Controlling Evaluation in LISP

- By default, LISP attempts to evaluate the expressions you enter.
- To enter a constant that should not be evaluated, preface it with a quote.

\[
\begin{align*}
y & \rightarrow "Error Undefined!" \\
'Y & \rightarrow Y \\
(+ 1 2) & \rightarrow 3
\end{align*}
\]
Controlling Evaluation in LISP

- By default, LISP attempts to evaluate the expressions you enter.

- To enter a constant that should not be evaluated, preface it with a quote.

\[ y \rightarrow "Error Undefined!" \]
\[ 'Y \rightarrow Y \]
\[ (+ 1 2) \rightarrow 3 \]
\[ '(+ 1 2) \rightarrow \]
Controlling Evaluation in LISP

- By default, LISP attempts to evaluate the expressions you enter.

- To enter a constant that should not be evaluated, preface it with a quote.

  \[y \rightarrow "Error Undefined!"\]
  \[\text{'}Y \rightarrow Y\]
  \[+ 1 2 \rightarrow 3\]
  \[\text{'}(+ 1 2) \rightarrow (+ 1 2)\]
Lisp's Data Structures

- In "pure" *Lisp* all compound data is represented by “symbolic expressions” (called *s-expressions*, or *s-exprs*)
Lisp’s Data Structures

- In "pure" Lisp all compound data is represented by “symbolic expressions” (called s-expressions, or s-exprs)

- An s-expr is either
  - an Atom (e.g., 1 able nil t 3.4)
  - or a List of s-exprs (e.g., (1 2 3))
Lisp’s Data Structures

- In "pure" *Lisp* all compound data is represented by "symbolic expressions" (called *s-expressions*, or *s-exprs*)

- An *s-expr* is either
  - an *Atom* (e.g., `1 able nil t 3.4`)
  - or a List of *s-exprs* (e.g., `(1 2 3)``

- An *Atom* is a number, or a string of 1 or more letters or digits.
  - e.g. `g`  `u-of-a`
  - `e.g. 24`  `cdr`
  - `1a2b`  `8088`
Lisp’s Data Structures

- In "pure" *Lisp* all compound data is represented by “symbolic expressions” (called *s-expressions*, or *s-exprs*)

- An *s-expr* is either
  - an *Atom* (e.g., `1 able nil t 3.4`)
  - or a List of s-exprs (e.g., `(1 2 3)`)  

- An *Atom* is a number, or a string of 1 or more letters or digits.
  
  e.g. `g u-of-a`
  
  `24 cdr`
  
  `1a2b 8088`

- Modern Lisp’s also implement vectors, hash tables, arrays, various types of numbers and even objects
Lists in LISP

- Def’n: A list is 0 or more s-exprs enclosed in parentheses.
Lists in LISP

- Def’n: A list is 0 or more s-exprs enclosed in parentheses.

  (a b c)

  ()

- Examples

  (+ 2 3)

  (plus x (times y 3))

  ( () () )
Lists in LISP

- **Def’n:** A *list* is 0 or more s-exprs enclosed in parentheses.
  
  \[(a\ b\ c)\]
  
  \[
  ()
  \]

- **Examples**
  
  \[(+\ 2\ 3)\]
  
  \[(\text{plus}\ x\ (\text{times}\ y\ 3))\]
  
  \[
  (()())
  \]

- **Generally:** \[(s_1\ s_2\ \cdots\ s_n),\ n \geq 0,\ s_i\ are\ s-exprs\]
Lists in LISP

► Def’n: A list is 0 or more s-exprrs enclosed in parentheses.

(a b c)
()

► Examples
(+ 2 3)
(plus x (times y 3))
( () () )

► Generally: \((s_1 \ s_2 \ \cdots \ s_n)\), \(n \geq 0\), \(s_i\) are s-exprrs

► Special case: () is the empty list. Also called nil. [It is both an atom and a list.]
Properties of Lists

▶ The size of a list does not have to be declared in advance.
Properties of Lists

- The size of a list does not have to be declared in advance.

- Differs from set (why?):
  - Multiple instances of the same element are allowed.
  - Elements are in a strict order.
  - Examples: `(bac b)`, `(1 2 3)`, `(1 2 (3 4) 5 () )`
Properties of Lists

- The size of a list does not have to be declared in advance.
- Differs from set (why?):
  - can contain multiple instances of the same element
Properties of Lists

- The size of a list does not have to be declared in advance.

- Differs from set (why?):
  - can contain multiple instances of the same element
  - elements are in a strict order.
Properties of Lists

- The size of a list does not have to be declared in advance.

- Differs from set (why?):
  - can contain multiple instances of the same element
  - elements are in a strict order.

(b a c b)

- Eg Lists
  (1 2 3)
  (1 2 (3 4) 5 () )
Building up Compound s-expr

- CONS builds an s-expr from 2 others

\[(\text{CONS} \ 1 \ 2) \rightarrow (1 . 2)\]
Building up Compound s-expr

- CONS builds an s-expr from 2 others
  \[(\text{CONS } 1 \ 2) \rightarrow (1 . \ 2)\]
Building up Compound s-expr

- CONS builds an s-expr from 2 others
  - \[(\text{CONS} \ 1 \ 2) \rightarrow (1 \ . \ 2)\]
- The 2 item result is also known as a "CONS" cell
Building up Compound s-expr

- CONS builds an s-expr from 2 others
  \[(\text{CONS} \ 1 \ 2) \rightarrow (1 \ . \ 2)\]

- The 2 item result is also known as a "CONS" cell

- If the second argument is a list, Lisp displays the result as a list
  \[(\text{CONS} \ 1 \ '(2 \ 3)) \rightarrow\]
Building up Compound s-expr

- CONS builds an s-expr from 2 others
  \[(\text{CONS} \ 1 \ 2) \rightarrow (1 \ . \ 2)\]
- The 2 item result is also known as a "CONS" cell
- If the second argument is a list, Lisp displays the result as a list
  \[(\text{CONS} \ 1 \ '((2 \ 3)) \rightarrow '(1 \ . \ (2 \ 3)) \equiv (1 \ 2 \ 3)\]
Building up Compound s-expr

- CONS builds an s-expr from 2 others
  \[(CONS \ 1 \ 2) \rightarrow (1 \ . \ 2)\]
- The 2 item result is also known as a "CONS" cell
- If the second argument is a list, Lisp displays the result as a list
  \[(CONS \ 1 \ '(2 \ 3)) \rightarrow \'(1 \ . \ (2 \ 3)) \equiv (1 \ 2 \ 3)\]
- When the second argument is "nil" (a.k.a. empty list) we get a singleton
  \[(CONS \ 1 \ nil) \rightarrow\]
Building up Compound s-expr

- CONS builds an s-expr from 2 others
  \[(CONS \, 1 \, 2) \rightarrow (1 . 2)\]

- The 2 item result is also known as a "CONS" cell

- If the second argument is a list, Lisp displays the result as a list
  \[(CONS \, 1 \, '(2 \, 3)) \rightarrow '(1 . (2 \, 3)) \equiv (1 \, 2 \, 3)\]

- When the second argument is "nil" (a.k.a empty list) we get a singleton
  \[(CONS \, 1 \, \text{nil}) \rightarrow (1 \, . \, \text{nil}) \equiv (1)\]
Building up Compound s-expr

- CONS builds an s-expr from 2 others
  \[(\text{CONS} \ 1 \ 2) \rightarrow (1 \ . \ 2)\]

- The 2 item result is also known as a "CONS" cell

- If the second argument is a list, Lisp displays the result as a list
  \[(\text{CONS} \ 1 \ '(2 \ 3)) \rightarrow '(1 \ . \ (2 \ 3)) \equiv (1 \ 2 \ 3)\]

- When the second argument is "nil" (a.k.a. empty list) we get a singleton
  \[(\text{CONS} \ 1 \ \text{nil}) \rightarrow (1 \ . \ \text{nil}) \equiv (1)\]
  \[(\text{CONS} \ 1 \ ()) \rightarrow\]
Building up Compound s-expr

- CONS builds an s-expr from 2 others
  
  \[(CONS \ 1 \ 2) \rightarrow (1 \ . \ 2)\]

- The 2 item result is also known as a "CONS" cell

- If the second argument is a list, Lisp displays the result as a list
  
  \[(CONS \ 1 \ '(2 \ 3)) \rightarrow ' (1 \ . \ (2 \ 3)) \equiv (1 \ 2 \ 3)\]

- When the second argument is "nil" (a.k.a empty list) we get a singleton

  \[(CONS \ 1 \ nil) \rightarrow (1 \ . \ nil) \equiv (1)\]

  \[(CONS \ 1 \ ()) \rightarrow (1)\]
Building up Compound s-expr

- CONS builds an s-expr from 2 others
  
  \((\text{CONS } 1 \ 2) \rightarrow (1 . \ 2)\)

- The 2 item result is also known as a "CONS" cell

- If the second argument is a list, Lisp displays the result as a list
  
  \((\text{CONS } 1 \ '(2 \ 3)) \rightarrow ' (1 . (2 \ 3)) \equiv (1 \ 2 \ 3)\)

- When the second argument is "nil" (a.k.a empty list) we get a singleton
  
  \((\text{CONS } 1 \ \text{nil}) \rightarrow (1 . \ \text{nil}) \equiv (1)\)
  \((\text{CONS } 1 \ ()) \rightarrow (1)\)

In general

\((\text{CONS } s0 \ (s1 \ \ldots \ sn))\)

\(\rightarrow\)
Building up Compound s-expr

- CONS builds an s-expr from 2 others
  \[(\text{CONS } 1 \ 2) \rightarrow (1 \ . \ 2)\]
- The 2 item result is also known as a "CONS" cell
- If the second argument is a list, Lisp displays the result as a list
  \[(\text{CONS } 1 \ '(2 \ 3)) \rightarrow ' (1 \ . \ (2 \ 3)) \equiv (1 \ 2 \ 3)\]
- When the second argument is "nil" (a.k.a empty list) we get a singleton
  \[(\text{CONS } 1 \ \text{nil}) \rightarrow (1 \ . \ \text{nil}) \equiv (1)\]
  \[(\text{CONS } 1 \ ()) \rightarrow (1)\]

In general
\[(\text{CONS } s0 \ (s1 \ldots sn)) \rightarrow (s0 \ . \ (s1 \ldots sn)) \equiv (s0 \ s1 \ldots sn)\]
Taking an s-expr apart

- Given $s_0 \equiv (s_1 . s_2)$, where $s_1$ and $s_2$ are expressions, CAR returns first element

$(\text{CAR } (s_1 . s_2)) \rightarrow$
Taking an s-expr apart

- Given $s_0 \equiv (s_1 \ . \ s_2)$, where $s_1$ and $s_2$ are expressions, CAR returns first element

$$\text{(CAR } '(s_1 \ . \ s_2)) \rightarrow s_1$$
Taking an s-expr apart

▷ Given \( s_0 \equiv (s_1 . s_2) \), where \( s_1 \) and \( s_2 \) are expressions, CAR returns first element

\[
(CAR ' (s_1 . s_2)) \rightarrow s_1
\]

▷ The CDR function returns the second element

\[
(CDR ' (s_1 . s_2)) \rightarrow
\]
Taking an s-expr apart

- Given \( s_0 \equiv ( s_1 \ . \ s_2 ) \), where \( s_1 \) and \( s_2 \) are expressions, CAR returns first element
  \[
  (\text{CAR } '(s_1 \ . \ s_2)) \rightarrow s_1
  \]

- The CDR function returns the second element
  \[
  (\text{CDR } '(s_1 \ . \ s_2)) \rightarrow s_2
  \]
Taking an s-expr apart

- Given \( s_0 \equiv (s_1 . s_2) \), where \( s_1 \) and \( s_2 \) are expressions, \( \text{CAR} \) returns first element
  \[
  (\text{CAR} ' (s_1 . s_2)) \rightarrow s_1
  \]
- The \( \text{CDR} \) function returns the second element
  \[
  (\text{CDR} ' (s_1 . s_2)) \rightarrow s_2
  \]
- Recall, a list is a CONS cell whose second element is a list
  \[
  (s_1 s_2 s_3 \ldots s_n) \equiv (s_1 . (s_2 s_3 \ldots s_n))
  \]
Taking an s-expr apart

▶ Given \( s_0 \equiv (s_1 \ . \ s_2) \), where \( s_1 \) and \( s_2 \) are expressions, \textsc{car} returns first element
\[
(CAR \ '(s_1 \ . \ s_2)) \rightarrow s_1
\]

▶ The \textsc{cdr} function returns the second element
\[
(CDR \ '(s_1 \ . \ s_2)) \rightarrow s_2
\]

▶ Recall, a list is a \textsc{cons} cell whose second element is a list
\[
(s_1 \ s_2 \ s_3 \ \ldots \ s_n) \equiv (s_1 \ . \ (s_2 \ s_3 \ \ldots \ s_n))
\]

▶ The \textsc{car} function returns the "first" element of the list
\[
(CAR \ '(s_1 \ s_2 \ s_3 \ \ldots \ s_n)) \rightarrow
\]
Taking an s-expr apart

- Given $s_0 \equiv (s_1 . s_2)$, where $s_1$ and $s_2$ are expressions, CAR returns first element

  $$(\text{CAR } '(s_1 . s_2)) \rightarrow s_1$$

- The CDR function returns the second element

  $$(\text{CDR } '(s_1 . s_2)) \rightarrow s_2$$

- Recall, a list is a CONS cell whose second element is a list

  $$(s_1 s_2 s_3 \ldots s_n) \equiv (s_1 . (s_2 s_3 \ldots s_n))$$

- The CAR function returns the "first" element of the list

  $$(\text{CAR } '(s_1 s_2 s_3 \ldots s_n)) \rightarrow s_1$$
Taking an s-expr apart

▶ Given $s_0 \equiv (s_1 . s_2)$, where $s_1$ and $s_2$ are expressions, CAR returns first element

$$(\text{CAR } '(s_1 . s_2)) \rightarrow s_1$$

▶ The CDR function returns the second element

$$(\text{CDR } '(s_1 . s_2)) \rightarrow s_2$$

▶ Recall, a list is a CONS cell whose second element is a list

$$(s_1 s_2 s_3 \ldots s_n) \equiv (s_1 . (s_2 s_3 \ldots s_n))$$

▶ The CAR function returns the "first" element of the list

$$(\text{CAR } '(s_1 s_2 s_3 \ldots s_n)) \rightarrow s_1$$

▶ The CDR function returns the "rest" of the list

$$(\text{CDR } '(s_1 s_2 s_3 \ldots s_n)) \rightarrow$$
Taking an s-expr apart

- Given \( s_0 \equiv (s_1 . s_2) \), where \( s_1 \) and \( s_2 \) are expressions, \texttt{CAR} returns first element
  
  \[
  \text{\texttt{(\texttt{CAR} '(s_1 . s_2)) \rightarrow s_1}}
  \]

- The \texttt{CDR} function returns the second element
  
  \[
  \text{\texttt{(\texttt{CDR} '(s_1 . s_2)) \rightarrow s_2}}
  \]

- Recall, a list is a CONS cell whose second element is a list
  
  \[
  (s_1 \ s_2 \ s_3 \ldots s_n) \equiv (\ s_1 \ . \ (s_2 \ s_3 \ldots s_n) \ )
  \]

- The \texttt{CAR} function returns the "first" element of the list
  
  \[
  \text{\texttt{(\texttt{CAR} '(s_1 s_2 s_3 \ldots s_n)) \rightarrow s_1}}
  \]

- The \texttt{CDR} function returns the "rest" of the list
  
  \[
  \text{\texttt{(\texttt{CDR} '(s_1 s_2 s_3 \ldots s_n)) \rightarrow (s_2 s_3 \ldots s_n)}}
  \]
Why ’car’ and ’cdr’

▶ Historical:

Historical:

The IBM 704 divided words into “address” and “decrement”.

CAR was an assembly instruction to extract the “Contents of the Address Register”, and CDR extracted the “Contents of the Decrement Register”.

The name of Lisp’s core functions derive from low-level assembly instructions!
Why 'car' and 'cdr'

- Historical:
  - The IBM 704 divided words into "address" and "decrement"
Why 'car' and 'cdr'

- **Historical:**
  - The IBM 704 divided words into "address" and "decrement"
  - CAR was assembly instruction to extract the "Contents of the Address Register", and
Why ’car’ and ’cdr’

- **Historical:**
  - The IBM 704 divided words into "address" and "decrement"
  - CAR was assembly instruction to extract the "Contents of the Address Register", and
  - CDR extracted the "Contents of the Decrement Register".
Why 'car' and 'cdr'

- Historical:
  - The IBM 704 divided words into "address" and "decrement"
  - CAR was assembly instruction to extract the "Contents of the Address Register", and
  - CDR extracted the "Contents of the Decrement Register".
  - The name of Lisp’s core functions derive from low-level assembly instructions!
FIRST, REST and the Rest

- CAR, CDR are traditional, but modern FIRST and REST are more readable
FIRST, REST and the Rest

- CAR, CDR are traditional, but modern FIRST and REST are more readable

(FIRST '(A B C)) →
FIRST, REST and the Rest

- CAR, CDR are traditional, but modern FIRST and REST are more readable

\[(\text{FIRST } '(A \ B \ C)) \rightarrow A\]
FIRST, REST and the Rest

- CAR, CDR are traditional, but modern FIRST and REST are more readable

\[
\begin{align*}
\text{(FIRST ' (A B C))} & \rightarrow \text{A} \\
\text{(REST ' (A B C))} & \rightarrow \\
\end{align*}
\]
FIRST, REST and the Rest

- CAR, CDR are traditional, but modern FIRST and REST are more readable

  \[(\text{FIRST } '(A B C)) \rightarrow A\]
  \[(\text{REST } '(A B C)) \rightarrow (B C)\]
FIRST, REST and the Rest

- CAR, CDR are traditional, but modern FIRST and REST are more readable
  
  \[(\text{FIRST } '(A \text{ B} \text{ C})) \rightarrow A\]
  
  \[(\text{REST } '(A \text{ B} \text{ C})) \rightarrow (\text{B C})\]

- The CAR and CDR functions can be composed
  
  \[(\text{CAR (CDR } '(\text{A (B) C})) \rightarrow\]
FIRST, REST and the Rest

- CAR, CDR are traditional, but modern FIRST and REST are more readable
  \[(\text{FIRST } '(A B C)) \rightarrow A\]
  \[(\text{REST } '(A B C)) \rightarrow (B C)\]
- The CAR and CDR functions can be composed
  \[(\text{CAR (CDR } '(A (B) C)) \rightarrow (B)\]
FIRST, REST and the Rest

▶ CAR, CDR are traditional, but modern FIRST and REST are more readable

\[
( \text{FIRST } '(A \ B \ C)) \rightarrow A \\
( \text{REST } '(A \ B \ C)) \rightarrow (B \ C)
\]

▶ The CAR and CDR functions can be composed

\[
( \text{CAR (CDR } '(A (B) C)) \rightarrow (B)
\]

▶ LISP provides abbreviations for common sequences of accessors

\[
( \text{CAR (CDR } '(A B C)) \equiv (\text{CADR } '(A B C))
\]
FIRST, REST and the Rest

- CAR, CDR are traditional, but modern FIRST and REST are more readable

\[
\begin{align*}
(FIRST '(A B C)) & \rightarrow A \\
(REST '(A B C)) & \rightarrow (B C)
\end{align*}
\]

- The CAR and CDR functions can be composed

\[
(CAR (CDR '(A (B) C))) \rightarrow (B)
\]

