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 ...
> (+ 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:

> (+ 3 4)
7

▶ For compactness in the lecture notes, we write:

(+ 3 4) → 7
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.

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

Lisp’s Data Structures

- In "pure" LISP all compound data is represented by “symbolic expressions” (called s-expressions, or s-exprs).
- 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.
  - g u-of-a
  - e.g. 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.

  \[(a \ b \ c)\]
  \[
  ()
  \]

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

- **Generally:** \((s_1 \ s_2 \ \cdots \ s_n), \ n \geq 0, \ s_i \text{ are s-exprs}\)

- **Special case:** \((\)\) is the *empty list*. Also called \textit{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.

- **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
  \[(\text{CONS } 1 \ 2) \rightarrow (1 \ . \ 2)\]
- 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
  \[(\text{CONS } s_0 \ (s_1 \ldots \ s_n))\]
  \[\rightarrow (s_0 \ . \ (s_1 \ldots \ s_n)) \equiv (s_0 \ s_1 \ldots \ s_n)\]

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
  \[(\text{CDR } '(s_1 \ s_2 \ s_3 \ldots \ s_n)) \rightarrow (s_2 \ s_3 \ldots \ s_n)\]
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
  
  \[(\text{FIRST } '(A \ B \ C)) \rightarrow A\]
  
  \[(\text{REST } '(A \ B \ C)) \rightarrow (B \ C)\]

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

- LISP provides abbreviations for common sequences of accessors
  
  \[(\text{CAR } (\text{CDR } '(A \ B \ C))) \equiv (\text{CADR } '(A \ B \ C))\]

- Any combination of CAR's and CDR's are defined up to 10
  
  \[(\text{CAADDR } '(A \ B \ (C) \ D )) \rightarrow C\]

- Additional accessors: FIRST, SECOND, ..., TENTH
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)
(CONS (CONS '(1 2) '(3 4)) '(5 6)) → 
   ( ( (1 2) 3 4) 5 6)

Predicates

- **Predicate** ≡ A Boolean-Valued Function
- Returns: True or False
- The Lisp representation:
  
  ```lisp
  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 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 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"

  \[(\text{EQUAL} \ 1 \ 1) \rightarrow T\]
  \[(\text{EQUAL} \ 1 \ 2) \rightarrow \text{nil}\]
  \[(\text{EQUAL} \ \text{nil} \ (\text{CDR} \ '(a))) \rightarrow T\]
  \[(\text{EQUAL} \ '(a \ b) \ '(a \ b)) \rightarrow T\]

**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
  
  \[ (= 1 \ 1) \rightarrow T \]
  \[ (= 1 \ 2) \rightarrow NIL \]
  \[ (= 'a \ 'b) \rightarrow "Error!" \]

Predicates: NULL and NOT

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

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

- NULL works like a negation operator

  \[ (\text{NULL} \ (\text{NULL} \ '1 \ 2 \ 3)) \rightarrow T \]
Predicate: NOT

▶ (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: > < = <= >=
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

- 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><code>age -&gt; 5</code></td>
</tr>
<tr>
<td>number</td>
<td>Convert ASCII digits to internal numeric representation</td>
<td><code>15 -&gt; 00001111</code></td>
</tr>
<tr>
<td>constant</td>
<td>Lookup string and convert to interned value, or create a new constant</td>
<td><code>'fred -&gt; constant125</code></td>
</tr>
</tbody>
</table>

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:

\begin{verbatim}
(SETQ a 5)
(SETQ b 6)
(CONS a b)
EVAL (CONS a b)
LOOKUP CONS \rightarrow \text{<cons-implementation>}
\{defined internally by LISP\}
EVAL a
   LOOKUP a \rightarrow 5
EVAL b
   LOOKUP b \rightarrow 6
APPLY <cons-implementation> to 5 6
\rightarrow (5 . 6)
\end{verbatim}

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 (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 \((\text{SETQ } a \ (+ \ 1 \ 1))\) 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

\((\text{SETQ } a \ 5)\)
\((\text{OR } (> a \ (+ \ 2 \ 1)) \ (> a \ 2) )\)
 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, +, ...)

- New functions are defined using the LAMBDA (λ) 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!

- 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

\[
(\textsc{lambda} \ (X \ Y) \ (+ \ X (* \ 2 \ Y)))
\]

  keyword parameters  body form
Comparison of Procedures with $\lambda$-Expressions

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

- Mathematically, it is a mapping:

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

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

  ```
  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 $\lambda$-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?

  ```
  ( (LAMBDA (X Y) (+ X (* 2 Y)) ) 4 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)):

\[ \text{EVAL}[(+ X (* 2 Y))] \]
\[ \sim + \text{ of EVAL}[X