- LISP provides abbreviations for common sequences of accessors

\[
(CAR (CDR '(A B C))) \equiv (CADR '(A B C))
\]

- Any combination of CAR’s and CDR’s are defined up to 10

\[
(CAADDR '(A B (C) D )) \rightarrow
\]
FIRST, REST and the Rest

- CAR, CDR are traditional, but modern FIRST and REST are more readable
  
  (FIRST '(A B C)) → A
  (REST '(A B C)) → (B C)

- The CAR and CDR functions can be composed
  
  (CAR (CDR '(A (B) C))) → (B)

- LISP provides abbreviations for common sequences of accessors
  
  (CAR (CDR '(A B C))) ≡ (CADR '(A B C))

- Any combination of CAR’s and CDR’s are defined up to 10
  
  (CAADDR '(A B (C) D )) → C
FIRST, REST and the Rest

- CAR, CDR are traditional, but modern FIRST and REST are more readable
  
  (FIRST '(A B C)) → A
  (REST '(A B C)) → (B C)

- The CAR and CDR functions can be composed
  
  (CAR (CDR '(A B C))) → (B)

- LISP provides abbreviations for common sequences of accessors
  
  (CAR (CDR '(A B C))) ≡ (CADR '(A B C))

- Any combination of CAR’s and CDR’s are defined up to 10
  
  (CAADDR '(A B (C) D)) → C

- Additional accessors: FIRST, SECOND, ..., TENTH
Examples of CONS usage

\[(\text{CONS } 1 \ 2) \rightarrow\]
Examples of CONS usage

(CONS 1 2) \rightarrow (1 . 2)
Examples of CONS usage

(CONS 1 2) → (1 . 2)
(CONS 1 nil) →
Examples of CONS usage

(CONS 1 2) → (1 . 2)
(CONS 1 nil) → (1)
Examples of CONS usage

(CONS 1 2) → (1 . 2)
(CONS 1 nil) → (1)
(CONS 1 '(2 3)) →
Examples of CONS usage

\[(\text{CONS} \ 1 \ 2) \rightarrow (1 . 2)\]

\[(\text{CONS} \ 1 \ \text{nil}) \rightarrow (1)\]

\[(\text{CONS} \ 1 \ '(2 \ 3)) \rightarrow (1 \ 2 \ 3)\]
Examples of CONS usage

\[
(\text{CONS } 1 \ 2) \rightarrow (1 \ . \ 2) \\
(\text{CONS } 1 \ \text{nil}) \rightarrow (1) \\
(\text{CONS } 1 \ ' (2 \ 3)) \rightarrow (1 \ 2 \ 3) \\
(\text{CONS } ' (1 \ 2) \ \text{nil}) \rightarrow
\]
Examples of CONS usage

\[(\text{CONS} \ 1 \ 2) \rightarrow (1 \ . \ 2)\]
\[(\text{CONS} \ 1 \ \text{nil}) \rightarrow (1)\]
\[(\text{CONS} \ 1 \ ' (2 \ 3)) \rightarrow (1 \ 2 \ 3)\]
\[(\text{CONS} \ ' (1 \ 2) \ \text{nil}) \rightarrow ((1 \ 2))\]
Examples of CONS usage

(CONS 1 2) → (1 . 2)
(CONS 1 nil) → (1)
(CONS 1 '(2 3)) → (1 2 3)
(CONS '(1 2) nil) → ((1 2) nil)
(CONS '(1 2) '(2 3)) →
Examples of CONS usage

\[(\text{CONS} \ 1 \ 2) \rightarrow (1 \ . \ 2)\]
\[(\text{CONS} \ 1 \ \text{nil}) \rightarrow (1)\]
\[(\text{CONS} \ 1 \ '(2 \ 3)) \rightarrow (1 \ 2 \ 3)\]
\[(\text{CONS} \ '(1 \ 2) \ \text{nil}) \rightarrow (1 \ 2)\]
\[(\text{CONS} \ '(1 \ 2) \ '(2 \ 3)) \rightarrow ((1 \ 2) \ 2 \ 3)\]
Examples of CONS usage

(CONS 1 2) → (1 . 2)
(CONS 1 nil) → (1)
(CONS 1 '(2 3)) → (1 2 3)
(CONS '(1 2) nil) → ((1 2))
(CONS '(1 2) '(2 3)) → ((1 2) 2 3)
(CONS nil 1)
Examples of CONS usage

(CONS 1 2) → (1 . 2)
(CONS 1 nil) → (1)
(CONS 1 '(2 3)) → (1 2 3)
(CONS '(1 2) nil) → (1 2)
(CONS '(1 2) '(2 3)) → ((1 2) 2 3)
(CONS nil 1) → (nil . 1)
Examples of CONS usage

(CONS 1 2) \rightarrow (1 . 2)  
(CONS 1 nil) \rightarrow (1)  
(CONS 1 '(2 3)) \rightarrow (1 2 3)  
(CONS '(1 2) nil) \rightarrow ((1 2))  
(CONS '(1 2) '(2 3)) \rightarrow ((1 2) 2 3)  
(CONS nil 1) \rightarrow (nil . 1)  
(CONS (CONS '(1 2) '(3 4)) '(5 6)) \rightarrow
Examples of CONS usage

\[(\text{CONS} \ 1 \ 2) \rightarrow (1 \ . \ 2)\]
\[(\text{CONS} \ 1 \ \text{nil}) \rightarrow (1)\]
\[(\text{CONS} \ 1 \ '(2 \ 3)) \rightarrow (1 \ 2 \ 3)\]
\[(\text{CONS} \ '(1 \ 2) \ \text{nil}) \rightarrow ((1 \ 2))\]
\[(\text{CONS} \ '(1 \ 2) \ '(2 \ 3)) \rightarrow ((1 \ 2) \ 2 \ 3)\]
\[(\text{CONS} \ \text{nil} \ 1) \rightarrow (\text{nil} \ . \ 1)\]
\[(\text{CONS} \ (\text{CONS} \ '(1 \ 2) \ '(3 \ 4)) \ '(5 \ 6)) \rightarrow ((1 \ 2) \ 3 \ 4) \ 5 \ 6)\]
Predicates

- *Predicate* $\equiv$ A Boolean-Valued Function
Predicates

- *Predicate* $\equiv$ A Boolean-Valued Function

- Returns: True or False
Predicates

- *Predicate* ≡ A Boolean-Valued Function
- Returns: True or False
- The *Lisp* representation:
  - NIL ≡ False
  - Any non-NIL s-expression ≡ True
Predicates

- **Predicate**: A Boolean-Valued Function
- Returns: True or False
- The *Lisp* representation:
  - NIL $\equiv$ False
  - Any non-NIL s-expression $\equiv$ True
- Conventionally, $t$ is used to represent true
  (Note $t$ is a non-NIL atom).
Predicates

- **Predicate**: ≡ A Boolean-Valued Function

- Returns: True or False

- The *Lisp* representation:
  
  • NIL ≡ False
  
  • Any non-NIL s-expression ≡ True

- Conventionally, t is used to represent true
  (Note t is a non-NIL atom).

- When the atom t is returned by LISP, it is displayed "T".
Predicates: ATOM

ATOM tests if \textit{s-expr} is atomic.

Examples:

\[(\text{ATOM } \text{'}a\text{')}\rightarrow\]
Predicates: ATOM

ATOM tests if s-exp is atomic.

Examples:

(ATOM 'a) → T
Predicates: ATOM

- ATOM tests if *s-expr* is atomic.

Examples:

(ATOM 'a) → T
(ATOM '(a)) →
Predicates: ATOM

- ATOM tests if *s-expr* is atomic.

Examples:

\[(ATOM \ 'a) \rightarrow T\]
\[(ATOM \ '(a)) \rightarrow NIL\]
Predicates: ATOM

- ATOM tests if \texttt{s-expr} is atomic.

Examples:

- \( \text{(ATOM 'a)} \rightarrow \text{T} \)
- \( \text{(ATOM '(a))} \rightarrow \text{NIL} \)
- \( \text{(SETQ a '(1 2))} \)
- \( \text{(ATOM a)} \)

Predicates: ATOM

ATOM tests if s-expr is atomic.

Examples:

(ATOM 'a) → T
(ATOM '(a)) → NIL
(SETQ a '(1 2))
(ATOM a) → NIL
Predicates: ATOM

- ATOM tests if s-exp is atomic.

Examples:

(ATOM 'a) → T
(ATOM '(a)) → NIL
(SEQT a '(1 2))
(ATOM a) → NIL
(ATOM 1) →
Predicates: ATOM

ATOM tests if s-expr is atomic.

Examples:

(ATOM 'a) → T
(ATOM '(a)) → NIL
(SETQ a '(1 2))
(ATOM a) → NIL
(ATOM 1) → T
Predicates: ATOM

ATOM tests if s-expr is atomic.

Examples:

(ATOM ’a) → T
(ATOM ’(a)) → NIL
(SETQ a ’(1 2))
(ATOM a) → NIL
(ATOM 1) → T
(ATOM ’( 1 . 2 )) →
Predicates: ATOM

ATOM tests if s-expr is atomic.

Examples:

(ATOM 'a) → T
(ATOM '(a)) → NIL
(SETQ a '(1 2))
(ATOM a ) → NIL
(ATOM 1) → T
(ATOM '( 1 . 2 ) ) → NIL
Predicates: ATOM

ATOM tests if s-expr is atomic.

Examples:

(ATOM ’a) → T
(ATOM ’(a)) → NIL
(SEQT a ’(1 2))
(ATOM a ) → NIL
(ATOM 1) → T
(ATOM ’( 1 . 2 )) → NIL
(ATOM nil ) →
Predicates: ATOM

ATOM tests if s-expr is atomic.

Examples:

(ATOM 'a) → T
(ATOM '(a)) → NIL
(SETQ a '(1 2))
(ATOM a ) → NIL
(ATOM 1) → T
(ATOM '( 1 . 2 )) → NIL
(ATOM nil ) → T
Predicates: LISTP

- LISTP tests if an \textit{s-expr} is a cons cell or the empty list.

Examples

\[(\text{LISTP nil}) \rightarrow \text{T}\]

\[(\text{LISTP 'a}) \rightarrow \text{NIL}\]

\[(\text{LISTP 1}) \rightarrow \text{NIL}\]

\[(\text{LISTP '(1 . 2)}) \rightarrow \text{T}\]

\[(\text{LISTP (CONS 1 2)}) \rightarrow \text{T}\]

\[(\text{LISTP b}) \rightarrow \text{"Error!"}\]
Predicates: LISTP

- LISTP tests if an s-expr is a cons cell or the empty list.

Examples

\[(\text{LISTP nil }) \rightarrow \text{T}\]
Predicates: LISTP

- LISTP tests if an s-expr is a cons cell or the empty list.

Examples

(LISTP nil) → T
(LISTP 'a) →
Predicates: LISTP

- LISTP tests if an s-expr is a cons cell or the empty list.

Examples

- (LISTP nil) → T
- (LISTP 'a) → NIL
Predicates: LISTP

- LISTP tests if an s-expr is a cons cell or the empty list.

Examples

- `(LISTP nil) → T``
- `(LISTP 'a) → NIL``
- `(LISTP 1) →`
Predicates: LISTP

LISTP tests if an s-expr is a cons cell or the empty list.

Examples

- (LISTP nil ) → T
- (LISTP ’a ) → NIL
- (LISTP 1 ) → NIL
Predicates: LISTP

- LISTP tests if an s-expr is a cons cell or the empty list.

Examples

- (LISTP nil) → T
- (LISTP 'a) → NIL
- (LISTP 1) → NIL
- (LISTP '(1 2)) →
Predicates: LISTP

- LISTP tests if an \textit{s-expr} is a cons cell or the empty list.

Examples

- (LISTP nil) $\rightarrow$ T
- (LISTP 'a) $\rightarrow$ NIL
- (LISTP 1) $\rightarrow$ NIL
- (LISTP '(1 2)) $\rightarrow$ T
Predicates: LISTP

- LISTP tests if an s-expr is a cons cell or the empty list.

Examples

- \((\text{LISTP \ nil }) \rightarrow \text{T}\)
- \((\text{LISTP \ 'a }) \rightarrow \text{NIL}\)
- \((\text{LISTP \ 1 }) \rightarrow \text{NIL}\)
- \((\text{LISTP \ '(1 \ 2)}) \rightarrow \text{T}\)
- \((\text{LISTP \ '(1 \ . \ 2)}) \rightarrow\)
Predicates: LISTP

- LISTP tests if an s-expr is a cons cell or the empty list.

Examples

(LISTP nil) → T
(LISTP 'a) → NIL
(LISTP 1) → NIL
(LISTP '(1 2)) → T
(LISTP '(1 . 2)) → T
Predicates: LISTP

- LISTP tests if an s-expr is a cons cell or the empty list.

Examples

(LISTP nil) → T
(LISTP 'a) → NIL
(LISTP 1) → NIL
(LISTP '(1 2)) → T
(LISTP '(1 . 2)) → T
(LISTP (CONS 1 2)) →
Predicates: LISTP

- LISTP tests if an s-exp is a cons cell or the empty list.

Examples

- `(LISTP nil ) → T`
- `(LISTP ’a ) → NIL`
- `(LISTP 1 ) → NIL`
- `(LISTP ’(1 2)) → T`
- `(LISTP ’(1 . 2)) → T`
- `(LISTP (CONS 1 2)) → T`
Predicates: LISTP

- LISTP tests if an s-expr is a cons cell or the empty list.

Examples

- `(LISTP nil) → T`
- `(LISTP 'a) → NIL`
- `(LISTP 1) → NIL`
- `(LISTP '(1 2)) → T`
- `(LISTP '(1 . 2)) → T`
- `(LISTP (CONS 1 2)) → T`
- `(LISTP b)`
Predicates: LISTP

- LISTP tests if an s-expr is a cons cell or the empty list.

Examples

- (LISTP nil )→ T
- (LISTP 'a )→ NIL
- (LISTP 1 )→ NIL
- (LISTP '(1 2))→ T
- (LISTP '(1 . 2))→ T
- (LISTP (CONS 1 2))→ T
- (LISTP b)→ "Error!"
Predicate: EQUAL

- Compares two s-expr for equality: "having the same value"

(EQUAL 1 1) →
Predicate: EQUAL

- Compares two s-expr for equality:
  "having the same value"

\[(\text{EQUAL} \ 1\ 1) \rightarrow \text{T}\]
Predicate: **EQUAL**

- Compares two s-expr for equality: "having the same value"

  - $(\text{EQUAL} \ 1 \ 1) \rightarrow \ T$
  - $(\text{EQUAL} \ 1 \ 2) \rightarrow \ nil$
  - $(\text{EQUAL} \ \text{nil} \ \text{CDR} \ '(\text{a})) \rightarrow \ T$
  - $(\text{EQUAL} \ '(\text{a} \ \text{b}) \ '(\text{a} \ \text{b})) \rightarrow \ T$
Predicate: EQUAL

- Compares two s-expr for equality: "having the same value"

  \( (\text{EQUAL} \ 1 \ 1) \rightarrow \text{T} \)
  \( (\text{EQUAL} \ 1 \ 2) \rightarrow \text{nil} \)
Predicate: EQUAL

- Compares two s-expr for equality: "having the same value"

(EQUAL 1 1) → T
(EQUAL 1 2) → nil
(EQUAL nil (CDR '(a))) →
Predicate: \texttt{EQUAL}

- Compares two s-expr for equality: "having the same value"

\begin{align*}
\text{(EQUAL 1 1)} & \rightarrow T \\
\text{(EQUAL 1 2)} & \rightarrow \text{nil} \\
\text{(EQUAL nil (CDR '(a))} & \rightarrow T
\end{align*}
Predicate: EQUAL

- Compares two s-expr for equality: "having the same value"

(EQUAL 1 1) → T
(EQUAL 1 2) → nil
(EQUAL nil (CDR '(a))) → T
(EQUAL '(a b) '(a b)) →
Predicate: EQUAL

▶ Compares two s-expr for equality: "having the same value"

(EQUAL 1 1) → T
(EQUAL 1 2) → nil
(EQUAL nil (CDR '(a))) → T
(EQUAL '(a b) '(a b)) → T
Predicate: EQ

- EQ compares two atoms for equivalence: "sharing the same representation"

\[(\text{EQ} \ 1 \ 1) \rightarrow\]
Predicate: EQ

- EQ compares two atoms for equivalence: "sharing the same representation"

(EQ 1 1) → T
Predicate: EQ

- EQ compares two atoms for equivalence:
  "sharing the same representation"

\[
\begin{align*}
(EQ \ 1 \ 1) & \rightarrow T \\
(EQ \ '(1 \ 2) \ '(1 \ 2)) & \rightarrow
\end{align*}
\]
Predicate: \texttt{EQ}

\begin{itemize}
  \item \texttt{EQ} compares two atoms for equivalence: "sharing the same representation"
  \begin{align*}
  \text{(EQ 1 1)} & \rightarrow \text{T} \\
  \text{(EQ '}(1 \ 2) \ '}(1 \ 2)) & \rightarrow \text{nil}
  \end{align*}
\end{itemize}
Predicate: EQ

- EQ compares two atoms for equivalence: "sharing the same representation"

\[
\begin{align*}
(EQ \ 1 \ 1) & \rightarrow \ T \\
(EQ \ '(1 \ 2) \ '(1 \ 2)) & \rightarrow \ nil \\
(EQUAL \ '(1 \ 2) \ '(1 \ 2)) & \rightarrow 
\end{align*}
\]
Predicate: EQ

EQ compares two atoms for equivalence: "sharing the same representation"

\[
\begin{align*}
(EQ\ 1\ 1) & \rightarrow \ T \\
(EQ\ '(1\ 2)\ '(1\ 2)) & \rightarrow\ \text{nil} \\
(EQUAL\ '(1\ 2)\ '(1\ 2)) & \rightarrow\ T
\end{align*}
\]
Predicate: EQ

- EQ compares two atoms for equivalence:
  "sharing the same representation"

(EQ 1 1) → T
(EQ '(1 2) '(1 2)) → nil
(EQUAL '(1 2) '(1 2)) → T
(SETF a '(1 2))
(EQ a a) →
Predicate: EQ

- EQ compares two atoms for equivalence: "sharing the same representation"

- \( (\text{EQ } 1 \ 1) \rightarrow T \)
- \( (\text{EQ } '(1 \ 2) \ '(1 \ 2)) \rightarrow \text{nil} \)
- \( (\text{EQUAL } '(1 \ 2) \ '(1 \ 2)) \rightarrow T \)
- \( (\text{SETF} \ a \ '(1 \ 2)) \)
- \( (\text{EQ} \ a \ a) \rightarrow T \)
Predicate: =

- Compares two numbers

\[(= 1 1) \rightarrow \]
Predicate: =

- Compares two numbers

\[(= 1 1) \rightarrow T\]
Predicate: =

- Compares two numbers

\[(= 1 1) \rightarrow T\]
\[(= 1 2) \rightarrow\]
Predicate: =

- Compares two numbers

\[
(= 1 1) \rightarrow \text{T} \\
(= 1 2) \rightarrow \text{NIL}
\]
Predicate: =

- Compares two numbers

(- 1 1) → T
(- 1 2) → NIL
(- 'a 'b) →
Predicate: =

- Compares two numbers

$(= 1 1) \rightarrow \text{T}$
$(= 1 2) \rightarrow \text{NIL}$
$(= ’a ’b) \rightarrow "\text{Error!}"$
Predicates: NULL and NOT

- NULL tests if s-expr is empty list (or the NIL atom)
Predicates: NULL and NOT

- NULL tests if s-expr is empty list (or the NIL atom)

  \[(\text{NULL }()) \rightarrow \]
Predicates: NULL and NOT

- NULL tests if s-expr is empty list (or the NIL atom)

\[(\text{NULL }()) \rightarrow \text{T}\]
Predicates: NULL and NOT

- NULL tests if s-expr is empty list (or the NIL atom)

  - (NULL ()) → T
  - (NULL nil) →
Predicates: NULL and NOT

- NULL tests if s-exp is empty list (or the NIL atom)
  
  \[(\text{NULL }()) \rightarrow \text{T}\]
  \[(\text{NULL }\text{nil}) \rightarrow \text{T}\]
Predicates: NULL and NOT

- NULL tests if s-expr is empty list (or the NIL atom)

  (NULL ()) → T
  (NULL nil) → T
  (NULL 't ) →
Predicates: NULL and NOT

- NULL tests if s-expr is empty list (or the NIL atom)

  (NULL ()) → T
  (NULL nil) → T
  (NULL 't) → NIL
Predicates: NULL and NOT

- NULL tests if s-expr is empty list (or the NIL atom)

\[
\begin{align*}
(NULL () &\rightarrow T \\
(NULL \text{ nil}) &\rightarrow T \\
(NULL \ 't) &\rightarrow \text{NIL} \\
(NULL \ '(1)) &\rightarrow
\end{align*}
\]
Predicates: NULL and NOT

- NULL tests if s-expr is empty list (or the NIL atom)

\[
\begin{align*}
(NULL ()) & \rightarrow T \\
(NULL \text{ nil}) & \rightarrow T \\
(NULL \ 't) & \rightarrow \text{NIL} \\
(NULL \ '(1)) & \rightarrow \text{NIL}
\end{align*}
\]
Predicates: NULL and NOT

- NULL tests if s-expr is empty list (or the NIL atom)

  \[(NULL \,()) \to T\]
  \[(NULL\,\text{nil}) \to T\]
  \[(NULL\,’t) \to \text{NIL}\]
  \[(NULL\,’(1)) \to \text{NIL}\]
  \[(NULL\,1\,2) \to \text{Error - Null takes 1 argument}\]
Predicates: NULL and NOT

- **NULL** tests if s-expr is empty list (or the NIL atom)

  \[
  \begin{align*}
  &(\text{NULL }()) \rightarrow T \\
  &(\text{NULL }\text{nil}) \rightarrow T \\
  &(\text{NULL }'t) \rightarrow \text{NIL} \\
  &(\text{NULL }'(1)) \rightarrow \text{NIL} \\
  &(\text{NULL 1 2}) \rightarrow \text{Error - Null takes 1 argument}
  \end{align*}
  \]
Predicates: NULL and NOT

- **NULL** tests if s-expr is empty list (or the NIL atom)
  
  (NULL ()) → T  
  (NULL nil) → T  
  (NULL ’t) → NIL  
  (NULL ’(1)) → NIL  
  (NULL 1 2) → Error - Null takes 1 argument

- **NULL** works like a negation operator
Predicates: NULL and NOT

- **NULL** tests if s-expr is empty list (or the NIL atom)

  \[
  \begin{align*}
  \text{(NULL ())} & \rightarrow T \\
  \text{(NULL nil)} & \rightarrow T \\
  \text{(NULL 't)} & \rightarrow \text{NIL} \\
  \text{(NULL '(1))} & \rightarrow \text{NIL} \\
  \text{(NULL 1 2)} & \rightarrow \text{Error - Null takes 1 argument}
  \end{align*}
  \]

- **NULL** works like a negation operator

  \[
  \begin{align*}
  \text{(NULL (NULL '(1 2 3)))} & \rightarrow \\
  \end{align*}
  \]
Predicates: NULL and NOT

- NULL tests if s-expr is empty list (or the NIL atom)
  
  \[
  (\text{NULL }()) \rightarrow T \\
  (\text{NULL } \text{nil}) \rightarrow T \\
  (\text{NULL } 't ) \rightarrow \text{NIL} \\
  (\text{NULL } '(1)) \rightarrow \text{NIL} \\
  (\text{NULL } 1 \ 2) \rightarrow \text{Error - Null takes 1 argument}
  \]

- NULL works like a negation operator
  
  \[
  (\text{NULL } (\text{NULL } '(1 \ 2 \ 3))) \rightarrow T
  \]
Predicate: NOT

\[
\begin{align*}
\texttt{(NOT t)} & \rightarrow \texttt{NIL} \\
\texttt{(NOT nil)} & \rightarrow \texttt{T} \\
\texttt{(NOT (> 5 4))} & \rightarrow \texttt{T}
\end{align*}
\]
Predicate: \text{NOT}

\[(\text{NOT } \mathit{t}) \rightarrow \text{NIL}\]
Predicate: NOT

\[
\begin{align*}
&(\text{NOT } t) \rightarrow \text{NIL} \\
&(\text{NOT } \text{nil}) \rightarrow
\end{align*}
\]
Predicate: NOT

- (NOT t) → NIL
- (NOT nil) → T
Predicate: NOT

- `(NOT t) → NIL`
- `(NOT nil) → T`

Convention: save NULL for lists, use NOT for logical negation.
Predicate: NOT

- `(NOT t) → NIL`
- `(NOT nil) → T`
- Convention: save NULL for lists, use NOT for logical negation
  
  `(NOT (> 5 4)) →`
Predicate: NOT

- (NOT t) → NIL
- (NOT nil) → T

Convention: save NULL for lists, use NOT for logical negation

(NOT (> 5 4)) → T
Other Useful LISP Predicates

- Type predicates:
  - symbolp
  - consp
  - numberp
  - stringp
  - functionp

- Numerical comparison predicates: \( > \), \( < \), \( = \), \( \leq \), \( \geq \)
LISP Forms

- A form is a syntactic expression that can be evaluated by LISP
LISP Forms

▶ A form is a syntactic expression that can be evaluated by LISP

▶ In general: A FORM is one of

▶ a constant
  (e.g., t, nil, a number, ...)
▶ a variable
▶ a compound form
  (fn a₁ ... aₙ)
  where fn is a symbol, and
  each aᵢ is a form
LISP Forms

- Evaluation of a form results in an s-expression—the internal representation used by LISP
LISP Forms

- Evaluation of a form results in an s-expression— the internal representation used by LISP

- Up to now we have not distinguished the underlying s-expression from the printed syntax humans use to communicate them
LISP Forms

- Evaluation of a form results in an s-expression— the internal representation used by LISP

- Up to now we have not distinguished the underlying s-expression from the printed syntax humans use to communicate them

- Each type of form defines how the syntax should be interpreted
## Examples of Simple Forms

<table>
<thead>
<tr>
<th>Form</th>
<th>Evaluation</th>
<th>Internal S-expr</th>
</tr>
</thead>
<tbody>
<tr>
<td>variable</td>
<td>Lookup variable name in the variable symbol table</td>
<td>age ( \new ) 5</td>
</tr>
<tr>
<td>number</td>
<td>Convert ASCII digits to internal numeric representation</td>
<td>15 ( \new ) 00001111</td>
</tr>
<tr>
<td>constant</td>
<td>Lookup string and convert to interned value, or create a new constant</td>
<td>'fred ( \new ) constant125</td>
</tr>
</tbody>
</table>
Compound Forms: The Function

- One common compound form is the function call
- To evaluate a function call of the form: \((F_1 \ F_2 \ F_3 \ldots \ F_N)\)
Compound Forms: The Function

- One common compound form is the function call
- To evaluate a function call of the form: \((F_1 \ F_2 \ F_3 \ldots \ F_N)\)
  1. Lookup the first argument "F1" in the function symbol table and retrieve its implementation
Compound Forms: The Function

- One common compound form is the function call
- To evaluate a function call of the form: (F1 F2 F3 ... FN)
  1. Lookup the first argument "F1" in the function symbol table and retrieve its implementation
  2. Evaluate the remaining arguments
     (A recursive evaluation of each of the forms F2 ... FN)
Compound Forms: The Function

- One common compound form is the function call

- To evaluate a function call of the form: (F1 F2 F3 ... FN)
  1. Lookup the first argument "F1" in the function symbol table and retrieve its implementation
  2. Evaluate the remaining arguments
     (A recursive evaluation of each of the forms F2 ... FN)
  3. Apply the retrieved implementation to the arguments
Evaluation of Functions Example:

\[(\text{SETQ } a \ 5)\]
\[(\text{SETQ } b \ 6)\]
\[(\text{CONS } a \ b)\]
Evaluation of Functions Example:

\[
\begin{align*}
\text{(SETQ } a & \text{ 5)} \\
\text{(SETQ } b & \text{ 6)} \\
\text{(CONS } a & \text{ b)} \\
\text{ EVAL (CONS } a & \text{ b)}
\end{align*}
\]
Evaluation of Functions Example:

\[(\text{SETQ } a \ 5)\]
\[(\text{SETQ } b \ 6)\]
\[(\text{CONS } a \ b)\]

EVAL (CONS a b)

LOOKUP CONS \(\sim\) `<cons-implementation>`

{defined internally by LISP}
Evaluation of Functions Example:

\[(\text{SETQ } a \ 5)\]
\[(\text{SETQ } b \ 6)\]
\[(\text{CONS } a \ b)\]
\[
\begin{align*}
\text{EVAL } & (\text{CONS } a \ b) \\
\text{LOOKUP } & \text{CONS} \sim <\text{cons-implementation}> \\
& \{\text{defined internally by LISP}\} \\
\text{EVAL } & a
\end{align*}
\]
Evaluation of Functions Example:

(SETQ a 5)
(SETQ b 6)
(CONS a b)

EVAL (CONS a b)

LOOKUP CONS \(\mapsto\) <cons-implementation>
{defined internally by LISP}

EVAL a

LOOKUP a \(\mapsto\) 5
Evaluation of Functions Example:

(SETQ a 5)
(SETQ b 6)
(CONS a b)

EVAL (CONS a b)

LOOKUP CONS \[\rightarrow\] <cons-implementation>
{defined internally by LISP}

EVAL a

LOOKUP a \[\rightarrow\] 5

EVAL b
Evaluation of Functions Example:

\[
\begin{align*}
\text{(SETQ a 5)} \\
\text{(SETQ b 6)} \\
\text{(CONS a b)} \\
\text{EVAL (CONS a b)} \\
\text{LOOKUP CONS} & \leadsto \text{<cons-implementation>} \\
& \{\text{defined internally by LISP}\} \\
\text{EVAL a} \\
\text{LOOKUP a} & \leadsto 5 \\
\text{EVAL b} \\
\text{LOOKUP b} & \leadsto 6
\end{align*}
\]
Evaluation of Functions Example:

\[(\text{SETQ } a \ 5)\]
\[\text{(SETQ } b \ 6)\]
\[\text{(CONS } a \ b)\]
\[\text{EVAL (CONS } a \ b)\]
\[\text{LOOKUP CONS } \rightsquigarrow \text{ <cons-implementation>}\]
\[\{\text{defined internally by LISP}\}\]
\[\text{EVAL } a\]
\[\text{LOOKUP } a \ \rightsquigarrow \ 5\]
\[\text{EVAL } b\]
\[\text{LOOKUP } b \ \rightsquigarrow \ 6\]
\[\text{APPLY <cons-implementation> to 5 6}\]
Evaluation of Functions Example:

\[(\text{SETQ}\ a\ 5)\]
\[(\text{SETQ}\ b\ 6)\]
\[(\text{CONS}\ a\ b)\]

\[
\text{EVAL}\ (\text{CONS}\ a\ b)
\]

\[
\text{LOOKUP}\ \text{CONS}\ \mapsto\ <\text{cons-implementation}>
\]
\[
\{\text{defined internally by LISP}\}
\]

\[
\text{EVAL}\ a
\]

\[
\text{LOOKUP}\ a\ \mapsto\ 5
\]

\[
\text{EVAL}\ b
\]

\[
\text{LOOKUP}\ b\ \mapsto\ 6
\]

\[
\text{APPLY}\ <\text{cons-implementation}>\ \text{to}\ 5\ 6
\]

\[
\rightarrow\ (5\ .\ 6)
\]
Evaluation of Compound Forms: Special Forms

- LISP defines a variety of special forms that define their own special evaluation rules
Evaluation of Compound Forms: Special Forms

- LISP defines a variety of special forms that define their own special evaluation rules

- What would happen if we evaluated $(\text{SETQ } a (+ 1 1))$ like a function?
LISP defines a variety of special forms that define their own special evaluation rules.

What would happen if we evaluated (SETQ a (+ 1 1)) like a function?

A standard evaluation would:

- Lookup SETQ's implementation
- Then evaluate the arguments.
- When we evaluate 'a' we would get "ERROR! 'a' undefined!"
Evaluation of Compound Forms: Special Forms II

- Actual evaluation rule for (SETQ a (+ 1 1)) is a special form
  - Unlike functions, LISP must not evaluate the second argument "a" as it is not defined!
  - LISP must evaluate the third argument to know what value should be assigned to "a"
  - LISP then stores a value for 'a' in a local symbol table
Evaluation of Compound Forms: Special Forms II

- Actual evaluation rule for `(SETQ a (+ 1 1))` is a special form
  - Unlike functions, LISP must not evaluate the second argument "a" as it is not defined!
  - LISP must evaluate the third argument to know what value should be assigned to "a"
  - LISP then stores a value for 'a' in a local symbol table

- LISP allows you to define your own special forms with unique evaluation rules
Evaluation of OR Special Form

(SET Q a 5)
Evaluation of OR Special Form

(SET Q a 5)
(OR (> a (+ 2 1)) (> a 2) )
Evaluation of OR Special Form

\[(\text{SET} \ Q \ a \ 5)\]
\[(\text{OR} \ (> \ a \ (+ \ 2 \ 1)) \ (> \ a \ 2))\]

LOOKUP OR - a special form!!
Evaluation of OR Special Form

(SET Q a 5)
(OR (> a (+ 2 1)) (> a 2))
     LOOKUP OR - a special form!!
     EVAL (> a (+ 2 1))
Evaluation of OR Special Form

(Set Q a 5)
(OR (> a (+ 2 1)) (> a 2))

LOOKUP OR - a special form!!
EVAL (> a (+ 2 1))

LOOKUP >
Evaluation of OR Special Form

(SET Q a 5)
(OR (> a (+ 2 1)) (> a 2))
  LOOKUP OR - a special form!!
  EVAL (> a (+ 2 1))
    LOOKUP >
    EVAL a →5
Evaluation of OR Special Form

\[(\text{SET Q a 5})\]
\[(\text{OR (> a (+ 2 1)) (> a 2)) )\]

LOOKUP OR - a special form!!
EVAL (> a (+ 2 1))
LOOKUP >
EVAL a \rightarrow 5
EVAL (+ 2 1)
Evaluation of OR Special Form

(SET Q a 5)
(OR (> a (+ 2 1)) (> a 2))

LOOKUP OR - a special form!!
EVAL (> a (+ 2 1))

LOOKUP >
EVAL a \rightarrow 5
EVAL (+ 2 1)

LOOKUP +
Evaluation of OR Special Form

(SET Q a 5)
(OR (> a (+ 2 1)) (> a 2))
LOOKUP OR - a special form!!
EVAL (> a (+ 2 1))
LOOKUP >
EVAL a → 5
EVAL (+ 2 1)
LOOKUP +
EVAL 2 → 2
Evaluation of OR Special Form

(SET Q a 5)

(OR (> a (+ 2 1)) (> a 2) )

LOOKUP OR - a special form!!

EVAL (> a (+ 2 1))

LOOKUP >

EVAL a → 5

EVAL (+ 2 1)

LOOKUP +

EVAL 2 → 2

EVAL 1 → 1
Evaluation of OR Special Form

(SET Q a 5)
(OR (> a (+ 2 1)) (> a 2) )

LOOKUP OR - a special form!!
EVAL (> a (+ 2 1))

LOOKUP >
EVAL a → 5
EVAL (+ 2 1)

LOOKUP +
EVAL 2 → 2
EVAL 1 → 1

→ 3
Evaluation of OR Special Form

(SET Q a 5)
(OR (> a (+ 2 1)) (> a 2) )
LOOKUP OR - a special form!!
EVAL (> a (+ 2 1))
LOOKUP >
EVAL a → 5
EVAL (+ 2 1)
LOOKUP +
EVAL 2 → 2
EVAL 1 → 1
→ 3
→ T
Evaluation of OR Special Form

(SET Q a 5)
(OR (> a (+ 2 1)) (> a 2) )

LOOKUP OR - a special form!!
EVAL (> a (+ 2 1))
  LOOKUP >
    EVAL a \rightarrow 5
    EVAL (+ 2 1)
      LOOKUP +
        EVAL 2 \rightarrow 2
        EVAL 1 \rightarrow 1
        \rightarrow 3
        \rightarrow T
Evaluation of OR Special Form

(SE T Q a 5)

(OR (> a (+ 2 1)) (> a 2))

LOOKUP OR - a special form!!

EVAL (> a (+ 2 1))

LOOKUP >

EVAL a → 5

EVAL (+ 2 1)

LOOKUP +

EVAL 2 → 2

EVAL 1 → 1

→ 3

→ T

→ T ; stops on first "true" argument
Defining Your Own Functions

- Until now, we have only used built-in *FUNCTIONS*. (Eg: CAR, NULL, +, ...)

The term LAMBDA comes from a mathematical theory of computation developed by Alonzo Church in the 1930's!
Defining Your Own Functions

- Until now, we have only used built-in *FUNCTIONS*. (Eg: CAR, NULL, +, ...)

- New functions are defined using the LAMBDA (λ) special form:

  (lambda-keyword parameter-list function-body-form)
Defining Your Own Functions

- Until now, we have only used built-in FUNCTIONS. (E.g.: CAR, NULL, +, ...)

- New functions are defined using the LAMBDA ($\lambda$) special form:

  (lambda-keyword parameter-list function-body-form)

- The term LAMBDA comes from a mathematical theory of computation developed by Alonzo Church in the 1930’s!
Defining Your Own Functions

▶ Until now, we have only used built-in *FUNCTIONS*. (Eg: CAR, NULL, +, ...)

▶ New functions are defined using the LAMBDA (\(\lambda\)) special form:

\[(\text{lambda-keyword parameter-list function-body-form})\]

▶ The term LAMBDA comes from a mathematical theory of computation developed by Alonzo Church in the 1930’s!

▶ We’ll skip the theory for now and plunge into using LAMBDA’s to create our own functions.
Simple Example of a LAMBDA Expression

Here is an example defining a two-argument mathematical function

\[
\text{( LAMBDA (X Y) (+ X (* 2 Y)))}
\]

keyword parameters  body form
Comparison of Procedures with λ-Expressions

- `(LAMBDA (X Y) (PLUS X (TIMES 2 Y)))`
  is a FUNCTION
Comparison of Procedures with $\lambda$-Expressions

- (LAMBDA (X Y) (PLUS X (TIMES 2 Y))) is a FUNCTION

- Mathematically, it is a mapping:

  $\mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$
Comparison of Procedures with $\lambda$-Expressions

- $(\text{LAMBDA (X Y) (PLUS X (TIMES 2 Y)))}$ is a FUNCTION
- Mathematically, it is a mapping:
  \[ \mathbb{R} \times \mathbb{R} \to \mathbb{R} \]
- What would a traditional procedure with the same intent look like?
Comparison of Procedures with $\lambda$-Expressions

- $(\text{LAMBDA} \ (X \ Y) \ (\text{PLUS} \ X \ (\text{TIMES} \ 2 \ Y)))$ is a FUNCTION.

- Mathematically, it is a mapping:
  \[ \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R} \]

- What would a traditional procedure with the same intent look like?
  ```c
  double foo( double x, double y) {
    return x + 2 * y;
  }
  ```
Comparison of Procedures with \( \lambda \)-Expressions

- \( \text{(LAMBDA (X Y) (PLUS X (TIMES 2 Y)))} \) is a FUNCTION

- Mathematically, it is a mapping:

  \[ \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R} \]

- What would a traditional procedure with the same intent look like?

  ```c
  double foo( double x, double y) {
    return x + 2 * y;
  }
  ```

- Notice that the \( \lambda \)-expression does not define a function name or the types of its parameters
Using λ-Expressions

- Evaluation of a LAMBDA returns an anonymous procedure
Using λ-Expressions

- Evaluation of a LAMBDA returns an anonymous procedure
- It can be created where it is needed and need not be stored in a symbol table

Evaluation of (LAMBDA (X Y) (+ X (* 2 Y)) ) with arguments 4 and 15:

```
(LAMBDA (X Y) (+ X (* 2 Y)) ) 4 15
```
Using $\lambda$-Expressions

- Evaluation of a LAMBDA returns an anonymous procedure
- It can be created where it is needed and need not be stored in a symbol table
- $\lambda$-expression can be used anywhere you would use a function symbol.
Using $\lambda$-Expressions

- Evaluation of a LAMBDA returns an anonymous procedure
- It can be created where it is needed and need not be stored in a symbol table
- $\lambda$-expression can be used anywhere you would use a function symbol.
- How would I apply (LAMBDA (X Y) (+ X (* 2 Y))) to the arguments 4 and 15?
Using $\lambda$-Expressions

- Evaluation of a LAMBDA returns an anonymous procedure
- It can be created where it is needed and need not be stored in a symbol table
- $\lambda$-expression can be used anywhere you would use a function symbol.

- How would I apply (LAMBDA (X Y) (+ X (* 2 Y)) ) to the arguments 4 and 15?

\[( (\text{LAMBDA} (X \ Y) (+ \ X \ (* \ 2 \ Y)) ) \ \ 4 \ 15 ) \]
Evaluating Forms with $\lambda$-Expressions

Evaluating:

\[
\left( \text{LAMBDA} \ (X \ Y) \ (+ \ X \ (* \ 2 \ Y)) \right) \ 4 \ 15
\]

Create a new environment
Within the env, assign $X \leftarrow 4$, $Y \leftarrow 15$
Within the env, evaluate: $(+ X (* 2 Y))$

- $EVAL[X]$;
  - $4$
- $EVAL[(* 2 Y)]$;
  - $*\ of\ EVAL[2]and\ EVAL[Y]$
  - $EVAL[2]$;
    - $2$
  - $EVAL[Y]$;
    - $15$
  - $30$
- $34$
Evaluating Forms with λ-Expressions

Evaluating:

\((\text{LAMBDA} (X \ Y) (+ X (* 2 Y)))\) 4 15\

Create a new environment
Evaluating Forms with λ-Expressions

Evaluating:

( (LAMBDA (X Y) (+ X (* 2 Y)) ) 4 15 )

Create a new environment
Within the env, assign X ← 4, Y← 15
Evaluating Forms with λ-Expressions

Evaluating:

( (LAMBDA (X Y) (+ X (* 2 Y)) ) 4 15 )

Create a new environment
Within the env, assign X ← 4, Y ← 15
Within the env, evaluate: (+ X (* 2 Y)): 
Evaluating Forms with λ-Expressions

Evaluating:

\[(\text{LAMBDA} (X \ Y) (+ X (* 2 \ Y))) \ 4 \ 15\]

Create a new environment

Within the env, assign \[X \leftarrow 4, \ Y \leftarrow 15\]

Within the env, evaluate: \[(+ X (* 2 \ Y))\]:

\[\text{EVAL}[(+ X (* 2 \ Y))]\]
Evaluating Forms with $\lambda$-Expressions

Evaluating:

\[
\left( \text{LAMBDA} \left( X \ Y \right) \left( + \ X \ \left( * \ 2 \ Y \right) \right) \right) \ 4 \ 15
\]

Create a new environment
Within the env, assign $X \leftarrow 4$, $Y \leftarrow 15$
Within the env, evaluate: $(+ \ X \ (* \ 2 \ Y))$

\[
\text{EVAL}[(+ \ X \ (* \ 2 \ Y))]
\]

$\mapsto$ $+$ of \text{EVAL}[X]$ and \text{EVAL}[(* \ 2 \ Y)]$
Evaluating Forms with $\lambda$-Expressions

Evaluating:

$$( \text{LAMBDA} (X \ Y) \ (+ \ X \ (* \ 2 \ Y)) ) \ 4 \ 15 )$$

Create a new environment

Within the env, assign $X \leftarrow 4$, $Y \leftarrow 15$

Within the env, evaluate: $+ \ X \ (* \ 2 \ Y))$:

EVAL[$(+ \ X \ (* \ 2 \ Y))]$

$\mapsto +$ of EVAL[$X$] and EVAL[$(* \ 2 \ Y)]$

EVAL[$X$] $\mapsto 4$
Evaluating Forms with λ-Expressions

Evaluating:

\((\text{LAMBDA} (X \ Y) (+ X (* 2 Y)))\) \(4\) \(15\)

Create a new environment

Within the env, assign \(X \leftarrow 4\), \(Y \leftarrow 15\)

Within the env, evaluate: \((+ X (* 2 Y))\):

\(\text{EVAL}[(+ X (* 2 Y))]\)

\(\sim\) + of \(\text{EVAL}[X]\) and \(\text{EVAL}[(* 2 Y)]\)

\(\text{EVAL}[X] \sim 4\)

\(\text{EVAL}[(* 2 Y)]\)
Evaluating Forms with λ-Expressions

Evaluating:

\[
(\text{LAMBDA} (X \ Y) (\text{+} \ X \ (\text{*} \ 2 \ Y)) \ 4 \ 15)
\]

Create a new environment

Within the env, assign \( X \leftarrow 4 \), \( Y \leftarrow 15 \)

Within the env, evaluate: \((+ \ X \ (\text{*} \ 2 \ Y))\):

\[
\text{EVAL}[(+ \ X \ (\text{*} \ 2 \ Y))] \\
\leadsto + \ of \ \text{EVAL}[X] \ and \ \text{EVAL}[(\text{*} \ 2 \ Y)] \\
\text{EVAL}[X] \leadsto 4 \\
\text{EVAL}[(\text{*} \ 2 \ Y)] \\
\leadsto * \ of \ \text{EVAL}[2] \ and \ \text{EVAL}[Y]
\]
Evaluating Forms with $\lambda$-Expressions

Evaluating:

\[
\left( \left( \text{LAMBDA} (X \ Y) \ (+ \ X \ (* \ 2 \ Y)) \right) \ 4 \ 15 \right)
\]

Create a new environment

Within the env, assign $X \leftarrow 4$, $Y \leftarrow 15$

Within the env, evaluate: $\left( + \ X \ (* \ 2 \ Y) \right)$:

$\text{EVAL}[\left( + \ X \ (* \ 2 \ Y) \right)]$

$\leadsto + \text{ of } \text{EVAL}[X] \text{ and } \text{EVAL}[\left( * \ 2 \ Y \right)]$

$\text{EVAL}[X] \leadsto 4$

$\text{EVAL}[\left( * \ 2 \ Y \right)]$

$\leadsto * \text{ of } \text{EVAL}[2] \text{ and } \text{EVAL}[Y]$

$\text{EVAL}[2] \leadsto 2$
Evaluating Forms with $\lambda$-Expressions

Evaluating:

$$\left( \left( \text{LAMBDA} \ (X \ Y) \ (+ \ X \ (* \ 2 \ Y)) \right) \ 4 \ 15 \right)$$

Create a new environment
Within the env, assign $X \leftarrow 4$, $Y \leftarrow 15$

Within the env, evaluate: $(+ \ X \ (* \ 2 \ Y))$:

- $\text{EVAL}[(+ \ X \ (* \ 2 \ Y))]$
- $\mapsto + \text{ of } \text{EVAL}[X] \text{ and } \text{EVAL}[(* \ 2 \ Y)]$
- $\text{EVAL}[X] \mapsto 4$
- $\text{EVAL}[(* \ 2 \ Y)]$
- $\mapsto * \text{ of } \text{EVAL}[2] \text{ and } \text{EVAL}[Y]$
- $\text{EVAL}[2] \mapsto 2$
- $\text{EVAL}[Y] \mapsto 15$
Evaluating Forms with $\lambda$-Expressions

Evaluating:

\[(\text{LAMBDA} (X \ Y) (+ X (* 2 Y)) \ 4 \ 15)\]

Create a new environment

Within the env, assign $X \leftarrow 4$, $Y \leftarrow 15$

Within the env, evaluate: $(+ X (* 2 Y))$:

\[
\text{EVAL}[(+ X (* 2 Y))] \\
\leadsto + \text{ of EVAL}[X] \text{ and EVAL}[(* 2 Y)] \\
\text{EVAL}[X] \leadsto 4 \\
\text{EVAL}[(* 2 Y)] \\
\leadsto * \text{ of EVAL}[2] \text{ and EVAL}[Y] \\
\text{EVAL}[2] \leadsto 2 \\
\text{EVAL}[Y] \leadsto 15 \\
\leadsto 30
\]
Evaluating Forms with $\lambda$-Expressions

Evaluating:

$$(\ (LAMBDA \ (X \ Y) \ (+ \ X \ (* \ 2 \ Y)) \ ) \ 4 \ 15)$$

Create a new environment

Within the env, assign $X \leftarrow 4$, $Y \leftarrow 15$

Within the env, evaluate: $(+ X (* 2 Y))$:

$EVAL[(+ X (* 2 Y))]$

$\mapsto +$ of $EVAL[X]$ and $EVAL[(* 2 Y)]$

$EVAL[X] \mapsto 4$

$EVAL[(* 2 Y)]$

$\mapsto *$ of $EVAL[2]$ and $EVAL[Y]$

$EVAL[2] \mapsto 2$

$EVAL[Y] \mapsto 15$

$\mapsto 30$

$\mapsto 34$
Summary of λ-expressions

- A LAMBDA Expression is: a list with 3 elements:

  (LAMBDA (a1 a2 ... an) <form>) where ai are atoms
Summary of $\lambda$-expressions

- A LAMBDA Expression is: a list with 3 elements:

  $$(\text{LAMBDA} \ (a_1 \ a_2 \ldots \ a_n) \ <\text{form}> \) \text{ where } a_i \text{ are atoms}$$

- A $\lambda$-expression is used in the following form:

  $$((\text{LAMBDA} \ (a_1 \ a_2 \ldots \ a_n) \ <\text{form}>)) \ v_1 \ v_2 \ldots \ v_n$$

- Each $v_i$ is a value or argument.
Summary of $\lambda$-expressions

- A LAMBDA Expression is: a list with 3 elements:

  \[(\text{LAMBDA} (a_1 \ a_2 \ldots \ a_n) \ <\text{form}>\) \text{ where } a_i \text{ are atoms}\]

- A $\lambda$-expression is used in the following form:

  \[(\ (\text{LAMBDA} (a_1 \ a_2 \ldots \ a_n) \ <\text{form}>\) \ v_1 \ v_2 \ldots \ v_n)\]

- Each $v_i$ is a value or argument.

- Evaluation:
  1. Create a new environment
  2. Bind each argument $a_i$ to its corresponding value $v_i$
  3. Evaluate $<\text{form}>$ in the environment.
Another Example of \( \lambda \) Evaluation

Evaluate: \((\text{LAMBDA} (X \ Y) (\text{CONS} (\text{CAR} \ X) \ Y)) \ '(A \ B) \ '(C)\):

Create a new environment
Another Example of λ Evaluation

Evaluate: ( (LAMBDA (X Y) (CONS (CAR X) Y)) ’(A B) ’(C) ):

Create a new environment
Assign X←(A B) and Y←(C)
Another Example of λ Evaluation

Evaluate: \((\text{LAMBDA} (X \ Y) (\text{CONS} (\text{CAR} X) Y)) \ ' (A \ B) \ ' (C) )\):

Create a new environment
Assign \(X \leftarrow (A \ B)\) and \(Y \leftarrow (C)\)
Evaluate: \((\text{CONS} (\text{CAR} X) Y)\):
Another Example of λ Evaluation

Evaluate: \((\text{LAMBDA} (X \ Y) (\text{CONS} (\text{CAR} X) Y)) \ '(A \ B) \ '(C)\):

Create a new environment
Assign \(X \leftarrow (A \ B)\) and \(Y \leftarrow (C)\)
Evaluate: \((\text{CONS} (\text{CAR} X) Y)\):
Lookup CONS
Another Example of $\lambda$ Evaluation

Evaluate: \((\text{LAMBDA} (X \ Y) (\text{CONS} (\text{CAR} \ X) \ Y)) \ '(A \ B) \ '(C)\):

Create a new environment
Assign \(X \leftarrow (A \ B)\) and \(Y \leftarrow (C)\)
Evaluate: \((\text{CONS} (\text{CAR} \ X) \ Y)\):
Lookup CONS
EVAL\([(\text{CAR} \ X)]\)
Another Example of \( \lambda \) Evaluation

Evaluate: \((\text{LAMBDA} (X \ Y) (\text{CONS} (\text{CAR} \ X) \ Y)) \ '(A \ B) \ '(C)\):

Create a new environment
Assign \(X \leftarrow (A \ B)\) and \(Y \leftarrow (C)\)
Evaluate: \((\text{CONS} (\text{CAR} \ X) \ Y)\):
Lookup CONS
EVAL\([\text{(CAR} \ X)\]\
EVAL \(X\leftarrow (A \ B)\)
Another Example of λ Evaluation

Evaluate: \((\text{LAMBDA} (X \ Y) (\text{CONS} (\text{CAR} \ X) \ Y)) \ '(A \ B) \ '(C)\):

Create a new environment
Assign \(X \leftarrow (A \ B)\) and \(Y \leftarrow (C)\)
Evaluate: \((\text{CONS} (\text{CAR} \ X) \ Y)\):
Lookup CONS
EVAL\([(\text{CAR} \ X)]\)
EVAL \(X \leftarrow (A \ B)\)
\(\mapsto A\)
Another Example of λ Evaluation

Evaluate: ( (LAMBDA (X Y) (CONS (CAR X) Y)) ’(A B) ’(C) ):

Create a new environment
Assign X←(A B) and Y←(C)
Evaluate: (CONS (CAR X) Y):
Lookup CONS
EVAL[(CAR X)]
  EVAL X←(A B)
  ~ A
EVAL[Y] ~ (C)
Another Example of λ Evaluation

Evaluate: ( (LAMBDA (X Y) (CONS (CAR X) Y)) ’(A B) ’(C) ):

Create a new environment
Assign X ← (A B) and Y ← (C)
Evaluate: (CONS (CAR X) Y):
  Lookup CONS
  EVAL[(CAR X)]
    EVAL X ~⇒ (A B)
    ~⇒ A
    EVAL[Y] ~⇒ (C)
    ~⇒ (A C)
Functions vs. Forms

Functions are abstract mathematical objects with some implementation.
Functions vs. Forms

Functions are abstract mathematical objects with some implementation.

- Applying the function CAR to (A B C) results in A.
Functions vs. Forms

Functions are abstract mathematical objects with some implementation.

- Applying the function CAR to (A B C) results in A

- There are two types of functions:
  - Primitives: (CONS, ATOM, ...)
  - User-defined: (( LAMBDA (v1 v2 ... vn) <form>) e1 e2 ...en)
Functions vs. Forms

Functions are abstract mathematical objects with some implementation.

- Applying the function CAR to (A B C) results in A

- There are two types of functions:
  - Primitives: (CONS, ATOM, ...)
  - User-defined: ((LAMBDA (v1 v2 ... vn) <form>) e1 e2 ...en)

Forms are syntactic expressions which get evaluated in a context.
Functions vs. Forms

Functions are abstract mathematical objects with some implementation.

- Applying the function CAR to \((A \ B \ C)\) results in \(A\)

- There are two types of functions:
  - Primitives: \((\text{CONS}, \text{ATOM}, \ldots)\)
  - User-defined: \(\left(\lambda (v_1 \ v_2 \ldots \ v_n) \ <\text{form}>\right) \ e_1 \ e_2 \ldots \ e_n\)

Forms are syntactic expressions which get evaluated in a context

- "\((+ \ X \ 5)\)" is a form that applies the function \(+\) to \(X\) and \(5\)
Functions vs. Forms

Functions are abstract mathematical objects with some implementation.

- Applying the function CAR to (A B C) results in A
- There are two types of functions:
  - Primitives: (CONS, ATOM, ...)
  - User-defined: (( LAMBDA (v1 v2 ... vn) <form>) e1 e2 ...en)

Forms are syntactic expressions which get evaluated in a context

- "(+ X 5)" is a form that applies the function + to X and 5
- "(CAR X)" is a form that applies the function CAR to X
Functions vs. Forms

Functions are abstract mathematical objects with some implementation.

- Applying the function `CAR` to `(A B C)` results in `A`

- There are two types of functions:
  - Primitives: `(CONS, ATOM, ...)`
  - User-defined: `(( LAMBDA (v1 v2 ... vn) <form>) e1 e2 ...en)`

Forms are syntactic expressions which get evaluated in a context

- "`(+ X 5)`" is a form that applies the function `+` to `X` and `5`
- "`(CAR X)`" is a form that applies the function `CAR` to `X`
- "`( (LAMBDA (X) X) 7)`" is a form which applies `λ` to `7`
More Examples of \( \lambda \)-Forms

\[
( (\text{LAMBDA} (X Y) (\text{CONS} X (\text{CDR} (\text{CDR} Y))) \ 'A \ (B \ C \ D)) )
\]
\[\rightarrow\]
More Examples of $\lambda$-Forms

\[
( \text{LAMBDA} (X \ Y) (\text{CONS} X (\text{CDR} (\text{CDR} Y))) \ 'A' 'B' 'C' 'D') \rightarrow (A \ D)
\]
More Examples of λ-Forms

\[
\begin{align*}
(\text{LAMBDA} \ (X \ Y) \ (\text{CONS} \ X \ (\text{CDR} \ (\text{CDR} \ Y))) \ &\ 'A' \ '(B \ C \ D) \\
\rightarrow &\ (A \ D)
\end{align*}
\]

\[
\begin{align*}
(\text{LAMBDA} \ (X \ Y) \ (\text{CONS} \ X \ '(\text{CDR} \ Y)) \ &\ 'A' \ '(B \ C \ D) \\
\rightarrow &\ \text{undefined} --- 'LAMBDA' \ is \ not \ function
\end{align*}
\]
More Examples of λ-Forms

\[
\begin{align*}
( \text{LAMBDA} (X \ Y) (\text{CONS} \ X \ (\text{CDR} \ (\text{CDR} \ Y))) \ 'A \ '(B \ C \ D) ) & \rightarrow (A \ D) \\
( \text{LAMBDA} (X \ Y) (\text{CONS} \ X \ '(\text{CDR} \ Y)) \ 'A \ '(B \ C \ D) ) & \rightarrow (A \ \text{CDR} \ Y)
\end{align*}
\]
More Examples of λ-Forms

\[
(\text{LAMBDA } (X \ Y) (\text{CONS } X \ (\text{CDR } (\text{CDR } Y))) \ 'A \ '(B \ C \ D) )
\]
→(A D)

\[
(\text{LAMBDA } (X \ Y) (\text{CONS } X \ '(\text{CDR } Y)) \ 'A \ '(B \ C \ D) )
\]
→(A CDR Y)

\[
(\text{LAMBDA } (X) (\text{CAR } X)) (\text{CONS } 'A \ \text{NIL})
\]
→
More Examples of λ-Forms

( (LAMBDA (X Y) (CONS X (CDR (CDR Y))) 'A '(B C D) )
→(A D)
( (LAMBDA (X Y) (CONS X '(CDR Y)) 'A '(B C D) )
→(A CDR Y)
( (LAMBDA (X) (CAR X)) (CONS 'A NIL) )
→A
More Examples of λ-Forms

\[
\begin{align*}
&\quad (\text{LAMBDA} (X \ Y) (\text{CONS} \ X (\text{CDR} (\text{CDR} \ Y))) \ 'A \ '(B \ C \ D) ) \\
&\quad \rightarrow (A \ D) \\
&\quad (\text{LAMBDA} (X \ Y) (\text{CONS} \ X '(\text{CDR} \ Y)) \ 'A \ '(B \ C \ D) ) \\
&\quad \rightarrow (A \ \text{CDR} \ Y) \\
&\quad (\text{LAMBDA} (X) (\text{CAR} \ X)) (\text{CONS} \ 'A \ \text{NIL}) ) \\
&\quad \rightarrow A \\
&\quad (\text{LAMBDA} (X) (\text{CAR} \ X)) '(\text{CONS} \ 'A \ \text{NIL}) ) \\
&\quad \rightarrow
\end{align*}
\]
More Examples of \( \lambda \)-Forms

\[
\begin{align*}
( (\text{LAMBDA} (X \ Y) (\text{CONS} \ X \ (\text{CDR} \ (\text{CDR} \ Y))) \ 'A \ 'B \ C \ D) ) & \rightarrow (A \ D) \\
( (\text{LAMBDA} (X \ Y) (\text{CONS} \ X \ '((\text{CDR} \ Y))) \ 'A \ 'B \ C \ D) ) & \rightarrow (A \ \text{CDR} \ Y) \\
( (\text{LAMBDA} (X) (\text{CAR} \ X)) \ (\text{CONS} \ 'A \ \text{NIL}) ) & \rightarrow (\text{CONS} 'A \ \text{NIL}) & \\
( (\text{LAMBDA} (X) (\text{CAR} \ X)) '((\text{CONS} 'A \ \text{NIL}) ) & \rightarrow \text{CONS}
\end{align*}
\]
More Examples of \( \lambda \)-Forms

\[
\begin{align*}
( \text{LAMBDA} (X \ Y) \ (\text{CONS} \ X \ (\text{CDR} \ (\text{CDR} \ Y))) & \quad 'A \ ' (B \ C \ D) \\
\rightarrow & (A \ D) \\
( \text{LAMBDA} (X \ Y) \ (\text{CONS} \ X \ '((\text{CDR} \ Y))) & \quad 'A \ ' (B \ C \ D) \\
\rightarrow & (A \ \text{CDR} \ Y) \\
( \text{LAMBDA} (X) \ (\text{CAR} \ X)) & \quad (\text{CONS} \ 'A \ \text{NIL}) \\
\rightarrow & \ A \\
( \text{LAMBDA} (X) \ (\text{CAR} \ X)) & \quad '((\text{CONS} \ 'A \ \text{NIL}) \\
\rightarrow & \ \text{CONS} \\
( \text{LAMBDA} (X) \ (\text{CAR} \ X)) & \quad '((\text{LAMBDA} (X) \ (\text{CAR} \ X)) \\
\rightarrow & \\
\end{align*}
\]
More Examples of $\lambda$-Forms

\[
\begin{align*}
(\text{LAMBDA} (X \ Y) (\text{CONS} \ X \ (\text{CDR} \ (\text{CDR} \ Y))) \ 'A \ 'B \ 'C \ 'D) \\
\rightarrow (A \ D)
\end{align*}
\]

\[
\begin{align*}
(\text{LAMBDA} (X \ Y) (\text{CONS} \ X \ '(\text{CDR} \ Y)) \ 'A \ 'B \ 'C \ 'D) \\
\rightarrow (A \ \text{CDR} \ Y)
\end{align*}
\]

\[
\begin{align*}
(\text{LAMBDA} (X) (\text{CAR} \ X)) \ (\text{CONS} \ 'A \ \text{NIL}) \\
\rightarrow \text{A}
\end{align*}
\]

\[
\begin{align*}
(\text{LAMBDA} (X) (\text{CAR} \ X)) \ '(\text{CONS} \ 'A \ \text{NIL}) \\
\rightarrow \text{CONS}
\end{align*}
\]

\[
\begin{align*}
(\text{LAMBDA} (X) (\text{CAR} \ X)) \ '(\text{LAMBDA} (X) (\text{CAR} \ X)) \\
\rightarrow \text{LAMBDA}
\end{align*}
\]
More Examples of $\lambda$-Forms

\[
\begin{align*}
&\left( (\text{LAMBDA} (X \ Y) (\text{CONS} X \ (\text{CDR} \ (\text{CDR} \ Y)))) \ 'A' \ '(B \ C \ D) \right) \\
&\rightarrow (A \ D) \\
&\left( (\text{LAMBDA} (X \ Y) (\text{CONS} X \ '(\text{CDR} \ Y))) \ 'A' \ '(B \ C \ D) \right) \\
&\rightarrow (A \ \text{CDR} \ Y) \\
&\left( (\text{LAMBDA} (X) (\text{CAR} X)) \ (\text{CONS} \ 'A' \ \text{NIL}) \right) \\
&\rightarrow A \\
&\left( (\text{LAMBDA} (X) (\text{CAR} X)) \ (\text{CONS} \ 'A' \ \text{NIL}) \right) \\
&\rightarrow \text{CONS} \\
&\left( (\text{LAMBDA} (X) (\text{CAR} X)) \ (\text{LAMBDA} (X) (\text{CAR} X)) \right) \\
&\rightarrow \text{LAMBDA} \\
&\left( (\text{LAMBDA} (X) (\text{CAR} X)) \ (\text{LAMBDA} (X) (\text{CAR} X)) \right) \\
&\rightarrow 
\end{align*}
\]
More Examples of λ-Forms

\[
\begin{align*}
(\text{(LAMBDA (X Y) (CONS X (CDR (CDR Y))) 'A '(B C D)}) & \rightarrow (A D) \\
(\text{(LAMBDA (X Y) (CONS X '(CDR Y)) 'A '(B C D)}) & \rightarrow (A \ CDR \ Y) \\
(\text{(LAMBDA (X) (CAR X)) (CONS 'A NIL)}) & \rightarrow A \\
(\text{(LAMBDA (X) (CAR X)) '(CONS 'A NIL)}) & \rightarrow \text{CONS} \\
(\text{(LAMBDA (X) (CAR X)) '(LAMBDA (X) (CAR X))}) & \rightarrow \text{LAMBDA} \\
(\text{(LAMBDA (X) (CAR X)) (LAMBDA (X) (CAR X))}) & \rightarrow \text{undefined --- "LAMBDA" is not function}
\end{align*}
\]
The LET Operator

- The LET operator creates a new environment with local bindings in it.
The LET Operator

- The LET operator creates a new environment with local bindings in it

\[
\text{(SETQ X 3)} \quad X \rightarrow \quad 3
\]

\[
\text{(LET ((X 5) (Y 2)) (* 2 X))} \quad \rightarrow \quad 10
\]

\[
\text{(* 2 X)} \quad \rightarrow \quad 6
\]

- Operations within the LET use the local values
- Variables outside of the LET are unaffected
The LET Operator

- The LET operator creates a new environment with local bindings in it.

(SETQ X 3)
X →3
The LET Operator

* The LET operator creates a new environment with local bindings in it

\[(\text{SETQ } X \ 3)\]
\[X \to 3\]

\[(\text{LET } ((X \ 5)
   \quad (Y \ 2))
   \quad (* \ 2 \ X))\]
\[\to\]
The LET Operator

- The LET operator creates a new environment with local bindings in it

\[
(\text{SETQ } X \ 3) \\
X \to 3
\]

\[
(\text{LET ((X 5)} \\
\quad (Y \ 2)) \\
\quad (* \ 2 \ X)) \\
\to 10
\]
The LET Operator

The LET operator creates a new environment with local bindings in it

\[(\text{SETQ} \ X \ 3)\]
\[X \rightarrow 3\]

\[(\text{LET} \ ((X \ 5) \n(Y \ 2)) \n(* \ 2 \ X))\]
\[\rightarrow 10\]

\[(* \ 2 \ X)\]
The LET Operator

- The LET operator creates a new environment with local bindings in it.

\[
\text{(SETQ X 3)}
\]
\[
X \rightarrow 3
\]

\[
\text{(LET ((X 5)}
\]
\[
\text{(Y 2))}
\]
\[
\text{(* 2 X))}
\]
\[
\rightarrow 10
\]

\[
\text{(* 2 X) \rightarrow 6}
\]
The LET Operator

- The LET operator creates a new environment with local bindings in it

\[
\text{(SETQ } X 3) \\
X \rightarrow 3
\]

\[
\text{(LET ((X 5) } \\
\quad (Y 2))
\quad (* 2 X)) \\
\rightarrow 10
\]

\[
(* 2 X) \rightarrow 6
\]

- Operations within the LET use the local values
The **LET Operator**

- The LET operator creates a new environment with local bindings in it

    (SETQ X 3)
    X → 3

    (LET ((X 5)
           (Y 2))
       (* 2 X))
    → 10

    (* 2 X) → 6

- Operations within the LET use the local values

- Variables outside of the LET are unaffected
The Relationship Between LET & λ

- The general form of the LET form is:

\[(\text{LET} \ (\ (v_1 \ e_1) \ldots (v_n \ e_n) \ ) \ \langle \text{form} \rangle)\]
The Relationship Between LET & λ

▶ The general form of the LET form is:

\[(\text{LET} ( (v_1 \ e_1) \ldots (v_n \ e_n) ) \langle \text{form} \rangle )\]

▶ It is equivalent to the following λ - form:

\[((\text{LAMBDA} (v_1 \ldots v_n) \langle \text{form} \rangle ) \ e_1 \ldots e_n)\]
Examples of LET and λ

(LET ((X 'A)
      (Y '(B C)))
  (CONS X Y))
Examples of LET and λ

(LET ((X ’A)
     (Y ’(B C)))
     (CONS X Y))
→(A B C)
Examples of LET and λ

\[
(\text{LET } ((X \ 'A) \\
   (Y \ '(B C))) \\
   (\text{CONS } X \ Y)) \\
\rightarrow (A \ B \ C) \\
(\text{LET } (X (\text{CONS } (\text{QUOTE} \ A) \ \text{NIL}) \ ) \\
   (\text{LET } (Y \ 2) \\
    (\text{CONS } X \ Y) \ ) \\
\rightarrow (A . 2) \\
(\text{LET } (X (\text{CONS } (\text{QUOTE} \ A) \ \text{NIL}) \ ) \\
   (\text{CAR} \ X) \ ) \\
\rightarrow A
\]
Examples of LET and λ

(LET ((X 'A)
      (Y '(B C)))
      (CONS X Y))
→(A B C)

( (LAMBDA (X Y) (CONS X Y)) 'A '(B C))
→(A B C)
Examples of LET and $\lambda$

\[
(\text{LET} ((X \ 'A) \\
          (Y \ '(B \ C))) \\
      (\text{CONS} \ X \ Y))
\rightarrow (A \ B \ C)
\]

\[
(\text{(LAMBDA} \ (X \ Y) \ (\text{CONS} \ X \ Y)) \ 'A \ '(B \ C))
\rightarrow (A \ B \ C)
\]

\[
(\text{LET} ((X \ 'A) \\
           (Y \ '(B \ C))) \\
      (\text{LET} ((Y \ 2)) \\
        (\text{CONS} \ X \ Y)))
\]
Examples of LET and \(\lambda\)

\[
(\text{LET} ((X 'A) \\
     (Y '(B C))) \\
     (\text{CONS} X Y)) \\
\rightarrow (A B C) \\
(\text{LET} ((X 'A) \\
     (Y '(B C))) \\
     (\text{LET} ((Y 2)) \\
     (\text{CONS} X Y)) \\
\rightarrow (A . 2)
\]
Examples of LET and \( \lambda \)

\[
\begin{align*}
(\text{LET} & \ ((X \ 'A) \\
& (Y \ '(B \ C)) ) \\
& (\text{CONS} \ X \ Y)) \\
\rightarrow & (A \ B \ C) \\
((\text{LAMBDA} (X \ Y) (\text{CONS} \ X \ Y)) \ 'A \ '(B \ C)) \\
\rightarrow & (A \ B \ C) \\
(\text{LET} & \ ((X \ 'A) \\
& (Y \ '(B \ C)))) \\
& (\text{LET} \ ((Y \ 2 )) \\
& (\text{CONS} \ X \ Y)) \\
\rightarrow & (A \ . \ 2) \\
(\text{LET} & \ ((X \ (\text{CONS} \ (\text{QUOTE} \ A) \ \text{NIL}))) ) \\
& (\text{CAR} \ X) )
\end{align*}
\]
Examples of LET and λ

\(\text{(LET ((X 'A)}\)
\(\text{ (Y '(B C))} )\)
\(\text{(CONS X Y)} ))\)
\(\rightarrow (A \ B \ C)\)
\(\text{( (LAMBDA (X Y) (CONS X Y)) 'A '(B C))}\)
\(\rightarrow (A \ B \ C)\)
\(\text{ (LET ( (X 'A)}\)
\(\text{ (Y '(B C ))})\)
\(\text{(LET (( Y 2 ))}\)
\(\text{(CONS X Y)) }))\)
\(\rightarrow (A . \ 2)\)
\(\text{(LET ( (X (CONS (QUOTE A) NIL) ) )}\)
\(\text{(CAR X) })\)
\(\rightarrow A\)
The Quote Special Form

- Special forms exercise control over the evaluation of their arguments
The Quote Special Form

- Special forms exercise control over the evaluation of their arguments
- We have seen the "QUOTE" form
The Quote Special Form

- Special forms exercise control over the evaluation of their arguments
- We have seen the "QUOTE" form
- Abbreviated to ’x, but it can be written (QUOTE X)
The Quote Special Form

- Special forms exercise control over the evaluation of their arguments
- We have seen the "QUOTE" form
- Abbreviated to 'x, but it can be written (QUOTE X)
- Unlike function-call forms, QUOTE performs no evaluation of arguments
The Quote Special Form

- Special forms exercise control over the evaluation of their arguments
- We have seen the "QUOTE" form
- Abbreviated to ‘x, but it can be written (QUOTE X)
- Unlike function-call forms, QUOTE performs no evaluation of arguments
- QUOTE is a "form" as it defines a way of interpreting a syntactic expression
Examples of the QUOTE Form

(SETQ B 5)

B
Examples of the QUOTE Form

(SETQ B 5)
B → 5
Examples of the QUOTE Form

(SETQ B 5)
B \rightarrow 5

(QUOTE B)
Examples of the QUOTE Form

\[(SETQ B 5)\]
\[B \rightarrow 5\]

\[(QUOTE B) \rightarrow B\]
Examples of the QUOTE Form

\[(\text{SETQ } B \ 5)\]
\[B \rightarrow 5\]

\[(\text{QUOTE } B) \rightarrow B\]

C
Examples of the QUOTE Form

\[(\text{SETQ} \ B \ 5)\]
\[B \rightarrow 5\]

\[(\text{QUOTE} \ B) \rightarrow B\]

\[C \rightarrow \text{"Error Undefined"}\]
Examples of the QUOTE Form

(SETQ B 5)
B → 5

(QUOTE B) → B

C → "Error Undefined"

(QUOTE C)
Examples of the QUOTE Form

(SETQ B 5)
B → 5

(QUOTE B) → B

C → "Error Undefined"

(QUOTE C) → C
Examples of the QUOTE Form

(SETQ B 5)
B → 5

(QUOTE B) → B

C → "Error Undefined"

(QUOTE C) → C

(+ 1 2)
Examples of the QUOTE Form

(SETQ B 5)
B → 5

(QUOTE B) → B

C → "Error Undefined"

(QUOTE C) → C

(+ 1 2) → 3
Examples of the QUOTE Form

(SETQ B 5)
B → 5

(QUOTE B) → B

C → "Error Undefined"

(QUOTE C) → C

(+ 1 2) → 3

(QUOTE (+ 1 2))
Examples of the QUOTE Form

(SETQ B 5)
B → 5

(QUOTE B) → B

C → "Error Undefined"

(QUOTE C) → C

(+ 1 2) → 3

(QUOTE (+ 1 2)) → (+ 1 2)
Examples of the QUOTE Form

(SETQ B 5)
B → 5

(QUOTE B) → B

C → "Error Undefined"

(QUOTE C) → C

(+ 1 2) → 3

(QUOTE (+ 1 2)) → (+ 1 2)

(QUOTE (QUOTE A))
Examples of the `QUOTE` Form

\[(\text{SETQ B 5})\]
\[\text{B} \rightarrow 5\]

\[(\text{QUOTE B}) \rightarrow \text{B}\]

\[\text{C} \rightarrow "\text{Error Undefined}"\]

\[(\text{QUOTE C}) \rightarrow \text{C}\]

\[(+ 1 2) \rightarrow 3\]

\[(\text{QUOTE (+ 1 2)}) \rightarrow (+ 1 2)\]

\[(\text{QUOTE (QUOTE A)}) \rightarrow 'A\]

Dr. B. Price & Dr. R. Greiner

CMPUT 325 - Lisp Basics
LIST Form

- The use of QUOTE and LIST are sometimes confused.
LIST Form

- The use of QUOTE and LIST are sometimes confused.

- The LIST function
  
  1. evaluates each of its arguments
  2. concatenates results into a list
LIST Form

- The use of QUOTE and LIST are sometimes confused

- The LIST function
  1. evaluates each of its arguments
  2. concatenates results into a list

- A simple example:
LIST Form

- The use of QUOTE and LIST are sometimes confused

- The LIST function
  1. *evaluates* each of its arguments
  2. concatenates results into a list

- A simple example:

  (LIST (+ 1 2) (+ 3 4) (cons 'a 'b))
LIST Form

- The use of QUOTE and LIST are sometimes confused

- The LIST function
  1. evaluates each of its arguments
  2. concatenates results into a list

- A simple example:

  (LIST (+ 1 2) (+ 3 4) (cons 'a 'b))
  → (3 7 (a . b))
LIST Form

- The use of QUOTE and LIST are sometimes confused

- The LIST function
  1. evaluates each of its arguments
  2. concatenates results into a list

- A simple example:

\[(\text{LIST } (+ 1 2) (+ 3 4) (\text{cons } \text{'a } \text{'b}))\]
\[\rightarrow (3 7 (\text{a . b}))\]

- Could use QUOTE to prevent evaluation of LIST’s arguments:
LIST Form

▶ The use of QUOTE and LIST are sometimes confused

▶ The LIST function

1. *evaluates* each of its arguments
2. concatenates results into a list

▶ A simple example:

\[
\text{(LIST} \ (\ + \ 1 \ 2) \ (+ \ 3 \ 4) \ (\text{cons} \ 'a \ 'b))
\rightarrow \ (3 \ 7 \ (a \ . \ b))
\]

▶ Could use QUOTE to prevent evaluation of LIST’s arguments:

\[
\text{(LIST} \ '(+ \ 1 \ 2) \ '+\ 3 \ 4) \ '(\text{cons} \ 'a \ 'b))
\]
LIST Form

- The use of QUOTE and LIST are sometimes confused

- The LIST function
  1. evaluates each of its arguments
  2. concatenates results into a list

- A simple example:

  (LIST (+ 1 2) (+ 3 4) (cons 'a 'b))
  → (3 7 (a . b))

- Could use QUOTE to prevent evaluation of LIST’s arguments:

  (LIST ’(+ 1 2) ’(+ 3 4) ’(cons ’a ’b))
  → ((+ 1 2) (+ 3 4) (cons ’a ’b))
Examples of LIST and QUOTE

(LIST t 5) →
Examples of LIST and QUOTE

\[(\text{LIST } t \ 5) \rightarrow (t \ 5)\]
Examples of LIST and QUOTE

\[(\text{LIST } t \ 5) \rightarrow (t \ 5)\]

\[(\text{LIST } (\text{QUOTE } \text{BAC})) \rightarrow\]
Examples of LIST and QUOTE

\[(\text{LIST } t \ 5) \rightarrow (t \ 5)\]

\[(\text{LIST } (\text{QUOTE} \ BAC)) \rightarrow (BAC)\]
Examples of LIST and QUOTE

\[(\text{LIST } t \, 5) \rightarrow (t \, 5)\]

\[(\text{LIST } (\text{QUOTE } \text{BAC})) \rightarrow (\text{BAC})\]

\[(\text{LIST } \text{'PLUS} \, 3 \, 4) \rightarrow \]
Examples of LIST and QUOTE

(LIST t 5) → (t 5)

(LIST (QUOTE BAC)) → (BAC)

(LIST 'PLUS 3 4) → (PLUS 3 4)
Examples of LIST and QUOTE

\[
\text{(LIST } t \ 5) \rightarrow (t\ 5)
\]

\[
\text{(LIST (QUOTE BAC)) } \rightarrow \text{(BAC)}
\]

\[
\text{(LIST 'PLUS 3 4) } \rightarrow \text{(PLUS 3 4)}
\]

\[
\text{(LIST '(PLUS 3 4)) } \rightarrow
\]
Examples of LIST and QUOTE

\[ (\text{LIST } t \ 5) \rightarrow (t \ 5) \]

\[ (\text{LIST } (\text{QUOTE BAC})) \rightarrow (\text{BAC}) \]

\[ (\text{LIST } '\text{PLUS} \ 3 \ 4) \rightarrow (\text{PLUS} \ 3 \ 4) \]

\[ (\text{LIST } '(\text{PLUS} \ 3 \ 4)) \rightarrow ((\text{PLUS} \ 3 \ 4)) \]
Examples of LIST and QUOTE

\[(\text{LIST } t \ 5) \rightarrow (t \ 5)\]

\[(\text{LIST (QUOTE BAC)}) \rightarrow (\text{BAC})\]

\[(\text{LIST 'PLUS 3 4}) \rightarrow (\text{PLUS 3 4})\]

\[(\text{LIST '(PLUS 3 4)}) \rightarrow ((\text{PLUS 3 4}))\]

\[(\text{LIST (PLUS 3 4)}) \rightarrow \]
Examples of LIST and QUOTE

\[
\text{(LIST t 5)} \rightarrow (t 5)
\]

\[
\text{(LIST (QUOTE BAC))} \rightarrow (\text{BAC})
\]

\[
\text{(LIST 'PLUS 3 4)} \rightarrow (\text{PLUS 3 4})
\]

\[
\text{(LIST '(PLUS 3 4))} \rightarrow ((\text{PLUS 3 4}))
\]

\[
\text{(LIST (PLUS 3 4))} \rightarrow (7)
\]
Examples of LIST and QUOTE

\[(\text{LIST } t \ 5) \rightarrow (t \ 5)\]

\[(\text{LIST } (\text{QUOTE } \text{BAC})) \rightarrow \text{BAC}\]

\[(\text{LIST } '(\text{PLUS } 3 \ 4)) \rightarrow (\text{PLUS } 3 \ 4)\]

\[(\text{LIST } '(\text{PLUS } 3 \ 4)) \rightarrow (\text{PLUS } 3 \ 4)\]

\[(\text{LIST } (\text{PLUS } 3 \ 4)) \rightarrow (7)\]

\[(\text{CAR } (\text{LIST } 'A \ 'B)) \rightarrow \]
Examples of LIST and QUOTE

(LIST t 5) → (t 5)

(LIST (QUOTE BAC)) → (BAC)

(LIST 'PLUS 3 4) → (PLUS 3 4)

(LIST '(PLUS 3 4)) → ((PLUS 3 4))

(LIST (PLUS 3 4)) → (7)

(CAR (LIST 'A 'B)) → A
Examples of LIST and QUOTE

\[
\begin{align*}
\text{(LIST t 5)} & \rightarrow (t 5) \\
\text{(LIST (QUOTE BAC))} & \rightarrow (\text{BAC}) \\
\text{(LIST ’PLUS 3 4)} & \rightarrow (\text{PLUS 3 4}) \\
\text{(LIST ’(PLUS 3 4))} & \rightarrow ((\text{PLUS 3 4})) \\
\text{(LIST (PLUS 3 4))} & \rightarrow (7) \\
\text{(CAR (LIST ’A ’B))} & \rightarrow A \\
\text{(CDR (LIST ’A ’B))} & \rightarrow
\end{align*}
\]
Examples of LIST and QUOTE

(LIST t 5) → (t 5)

(LIST (QUOTE BAC)) → (BAC)

(LIST ’PLUS 3 4) → (PLUS 3 4)

(LIST ’(PLUS 3 4)) → ((PLUS 3 4))

(LIST (PLUS 3 4)) → (7)

(CAR (LIST ’A ’B)) → A

(CDR (LIST ’A ’B)) → (B)
Using \textsc{List} to Create \(\lambda\)-Forms

- These expressions create \(\lambda\)-forms:

\begin{align*}
\text{(LIST 'LAMBDA '(x) (LIST 'CAR 'x))} & \rightarrow \text{(LAMBDA (x) (CAR x))} \\
\text{((LAMBDA (fn) (LIST 'LAMBDA '(x) (LIST fn 'x))) 'CAR)} & \rightarrow \text{(LAMBDA (x) (CAR x))} \\
\text{((LAMBDA (fn) (LIST 'LAMBDA '(x) (LIST fn 'x))) 'CDR)} & \rightarrow \text{(LAMBDA (x) (CDR x))}
\end{align*}
Using LIST to Create λ- Forms

These expressions create λ-forms:

\[(\text{LIST } \text{'LAMBDA } \text{'(x) (LIST } \text{'CAR } \text{'x))}\]
Using LIST to Create λ- Forms

These expressions create λ-forms:

\[
\text{(LIST 'LAMBDA '(x) (LIST 'CAR 'x))} \\
\rightarrow (\text{LAMBDA (x) (CAR x))}
\]
Using \textbf{LIST} to Create λ- Forms

- These expressions create λ-forms:

\[
\text{(LIST 'LAMBDA '(x) (LIST 'CAR 'x))} \\
\rightarrow \text{(LAMBDA (x) (CAR x))}
\]

\[
\text{((LAMBDA (fn) (LIST 'LAMBDA '(x) (LIST fn 'x))) 'CAR)}
\]
Using \textit{LIST} to Create $\lambda$- Forms

$\triangleright$ These expressions create $\lambda$-forms:

$$(\text{LIST } '\text{LAMBDA } '(x) (\text{LIST } '\text{CAR } 'x))$$
$\rightarrow (\text{LAMBDA } (x) (\text{CAR } x))$

$$((\text{LAMBDA } (\text{fn}) (\text{LIST } '\text{LAMBDA } '(x) (\text{LIST } \text{fn } 'x))) '\text{CAR})$$
$\rightarrow (\text{LAMBDA } (x) (\text{CAR } x))$$
Using \textsc{List} to Create $\lambda$-\textsc{Forms}

- These expressions create $\lambda$-forms:

\[
\text{LIST } \text{LAMBDA } '(x) \text{ (LIST } \text{CAR } 'x)\text{)} \\
\rightarrow \text{LAMBDA } (x) \text{ (CAR } x)\text{)} \\
\]

\[
\text{((LAMBDA } (fn) \text{ (LIST } \text{LAMBDA } '(x) \text{ (LIST fn } 'x)\text{)}) } \text{ 'CAR} \text{)} \\
\rightarrow \text{LAMBDA } (x) \text{ (CAR } x)\text{)} \\
\]

\[
\text{((LAMBDA } (fn) \text{ (LIST } \text{LAMBDA } '(x) \text{ (LIST fn } 'x)\text{)}) } \text{ 'CDR} \text{)} \\
\]
Using LSDT to Create $\lambda$-Forms

- These expressions create $\lambda$-forms:

\[
\text{LIST 'LAMBDA '(x) (LIST 'CAR 'x))} \\
\rightarrow \text{(LAMBDA (x) (CAR x))}
\]

\[
((\text{LAMBDA (fn) (LIST 'LAMBDA '(x) (LIST fn 'x))) 'CAR}) \\
\rightarrow \text{(LAMBDA (x) (CAR x))}
\]

\[
((\text{LAMBDA (fn) (LIST 'LAMBDA '(x) (LIST fn 'x))) 'CDR}) \\
\rightarrow \text{(LAMBDA (x) (CDR x))}
\]
Using LIST to Create \( \lambda \)-Forms

▶ These expressions create \( \lambda \)-forms:

\[
\text{\begin{align*}
(\text{LIST 'LAMBDA } '(x) (\text{LIST 'CAR } 'x)) \\
\Rightarrow (\text{LAMBDA (x) (CAR x)}) \\
((\text{LAMBDA (fn) (LIST 'LAMBDA } '(x) (\text{LIST fn } 'x)))) '\text{CAR} \\
\Rightarrow (\text{LAMBDA (x) (CAR x)}) \\
((\text{LAMBDA (fn) (LIST 'LAMBDA } '(x) (\text{LIST fn } 'x)))) '\text{CDR} \\
\Rightarrow (\text{LAMBDA (x) (CDR x)})
\end{align*}}
\]

▶ This is just a three element list. We’ll explain how to use it below.
Examples of COND

▶ Can write “If $x > 0$ then $x$ else $-x$.” as:

\[
\text{(COND ((> X 0) x )
( t (- x)) )}
\]
Examples of \texttt{COND}

- Can write “If $x > 0$ then $x$ else $-x$.” as:

\[
\text{(COND ( (> X 0) X ) ( t ( - X )))}
\]

- Note the use of the constant \texttt{t} here to represent the always true condition
Examples of COND

- Can write “If $x > 0$ then $x$ else $-x$.” as:

  ```lisp
  (COND ((> X 0) x)
          (t (- x)))
  ```

- Note the use of the constant $t$ here to represent the always true condition

- Suppose we wish to write a "lookup" function:

  ```lisp
  (lambda (name)
            (COND ((EQ name 'bob) 'id321)
                   ((EQ name 'russ) 'id452)
                   ((EQ name 'lisa) 'id621)
                   (t 'unknown)))
  ```
Examples of **COND**

- Can write “If $x > 0$ then $x$ else $-x$.” as:
  
  ```lisp
  (COND ((> X 0) x )
        ( t ( - x)))
  ```

- Note the use of the constant `t` here to represent the always true condition

- Suppose we wish to write a "lookup" function:
  
  ```lisp
  (lambda (name)
    (COND ( (EQ name 'bob ) 'id321)
           ( (EQ name 'russ) 'id452)
           ( (EQ name 'lisa) 'id621)
           ( t 'unknown))
  ```

- Again, the constant `t` represents a default action
Conditional Forms: COND

- The general form of the COND expression is:

  \[
  \text{(COND} \quad (\text{CONDITION-1} \quad \text{FORM-1}) \\
  (\text{CONDITION-2} \quad \text{FORM-2}) \\
  : \\
  (\text{CONDITION-N} \quad \text{FORM-N}) \quad )
  \]
**Conditional Forms: COND**

- The general form of the COND expression is:

  ```lisp
  (COND (CONDITION-1 FORM-1)
        (CONDITION-2 FORM-2)
        ...
        (CONDITION-N FORM-N))
  ```

- Check conditions sequentially until one succeeds
Conditional Forms: COND

- The general form of the COND expression is:

  \[
  \text{(COND (CONDITION-1 FORM-1) (CONDITION-2 FORM-2) \ldots (CONDITION-N FORM-N))}
  \]

- Check conditions sequentially until one succeeds

- Evaluate corresponding form and return
Conditional Forms: COND

- The general form of the COND expression is:

  \[
  (\text{COND} \quad (\text{CONDITION}-1 \ \text{FORM}-1) \\
  (\text{CONDITION}-2 \ \text{FORM}-2) \\
  \vdots \\
  (\text{CONDITION}-N \ \text{FORM}-N) )
  \]

- Check conditions sequentially until one succeeds
- Evaluate corresponding form and return
- If no condition succeeds, COND results in nil
Conditional Forms: COND

- The general form of the COND expression is:

  \[
  \text{(COND (CONDITION-1 FORM-1)} \\
  \text{(CONDITION-2 FORM-2)} \\
  \vdots \\
  \text{(CONDITION-N FORM-N) )}
  \]

- Check conditions sequentially until one succeeds
- Evaluate corresponding form and return
- If no condition succeeds, COND results in nil
- Is COND a function?
Conditional Forms: **COND**

- The general form of the COND expression is:

  \[
  \text{(COND (CONDITION-1 FORM-1) (CONDITION-2 FORM-2) \ldots (CONDITION-N FORM-N))}
  \]

- Check conditions sequentially until one succeeds
- Evaluate corresponding form and return
- If no condition succeeds, COND results in nil
- Is COND a function? No - only partially evaluated
COND vs. the Procedural IF Statement

» The "COND" expression:

```
(COND  (CONDITION-1 FORM-1)
       (CONDITION-2 FORM-2)
       ...
       (CONDITION-N FORM-N)
)
```
COND vs. the Procedural IF Statement

➤ The "COND" expression:

```
(COND  (CONDITION-1 FORM-1)
       (CONDITION-2 FORM-2)
       ...
       (CONDITION-N FORM-N))
```

➤ The equivalent procedural "IF"

```
IF CONDITION-1 THEN
  FORM-1
ELSEIF CONDITION-2
  FORM-2
ELSEIF CONDITION-N
  FORM-N
END
```
COND vs C "?" Macro

- LISP COND function is closer C’s condition-value macro, "?"
COND vs C "??" Macro

- LISP COND function is closer C’s condition-value macro, "??"
- The COND is conditional valued function, and not a control structure
**COND vs C "?" Macro**

- **LISP COND function** is closer C’s condition-value macro, "?"

- The **COND** is conditional valued function, and not a control structure

- In **C**, one can write: \( y = (x > 0) ? x : -x \)
COND vs C "?" Macro

- **LISP COND function is closer C’s condition-value macro, "?"**
- The COND is conditional valued function, and not a control structure
- In C, one can write: \( y = (x > 0) \ ? \ x : -x \)
- This is, of course, the ABS function we defined earlier:

\[
\text{(COND ((> X 0) x )}
\text{ ( t (- x)) )}
\]
COND vs C "?" Macro

- LISP COND function is closer C’s condition-value macro, "?"

- The COND is conditional valued function, and not a control structure

- In C, one can write: \( y = (x > 0) \ ? \ x : \ -x \)

- This is, of course, the ABS function we defined earlier:

  \[
  \text{(COND ((> X 0) x )} \\
  \text{ ( t \ (- x)))}
  \]

- Can’t ask the value of a procedural "IF" statement
More examples of COND I

> Only code for satisfied conditions is executed

(LAMBDA (x y)
    (COND ( (= y 0) ’error)
           ( t ( \ x y)))))
More examples of COND I

- Only code for satisfied conditions is executed

(LAMBDA (x y)
  (COND ( (= y 0) 'error)
          ( t (\ x y)))))

(apply \ '(10 2)) →
More examples of COND

- Only code for satisfied conditions is executed

\[
\text{(LAMBDA (x y)}
\text{  (COND ( (= y 0) 'error))}
\text{  ( t (\ x y)))}
\]

\[(\text{apply } \lambda ' (10 2)) \rightarrow 2\]
More examples of COND I

▶ Only code for satisfied conditions is executed

(LAMBDA (x y)
   (COND ( (= y 0) ’error)
       ( t ( \ x y))))

(apply λ ’(10 2)) → 2
(apply λ ’(10 0)) →
More examples of \texttt{COND} I

- Only code for satisfied conditions is executed

\begin{verbatim}
(LAMBDA (x y)
    (COND ((= y 0) 'error)
           (t (\ x y)))

(apply \'(10 2)) \rightarrow 2
(apply \'(10 0)) \rightarrow 'error
\end{verbatim}
More examples of COND II

(COND (t 5)) →
More examples of COND II

(COND (t 5)) → 5
More examples of COND II

(COND (t 5)) → 5
(COND (nil 5)) →
More examples of COND II

(COND (t 5)) → 5
(COND (nil 5)) → nil
More examples of COND II

(COND (t 5)) → 5
(COND (nil 5)) → nil
(COND (t 5)
  (t 6)) →
More examples of COND II

(COND (t 5)) → 5
(COND (nil 5)) → nil
(COND (t 5)
    (t 6)) → 5
More examples of COND II

(COND (t 5)) → 5
(COND (nil 5)) → nil
(COND (t 5)
  (t 6)) → 5
(COND (nil 5)
  (t 6)) →
More examples of COND II

(COND (t 5)) → 5
(COND (nil 5)) → nil
(COND (t 5)
    (t 6)) → 5
(COND (nil 5)
    (t 6)) → 6
More examples of COND II

\[
\begin{align*}
\text{(COND (t 5))} & \rightarrow 5 \\
\text{(COND (nil 5))} & \rightarrow \text{nil} \\
\text{(COND (t 5)} & \quad \text{(t 6))} \rightarrow 5 \\
\text{(COND (nil 5)} & \quad \text{(t 6))} \rightarrow 6 \\
\text{(COND (nil 3)} & \quad \text{(t 1))} \rightarrow
\end{align*}
\]
More examples of `COND` II

\[
\begin{align*}
\text{(COND (t 5)) } & \rightarrow 5 \\
\text{(COND (nil 5)) } & \rightarrow \text{ nil} \\
\text{(COND (t 5) } & \rightarrow 5 \\
\text{ \hphantom{(COND (t 5) } } & \rightarrow 5 \\
\text{(COND (nil 5) } & \rightarrow 6 \\
\text{ \hphantom{(COND (nil 5) } } & \rightarrow 6 \\
\text{(COND (nil 3) } & \rightarrow 1 \\
\text{ \hphantom{(COND (nil 3) } } & \rightarrow 1
\end{align*}
\]
More examples of \texttt{COND} II

\begin{align*}
\text{(COND (t 5))} & \rightarrow 5 \\
\text{(COND (nil 5))} & \rightarrow \text{nil} \\
\text{(COND (t 5)} \\
\text{\hspace{1em} (t 6))} & \rightarrow 5 \\
\text{(COND (nil 5)} \\
\text{\hspace{1em} (t 6))} & \rightarrow 6 \\
\text{(COND (nil 3)} \\
\text{\hspace{1em} (t 1))} & \rightarrow 1 \\
\text{(COND nil)} & \rightarrow
\end{align*}
More examples of COND II

\[
\begin{align*}
\text{(COND (t 5)) } & \rightarrow 5 \\
\text{(COND (nil 5)) } & \rightarrow \text{ nil} \\
\text{(COND (t 5)} \\
\text{\quad (t 6)) } & \rightarrow 5 \\
\text{(COND (nil 5)} \\
\text{\quad (t 6)) } & \rightarrow 6 \\
\text{(COND (nil 3)} \\
\text{\quad (t 1)) } & \rightarrow 1 \\
\text{(COND nil) } & \rightarrow \text{ Error 'nil' should be a list}
\end{align*}
\]
Boolean Special Forms

- Boolean operators: `and`, `and`, `or` are implemented as special forms
Boolean Special Forms

- Boolean operators: and and or are implemented as special forms

- Value is value of last argument evaluated
Boolean Special Forms

- Boolean operators: `and` and `or` are implemented as special forms
- Value is value of last argument evaluated
- Both `and` and `or` implement short-circuiting (evaluate only enough arguments to determine truth value)
Boolean Special Forms

- Boolean operators: `and` and `or` are implemented as special forms
- Value is value of last argument evaluated
- Both `and` and `or` implement short-circuiting (evaluate only enough arguments to determine truth value)
- The `and` special form

\[
\text{(and (< 6 4) (>= (/ 10 0) 0))} \rightarrow \text{T}
\]

\[
\text{(and (not (= y 0)) (x y))} \rightarrow \text{X/Y if } y \neq 0 \text{ otherwise nil}
\]

\[
\text{(and (> 6 4) (>= 4 2))} \rightarrow \text{T}
\]

\[
\text{(and 5 6)} \rightarrow \text{6}
\]

\[
\text{(and t (cdr '(1))} \rightarrow \text{NIL}
\]
Boolean Special Forms

- Boolean operators: `and` and `or` are implemented as special forms
- Value is value of last argument evaluated
- Both `and` and `or` implement short-circuiting
  (evaluate only enough arguments to determine truth value)
- The `and` special form

```
(and (< 6 4) (>= (/ 10 0) 0)) →
```

Dr. B. Price & Dr. R. Greiner
CMPUT 325 - Lisp Basics 54
### Boolean Special Forms

- **Boolean operators:** `and` and `or` are implemented as special forms.
- **Value is value of last argument evaluated**
- **Both `and` and `or` implement short-circuiting**
  (evaluate only enough arguments to determine truth value)
- **The `and` special form**

\[
\text{(and (< 6 4) (>= (/ 10 0) 0)) } \rightarrow \ T
\]
Boolean Special Forms

- Boolean operators: and and or are implemented as special forms
- Value is value of last argument evaluated
- Both and and or implement short-circuiting (evaluate only enough arguments to determine truth value)
- The and special form

\[
\text{(and (< 6 4) (>= (/ 10 0) 0)) } \rightarrow \text{T} \\
\text{(and (not (= y 0)) (\ x y)) } \rightarrow 
\]
Boolean Special Forms

- Boolean operators: `and` and `or` are implemented as special forms
- Value is value of last argument evaluated
- Both `and` and `or` implement short-circuiting (evaluate only enough arguments to determine truth value)
- The `and` special form

\[
(\text{and} \ (< \ 6 \ 4) \ (\geq \ (/ \ 10 \ 0) \ 0)) \rightarrow \ T \\
(\text{and} \ (\text{not} \ (= \ y \ 0)) \ (\backslash \ x \ y)) \rightarrow \\
\text{X/Y if } y \neq 0 \text{ otherwise } \text{nil}
\]
Boolean Special Forms

- Boolean operators: `and`, `and`, and `or` are implemented as special forms.
- Value is value of last argument evaluated.
- Both `and` and `or` implement short-circuiting (evaluate only enough arguments to determine truth value).
- The `and` special form:

  
  \[
  \begin{align*}
  (\text{and} (\lt 6 4) (\geq (\div 10 0) 0)) & \rightarrow T \\
  (\text{and} (\text{not} (= y 0)) (\text{\textbackslash} x y)) & \rightarrow \text{X/Y if } y \neq 0 \text{ otherwise nil} \\
  (\text{and} (> 6 4) (\geq 4 2)) & \rightarrow
  \end{align*}
  \]
Boolean Special Forms

- Boolean operators: `and` and `or` are implemented as special forms.
- Value is value of last argument evaluated.
- Both `and` and `or` implement short-circuiting (evaluate only enough arguments to determine truth value).
- The `and` special form:

\[
\begin{align*}
\text{(and (< 6 4) (>= (/ 10 0) 0))} & \rightarrow T \\
\text{(and (not (= y 0)) (\ x y))} & \rightarrow X/Y \text{ if } y \neq 0 \text{ otherwise nil} \\
\text{(and (> 6 4) (>= 4 2))} & \rightarrow T
\end{align*}
\]
Boolean Special Forms

- Boolean operators: `and` and `or` are implemented as special forms
- Value is value of last argument evaluated
- Both `and` and `or` implement short-circuiting (evaluate only enough arguments to determine truth value)

- The `and` special form

  `(and (< 6 4) (>= (/ 10 0) 0)) → T`
  `(and (not (= y 0)) (\ x y)) →
   X/Y if y≠0 otherwise nil`
  `(and (> 6 4) (>= 4 2)) → T`
  `(and 5 6) →`
Boolean Special Forms

- Boolean operators: `and`, `and`, `or` are implemented as special forms.
- Value is value of last argument evaluated.
- Both `and`, `and`, `or` implement short-circuiting
  (evaluate only enough arguments to determine truth value).
- The `and` special form

  \[
  \begin{align*}
  (\text{and} \ (< 6 \ 4) \ (\geq (\div 10 \ 0) \ 0)) & \rightarrow \ T \\
  (\text{and} \ (\text{not} \ (= \ y \ 0)) \ (\backslash \ x \ y)) & \rightarrow \\
  & \text{X/Y if } y \neq 0 \text{ otherwise nil} \\
  (\text{and} \ (> 6 \ 4) \ (\geq 4 \ 2)) & \rightarrow \ T \\
  (\text{and} \ 5 \ 6) & \rightarrow \ 6
  \end{align*}
  \]
Boolean Special Forms

- Boolean operators: `and`, `and`, and `or` are implemented as special forms.
- Value is value of last argument evaluated.
- Both `and`, `and`, and `or` implement short-circuiting (evaluate only enough arguments to determine truth value).

The `and` special form:

- \[(\text{and} \ (< \ 6 \ 4) \ (>\ = \ (/ \ 10 \ 0) \ 0)) \rightarrow \ T\]
- \[(\text{and} \ (>\ = \ 4 \ 2)) \rightarrow \ T\]
- \[(\text{and} \ 5 \ 6) \rightarrow \ 6\]
- \[(\text{and} \ t \ (\text{cdr} \ '(1))) \rightarrow \]\n
\[\text{X/Y if } y \neq 0 \text{ otherwise nil}\]
Boolean Special Forms

- Boolean operators: and and or are implemented as special forms
- Value is value of last argument evaluated
- Both and and or implement short-circuiting (evaluate only enough arguments to determine truth value)

The and special form

- \((\text{and} (< 6 4) (>= (/ 10 0) 0)) \rightarrow T\)
- \((\text{and} (\text{not} (= y 0)) (\text{\textbackslash} x y)) \rightarrow X/Y \text{ if } y \neq 0 \text{ otherwise nil}\)
- \((\text{and} (> 6 4) (>= 4 2)) \rightarrow T\)
- \((\text{and} 5 6) \rightarrow 6\)
- \((\text{and} t (\text{cdr } '(1))) \rightarrow \text{NIL}\)
OR Special Forms

- The `or` special form

```lisp
(or (= 3 9) (eq 'price 'price))
→ T
(or 5 6 7)
→ 5
```
OR Special Forms

- The or special form

\[(\text{or } (= 3 9) (\text{eq } \text{'price } \text{'price})) \rightarrow\]
OR Special Forms

- The or special form

\( (\text{or} \ (\text{=} \ 3 \ 9) \ (\text{eq} \ \text{'price} \ \text{'price})) \rightarrow T \)
OR Special Forms

- The `or` special form

\[
\begin{align*}
(\text{or} \ (= \ 3 \ 9) \ (\text{eq} \ \text{'price} \ \text{'price})) & \rightarrow T \\
(\text{or} \ 5 \ 6 \ 7) & \rightarrow 
\end{align*}
\]
OR Special Forms

- The or special form

\[
\text{(or (= 3 9) (eq 'price 'price))} \rightarrow \text{T}
\]
\[
\text{(or 5 6 7)} \rightarrow \text{5}
\]
Boolean Special Forms

Why are and and or special forms?
Boolean Special Forms

Why are and and or special forms?
- They do not evaluate all arguments apriori
Boolean Special Forms

- Why are and and or special forms?
  - They do not evaluates all arguments apriori

- Define your own version of the NULL predicate

\[
(LAMBDA (x) \\
  (COND (X nil) \\
    (t t)))
\]
Boolean Special Forms

- Why are `and`, `and`, and `or` special forms?
  - They do not evaluate all arguments apriori

- Define your own version of the NULL predicate

\[
\begin{align*}
&\text{(LAMBDA (x)} \\
&\text{(COND (X nil)} \\
&\text{(t t))} \\
&\text{(LAMBDA (x) (COND (X nil)(t t)))) nil)} \to
\end{align*}
\]
Boolean Special Forms

- Why are and and or special forms?
  - They do not evaluate all arguments apriori

- Define your own version of the NULL predicate

\[
\text{(LAMBDA (x)} \\
\text{ (COND (X nil) } \\
\text{ (t t))} \\
\text{(LAMBDA (x) (COND (X nil)(t t))) nil) → t}
\]
Boolean Special Forms

- Why are and and or special forms?
  - They do not evaluate all arguments apriori

- Define your own version of the NULL predicate

\[
(LAMBDA (x) \\
  (COND (X nil) \\
    (t t)))
\]

\[
(LAMBDA (x) (COND (X nil) (t t))) \text{nil} \rightarrow t \\
(LAMBDA (x) (COND (X nil) (t t))) \text{'(1 2 3)} \rightarrow \]
Boolean Special Forms

- Why are `and` and `or` special forms?
  - They do not evaluate all arguments apriori

- Define your own version of the NULL predicate

```lisp
(LAMBDA (x)
  (COND (X nil)
    (t t)))
```

```
(LAMBDA (x) (COND (X nil)(t t))) nil) \rightarrow t
(LAMBDA (x) (COND (X nil)(t t))) '(1 2 3)) \rightarrow nil
```
More Examples of AND and OR

gorge →
More Examples of AND and OR

george → Error: undefined variable
More Examples of AND and OR

george → Error: undefined variable
(and t nil) →
More Examples of AND and OR

gorge → Error: undefined variable
(and t nil) → nil
More Examples of AND and OR

george → Error: undefined variable
(and t nil) → nil
(and t nil george) →
More Examples of AND and OR

george → Error: undefined variable
(and t nil) → nil
(and t nil george) → nil
More Examples of AND and OR

george → Error: undefined variable
(and t nil) → nil
(and t nil george) → nil
(or t nil) →

More Examples of AND and OR

george → Error: undefined variable
(and t nil) → nil
(and t nil george) → nil
(or t nil) → t
More Examples of AND and OR

george → Error: undefined variable
(and t nil) → nil
(and t nil george) → nil
(or t nil) → t
(or t nil george) →
More Examples of AND and OR

george → Error: undefined variable
(and t nil) → nil
(and t nil george) → nil
(or t nil) → t
(or t nil george) → t
More Examples of AND and OR

gorge → Error: undefined variable
(and t nil) → nil
(and t nil george) → nil
(or t nil) → t
(or t nil george) → t
(and (atom 'tom) (null nil)) →
More Examples of AND and OR

george → Error: undefined variable
(and t nil) → nil
(and t nil george) → nil
(or t nil) → t
(or t nil george) → t
(and (atom 'tom) (null nil)) → t
More Examples of AND and OR

george → Error: undefined variable
(and t nil) → nil
(and t nil george) → nil
(or t nil) → t
(or t nil george) → t
(and (atom 'tom) (null nil)) → t
(or (atom '(a b)) (atom 'fred)) →
More Examples of AND and OR

george → Error: undefined variable
(and t nil) → nil
(and t nil george) → nil
(or t nil) → t
(or t nil george) → t
(and (atom 'tom) (null nil)) → t
(or (atom '(a b)) (atom 'fred)) → t
And More Examples of \textsf{AND} and \textsf{OR}

\[
\begin{align*}
(\text{or} & \ (\text{and} \ (\text{listp} \ 'tom) \ (\text{atom} \ 'tom))) \\
(\text{or} & \ (\text{atom} \ '(a \ b)) \ (\text{listp} \ '(a))))
\end{align*}
\]
And More Examples of AND and OR

\[
(\text{or} \ (\text{and} \ (\text{listp} \ 'tom) \ (\text{atom} \ 'tom)) \\
(\text{or} \ (\text{atom} \ '(a b)) \ (\text{listp} \ '(a))))
\]

\[\equiv (\text{or} \ (\text{and} \ \text{nil} \ t) \\
(\text{or} \ \text{nil} \ t))\]

\[\rightarrow\]
And More Examples of AND and OR

\[
\begin{align*}
&\text{(or (and (listp 'tom) (atom 'tom)) )} \\
&\ \ \ \ \ \ \ \ (\text{or (atom '(a b)) (listp '(a))) ) \\
\equiv &\text{(or (and nil t) )} \\
&\ \ \ \ \ \ \ \ (\text{or nil t) )} \\
\rightarrow &\text{ t}
\end{align*}
\]
And More Examples of AND and OR

\[
\text{(or (and (listp 'tom) (atom 'tom))}
\text{ (or (atom '(a b)) (listp '(a))))}
\equiv\text{(or (and nil t)}
\text{ (or nil t))}
\rightarrow t
\text{(and 'fred 'george)}\rightarrow
\]
And More Examples of AND and OR

(\text{or} (\text{and} (\text{listp} \ 'tom) (\text{atom} \ 'tom))
(\text{or} (\text{atom} \ '(a b)) (\text{listp} \ '(a))) )
≡(\text{or} (\text{and} \ \text{nil} \ \text{t})
(\text{or} \ \text{nil} \ \text{t}) )
→ \text{t}
(\text{and} \ 'fred \ 'george)→ \text{george}
And More Examples of **AND** and **OR**

\[
\begin{align*}
\text{(or (and (listp 'tom) (atom 'tom)))} & \quad \equiv \quad \text{(or (atom '(a b)) (listp '(a)))} \\
\equiv & \quad \text{(or (and nil t)} \\
& \quad \text{(or nil t) } \\
\rightarrow & \quad t \\
(\text{and 'fred 'george}) & \rightarrow \text{ george} \\
(\text{or 'fred 'george}) & \rightarrow \\
\end{align*}
\]
And More Examples of AND and OR

\[
\begin{align*}
(\text{or} & \ (\text{and} \ (\text{listp} \ \text{'tom}) \ (\text{atom} \ \text{'tom})) \\
& \quad (\text{or} \ (\text{atom} \ \text{'(a b)}) \ (\text{listp} \ \text{'(a))))) \\
\equiv & \ (\text{or} \ (\text{and} \ \text{nil} \ \text{t}) \\
& \quad (\text{or} \ \text{nil} \ \text{t}) \)
\end{align*}
\]
\[
\rightarrow \ \text{t} \\
(\text{and} \ \text{'fred} \ \text{'george}) \rightarrow \ \text{george} \\
(\text{or} \ \text{'fred} \ \text{'george}) \rightarrow \ \text{fred}
\]
And More Examples of AND and OR

(or (and (listp 'tom) (atom 'tom))
  (or (atom '(a b)) (listp '(a))))
≡(or (and nil t)
     (or nil t))
→ t
(and 'fred 'george)→ george
(or 'fred 'george)→ fred
(or nil (list 5 4))→
And More Examples of AND and OR

\[
\text{(or (and (listp 'tom) (atom 'tom))}
\]
\[
\quad \text{(or (atom '(a b)) (listp '(a))) )
\]
\[
\equiv \text{(or (and nil t)}
\]
\[
\quad \text{(or nil t) )}
\]
\[
\rightarrow \text{t}
\]
\[
\text{(and 'fred 'george)} \rightarrow \text{george}
\]
\[
\text{(or 'fred 'george)} \rightarrow \text{fred}
\]
\[
\text{(or nil (list 5 4))} \rightarrow \text{(5 4)}
\]
And More Examples of AND and OR

(or (and (listp 'tom) (atom 'tom))
  (or (atom '(a b)) (listp '(a))) )
≡(or (and nil t)
    (or nil t) )
→ t
(and 'fred 'george)→ george
(or 'fred 'george)→ fred
(or nil (list 5 4))→ (5 4)
((lambda (m)
   (or (+ m 1) (= m 0)) ) 0 )
→
And More Examples of AND and OR

(or (and (listp 'tom) (atom 'tom))
  (or (atom '(a b)) (listp '(a))))
≡(or (and nil t)
  (or nil t))
→ t
(and 'fred 'george)→ george
(or 'fred 'george)→ fred
(or nil (list 5 4))→ (5 4)
((lambda (m)
   (or (+ m 1) (= m 0)) ) 0 )
→ 1 ; Note: (+ m 1) has no side effect
Passing a Function as a Parameter

Conceptually (THIS CODE WILL NOT EXECUTE IN LISP)

```lisp
( (LAMBDA (fn)
   (fn 1 2)
   ) 'max )→
```
Passing a Function as a Parameter

Conceptually (THIS CODE WILL NOT EXECUTE IN LISP)

```
(LAMBDA (fn)
  (fn 1 2)
  'max )  →  2
```
Passing a Function as a Parameter

- Conceptually (THIS CODE WILL NOT EXECUTE IN LISP)

```
( (LAMBDA (fn)
    (fn 1 2)
    ) 'max ) → 2

( (LAMBDA (fn)
    (fn 1 2)
    ) '+ ) →
```
Passing a Function as a Parameter

- Conceptually (THIS CODE WILL NOT EXECUTE IN LISP)

```lisp
(defun examplefn (fn)
  (fn 1 2)
  (max)
)
```

```
> (examplefn 'max)
2
```

```
(defun examplefn (fn)
  (fn 1 2)
  (+)
)
```

```
> (examplefn '+)
3
```
Using FUNCALL in Lisp

- By convention, Lisp does not evaluate first argument (unless it is a LAMBDA)
Using **FUNCALL** in Lisp

- By convention, Lisp does not evaluate first argument (unless it is a LAMBDA)
- **FUNCALL** provides the necessary hack
Using `FUNCALL` in Lisp

- By convention, Lisp does not evaluate first argument (unless it is a LAMBDA)

- `FUNCALL` provides the necessary hack

\[
\begin{align*}
& ( \text{(LAMBDA} (\text{fn}) \\
& \quad \text{(FUNCALL} \text{ fn} \ 1 \ 2) \\
& \text{)} \\
& \quad \text{'}\text{max} \text{)} \rightarrow
\end{align*}
\]
Using `FUNCALL` in Lisp

- By convention, Lisp does not evaluate first argument (unless it is a `LAMBDA`)
- `FUNCALL` provides the necessary hack

```lisp
((LAMBDA (fn)
  (FUNCALL fn 1 2)
)

'max) → 2
```
Naming Your Own Functions

'(LAMBDA (X Y) (IF (> X Y) X Y)) ; what do I do?

▶ Define your function
Naming Your Own Functions

```
( (LAMBDA (big)

  (FUNCALL big (FUNCALL big 1 2) (FUNCALL big 3 4))
)
'(LAMBDA (X Y) (IF (> X Y) X Y)) ; what do I do?
)
```

- Define your function
- Pass to \( \lambda \) where bound to parameter 'big'
Naming Your Own Functions

( (LAMBDA (big)
    (FUNCALL big (FUNCALL big 1 2) (FUNCALL big 3 4) )
)
'(LAMBDA (X Y) (IF (> X Y) X Y)) ; what do I do?
)

▶ Define your function
▶ Pass to λ where bound to parameter 'big'
▶ Use 'big' repeatedly
Naming Your Own Functions

```
( (LAMBDA (big)
   (FUNCALL big (FUNCALL big 1 2) (FUNCALL big 3 4) )
)
'(LAMBDA (X Y) (IF (> X Y) X Y)) ; what do I do?
)

→
```

- Define your function
- Pass to λ where bound to parameter ’big’
- Use ’big’ repeatedly
- Evaluate the result!
Naming Your Own Functions

```
( (LAMBDA (big)
   (FUNCALL big (FUNCALL big 1 2) (FUNCALL big 3 4)) )

)'
(LAMBDA (X Y) (IF (> X Y) X Y)) ; what do I do?
)
→ 4
```

- Define your function
- Pass to λ where bound to parameter 'big'
- Use 'big' repeatedly
- Evaluate the result!
Named Functions with FLET

▶ A convenient short-form for naming functions

\[
(FLET ((big (x y) (IF (> x y) x y)))
   (big
      (big 1 2)
      (big 3 4))
   )
\]

→
Named Functions with FLET

- A convenient short-form for naming functions

```lisp
(FLET ((big (x y) (IF (> x y) x y)))
  (big
    (big 1 2)
    (big 3 4))
)
→ 4
```
Multiple Functions with FLET

- List as many functions as you like

\[
\text{(FLET ( (big (x y) (IF (> x y) x y))}
\text{ (sum (x y) (+ x y)))}
\text{(big}
\text{ (sum 1 2)
\text{ (sum 3 4) )
\text{ )
\text{ )
\rightarrow}
\]

Writing helper functions simplifies code and makes it more transparent
Multiple Functions with FLET

- List as many functions as you like

```
(FLET ( (big (x y) (IF (> x y) x y))
      (sum (x y) (+ x y)))
  (big
    (sum 1 2)
    (sum 3 4) )
)
)
→ 7
```
Multiple Functions with FLET

- List as many functions as you like

\[
(FLET ((big (x y) (if (> x y) x y))
         (sum (x y) (+ x y)))

(big
   (sum 1 2)
   (sum 3 4))
)

→ 7

- Writing helper functions simplifies code and makes it more transparent
Co-reference with FLET

What does this return?

```lisp
(FLET ((square (x) (* x x))
       (sos (x y) (+ (square x) (square y))))
  (sos 3 4))
```

→

25
Co-reference with FLET

What does this return?

```lisp
(FLET ( (square (x) (* x x))
   (sos (x y) (+ (square x) (square y)))
)
  (sos 3 4)
)
→ 25
```
Local Shadowing of Global Functions

- Can also redefine system functions locally

Example:

```lisp
(FLET ( (max (x y) (+ x y))
  (max 3 4) )
→ 7
```

```lisp
(FLET ( (max (x y) (- (max x y)))
  (max 3 4) )
→ -4
```
Local Shadowing of Global Functions

- Can also redefine system functions locally

```
(FLET ((max (x y) (+ x y))
  (max 3 4))

→
```
Local Shadowing of Global Functions

- Can also redefine system functions locally

\[
\begin{align*}
(FLET \ ( \ (\text{max}\ (x\ y)\ (+\ x\ y)) \\
\quad) \\
\quad\ (\text{max}\ 3\ 4) \ )
\end{align*}
\]
\[
\rightarrow 7
\]
Local Shadowing of Global Functions

- Can also redefine system functions locally

  \[
  \text{(FLET } ( (\text{max } x \ y) (+ x y)) \\
  \quad (\text{max } 3 \ 4) \ )
  \]

  \[
  \rightarrow 7
  \]

- Redefine local functions in terms of system definition
Local Shadowing of Global Functions

- Can also redefine system functions locally

\[
\text{(FLET ( (max (x y) (+ x y)))} \\
\text{ (max 3 4) )} \\
\rightarrow 7
\]

- Redefine local functions in terms of system definition

\[
\text{(FLET ( (max (x y) (- (max x y)))} \\
\text{ (max 3 4) )}
\]
Local Shadowing of Global Functions

- Can also redefine system functions locally
  
  \[(\text{FLET} \ ( (\text{max} \ (x \ y)) \ (+ \ x \ y)) ) \]
  
  \[(\text{max} \ 3 \ 4) \] → 7

- Redefine local functions in terms of system definition
  
  \[(\text{FLET} \ ( (\text{max} \ (x \ y)) \ (- \ (\text{max} \ x \ y))) ) \]
  
  \[(\text{max} \ 3 \ 4) \] →
Local Shadowing of Global Functions

- Can also redefine system functions locally

\[
(FLET ((max (x y) (+ x y))
       (max 3 4))
\rightarrow 7
\]

- Redefine local functions in terms of system definition

\[
(FLET ((max (x y) (- (max x y)))
       (max 3 4))
\rightarrow -4
\]
More Examples of Functional Args

(FLET ((applier (x y fn) (FUNCALL fn x (CDR y)))))
  (LIST
   (applier 'a '(bcd) 'CONS)
   )
  )
→ ( )
More Examples of Functional Args

(FLET ((applier (x y fn) (FUNCALL fn x (CDR y))))
  (LIST
   (applier 'a '(bcd) 'CONS)
   (applier '(t t) '(t) 'CONS)
   (applier '5 '(a) 'cons)
   (applier '(a b c) '(d e f) 'APPEND)
   (applier '(a b c) '(t) 'APPEND)
   (applier 'a '(b c) '(LAMBDA (x y) x))
)
→ ( (A)
  
  )
)
)
More Examples of Functional Args

(FLET ((applier (x y fn) (FUNCALL fn x (CDR y))))
     (LIST
      (applier 'a '(bcd) 'CONS)
      (applier '(t t) '(t) 'CONS)
      (applier '5 '(a) 'cons)
      (applier '(a b c) '(d e f) 'APPEND)
      (applier '(a b c) '(t) 'APPEND)
      (applier 'a '(b c) '(LAMBDA (x y) x)))))

→ ( (A) )
More Examples of Functional Args

(FLET ((applier (x y fn) (FUNCALL fn x (CDR y))))
  (LIST
    (applier 'a '(bcd) 'CONS)
    (applier '(t t) '(t) 'CONS)
    (applier '5 '(a) 'cons)
    (applier '(a b c) '(d e f) 'APPEND)
    (applier '(a b c) '(t) 'APPEND)
    (applier 'a '(b c) '(LAMBDA (x y) x))
  )
)
→ ((A) ((T T))
  )
More Examples of Functional Args

(FLET ((applier (x y fn) (FUNCALL fn x (CDR y)))))
(LIST
  (applier 'a '(bcd) 'CONS)
  (applier '(t t) '(t) 'CONS)
  (applier '5 '(a) 'cons)

  )
)
)
→ ((A) ((T T)))
)
More Examples of Functional Args

\[
(FLET ((applier (x y fn) (FUNCALL fn x (CDR y)))))
(LIST
 (applier 'a '(bcd) 'CONS)
 (applier '(t t) '(t) 'CONS)
 (applier '5 '(a) 'cons)
 )
\]
\[
→ ((A) ((T T)) (5))
\]
More Examples of Functional Args

(FLET ((applier (x y fn) (FUNCALL fn x (CDR y))))
      (LIST
       (applier 'a '(bcd) 'CONS)
       (applier '(t t) '(t) 'CONS)
       (applier '5 '(a) 'cons)
       (applier '(a b c) '(d e f) 'APPEND)

       ))

→ ( (A) ((T T)) (5) )
More Examples of Functional Args

(FLET ((applier (x y fn) (FUNCALL fn x (CDR y))))
   (LIST
    (applier 'a '(bcd) 'CONS)
    (applier '(t t) '(t) 'CONS)
    (applier '5 '(a) 'cons)
    (applier '(a b c) '(d e f) 'APPEND)
)
)
→ ( (A) ((T T)) (5) (A B C D E F) )
More Examples of Functional Args

(FLET ((applier (x y fn) (FUNCALL fn x (CDR y))))
    (LIST
        (applier 'a '(bcd) 'CONS)
        (applier '(t t) '(t) 'CONS)
        (applier '5 '(a) 'cons)
        (applier '(a b c) '(d e f) 'APPEND)
        (applier '(a b c) '(t) 'APPEND)
    )
    )
→ ( (A) ((T T)) (5) (A B C D E F) )
More Examples of Functional Args

```lisp
(FLET ((applier (x y fn) (FUNCALL fn x (CDR y))))
  (LIST
    (applier 'a '(bcd) 'CONS)
    (applier '(t t) '(t) 'CONS)
    (applier '5 '(a) 'cons)
    (applier '(a b c) '(d e f) 'APPEND)
    (applier '(a b c) '(t) 'APPEND)
  )
)
→ ( (A) ((T T)) (5) (A B C D E F) (A B C) )
)
More Examples of Functional Args

(FLET ((applier (x y fn) (FUNCALL fn x (CDR y))))
  (LIST
    (applier 'a '(bcd) 'CONS)
    (applier '(t t) '(t) 'CONS)
    (applier '5 '(a) 'cons)
    (applier '(a b c) '(d e f) 'APPEND)
    (applier '(a b c) '(t) 'APPEND)
    (applier 'a '(b c) '(LAMBDA (x y) x))
  )
)
)
→ ((A) ((T T)) (5) (A B C D E F) (A B C))
More Examples of Functional Args

(FLET ((applier (x y fn) (FUNCALL fn x (CDR y))))
  (LIST
    (applier 'a '(bcd) 'CONS)
    (applier '(t t) '(t) 'CONS)
    (applier '5 '(a) 'cons)
    (applier '(a b c) '(d e f) 'APPEND)
    (applier '(a b c) '(t) 'APPEND)
    (applier 'a '(b c) '(LAMBDA (x y) x))
  )
)
→ ( (A) ((T T)) (5) (A B C D E F) (A B C) A )
FLET Does Not Permit Self-Reference

Here

(FLET ((foo (x) (IF (= x 0)
                   0
                   (foo (- x 1))))
        (foo 1)))
FLET Does Not Permit Self-Reference

Here

\[
(FLET ( (foo (x) (IF (= x 0) 0 (foo (- x 1)) )) )
\]

\[
(foo 1) \)
\]

→
FLET Does Not Permit Self-Reference

Here

(FLET ( (foo (x) (IF (= x 0) 0 (foo (- x 1)) )) )
    (foo 1) )
→ ** Error: foo undefined
LABELS Permits Self-Reference

Here

(LABELS ( (foo (x) (IF (= x 0) 0 (foo (- x 1))) ))

(foo 1)

)
LABELS Permits Self-Reference

Here

```
(LABELS ( (foo (x) (IF (= x 0)
                      0
                      (foo (- x 1))
                     )))
       (foo 1))
```

→ 0

Self-reference allows functional paradigm to compute anything!
We will explore this at length in the next unit.
LABELS Permits Self-Reference

- Here

```
(LABELS ( (foo (x) (IF (= x 0)
    0
    (foo (- x 1)))
  )
  (foo 1)
)
→ 0
```
LABELS Permits Self-Reference

Here

(LABELS ( (foo (x) (IF (= x 0) 0 (foo (- x 1)) )))

(foo 1 )

→ 0

Self-reference allows functional paradigm to compute anything!
LABELS Permits Self-Reference

- Here

\[
\text{(LABELS ( (foo (x) (IF (= x 0) 0 (foo (- x 1)) ))) (foo 1) )} \rightarrow 0
\]

- Self-reference allows functional paradigm to compute anything!

- We will explore this at length in the next unit.
Summary of Pure Lisp

- A FORM is
Summary of Pure Lisp

➤ A FORM is
  ➤ Atom [constant or variable]: nil, 5, X
Summary of Pure Lisp

▶ A FORM is
  ▶ Atom [constant or variable]: nil, 5, X
  ▶ Function application: (fn f₁ ... fₙ)
Summary of Pure Lisp

- A FORM is
  - Atom [constant or variable]: nil, 5, X
  - Function application: (fn f₁ ... fₙ)
  - Quoted expr: (QUOTE s)
Summary of Pure Lisp

- A **FORM** is
  - Atom [constant or variable]: nil, 5, X
  - Function application: (fn f₁ ... fₙ)
  - Quoted expr: (QUOTE s)
  - Cond expr:
    
    (COND ((c₁ f₁)  
        :  
        (cₙ fₙ)) )
Summary of Pure Lisp

- A **FORM** is
  - Atom [constant or variable]: nil, 5, X
  - Function application: (fn f₁ ... fn)
  - Quoted expr: (QUOTE s)
  - Cond expr:
    \[
    (COND (\((c₁ f₁)\) ... (cₙ fₙ))
    \]

- A **FUNCTION** is
Summary of Pure Lisp

- **A FORM is**
  - Atom [constant or variable]: nil, 5, X
  - Function application: (fn f₁ ... fₙ)
  - Quoted expr: (QUOTE s)
  - Cond expr:
    (COND ((c₁ f₁)
            ...
            (cₙ fₙ))
  
- **A FUNCTION is**
  - Primitive atomic: CONS, CAR, EQ, ...
Summary of Pure Lisp

A FORM is

- Atom [constant or variable]: nil, 5, X
- Function application: (fn f_1 \ldots f_n)
- Quoted expr: (QUOTE s)
- Cond expr:
  
  \[
  (COND \ ((c_1 f_1)
  \vdots
  (c_n f_n))
  \]

A FUNCTION is

- Primitive atomic: CONS, CAR, EQ, \ldots
- \lambda\text{-expression}: (LAMBDA (v_1 \ldots v_n) \langle\text{form}\rangle)
Summary of Pure Lisp

- **A FORM is**
  - Atom [constant or variable]: `nil`, `5`, `X`
  - Function application: `(fn f₁ ... fₙ)`
  - Quoted expr: `(QUOTE s)`
  - Cond expr:
    
    \[
    \text{(COND ((c₁ f₁) \)
    \quad \vdots \n    \quad (cₙ fₙ))}
    \]

- **A FUNCTION is**
  - Primitive atomic: `CONS`, `CAR`, `EQ`, ...
  - λ-expression: `(LAMBDA (v₁ ... vₙ) ⟨form⟩)`
  - Variable (evaluating to Function)
Summary of Pure Lisp

- A **FORM** is
  - Atom [constant or variable]: \texttt{nil}, 5, \(X\)
  - Function application: \((\texttt{fn f}_1 \ldots \texttt{f}_n)\)
  - Quoted expr: \((\texttt{QUOTE s})\)
  - Cond expr:
    \[
    (\texttt{COND ((c}_1\texttt{f}_1)
       \ldots \\
       \cdot \\
       \texttt{(c}_n\texttt{f}_n))
    \]

- A **FUNCTION** is
  - Primitive atomic: \texttt{CONS}, \texttt{CAR}, \texttt{EQ}, ... 
  - \(\lambda\)-expression: \((\texttt{LAMBDA (v}_1 \ldots \texttt{v}_n \langle \texttt{form} \rangle)\)
  - Variable (evaluating to Function)
  - Labeled \(\lambda\)-expression: \texttt{FLET} or \texttt{LABEL}
Impure Lisp I

To facilitate marking of your code we ask you to be impure:

- Define global named functions in Lisp environment with
  ```lisp
  (SETF f '(LAMBDA (x) (- x)))
  (FUNCALL f 1)
  → -1
  ```
Impure Lisp I

- To facilitate marking of your code we ask you to be impure:

- Conceptually, define global named functions in Lisp Environment with

  \[
  (\text{SETF } f \ ' (\text{LAMBDA} (x) (- x)))
  \]

  \[
  (\text{FUNCALL} f \ 1) \rightarrow
  \]
Impure Lisp I

- To facilitate marking of your code we ask you to be impure:

- Conceptually, define global named functions in Lisp Environment with

  \[
  \text{(SETF } f \ ' (LAMBDA (x) (- x)))
  \]

  \[
  \text{(FUNCALL } f \ 1) \rightarrow -1
  \]
Impure Lisp II

- Short cut which also puts $f$ into function space

(DEFUN f (x)
  (if (> x 0)
    x
    -x))
Impure Lisp II

- Short cut which also puts \( f \) into function space

```lisp
(DEFUN f (x)
  (if (> x 0)
    x
    -x))
(f 1) → 1
```
Impure Lisp II

▶ Short cut which also puts f into function space

(DEFUN f (x)
   (if (> x 0)
      x
      -x))

(f 1) → 1

(f -1) → 1
Impure Lisp II

- Short cut which also puts f into function space

(DEFUN f (x)
   (if (> x 0)
       x
       -x))

(f 1) → 1
(f -1) → 1

- Watch out for side effects (old definitions of functions lying around)
Functional Arguments & DEFUN

- Conceptually, we can also pass functions as arguments

```lisp
(DEFUN applier (fn) (fn 5 11))
(applier '+) →
```
Functional Arguments & DEFUN

- Conceptually, we can also pass functions as arguments

(DEFUN applier (fn) (fn 5 11))

(applier ’+) → 16
Functional Arguments & DEFUN

Conceptually, we can also pass functions as arguments

(DEFUN applier (fn) (fn 5 11))

(applier ’+) → 16
(applier ’*) →
Functional Arguments & DEFUN

Conceptually, we can also pass functions as arguments

\[(\text{DEFUN} \ \text{applier} \ (\text{fn}) \ (\text{fn} \ 5 \ 11))\]

\[\text{applier ' + } \rightarrow 16\]
\[\text{applier ' * } \rightarrow 55\]
Functional Arguments & DEFUN

- Conceptually, we can also pass functions as arguments

```lisp
(DEFUN applier (fn) (fn 5 11))
```

```lisp
(applier '+) → 16
(applier '*) → 55
(applier '(LAMBDA (x y) (+ (* 2 x) (- y 3)))) →
```

> THE ABOVE WILL NOT WORK IN LISP (ok in Scheme)

Lisp does not evaluate its first argument. Write instead:

```lisp
(DEFUN applier (fn) (funcall fn 1 2))
```
Functional Arguments & DEFUN

Conceptually, we can also pass functions as arguments

(DEFUN applier (fn) (fn 5 11))

(applier ’+) → 16
(applier ’*) → 55
(applier ’(LAMBDA (x y) (+ (* 2 x) (- y 3))) ) → 18

THE ABOVE WILL NOT WORK IN LISP (ok in Scheme)

Lisp does not evaluate its first argument. Write instead:

(DEFUN applier (fn) (funcall fn 1 2))
Functional Arguments & DEFUN

- Conceptually, we can also pass functions as arguments

  \[\text{(DEFUN applier (fn) (fn 5 11))}\]

  \[\text{(applier ’+) \rightarrow 16}\]
  \[\text{(applier ’*) \rightarrow 55}\]
  \[\text{(applier ’(LAMBDA (x y) (+ (* 2 x) (- y 3))) \rightarrow 18)}\]

- THE ABOVE WILL NOT WORK IN LISP (ok in Scheme)
  Lisp does not evaluate its first argument. Write instead:

  \[\text{(DEFUN applier (fn) (funcall fn 1 2))}\]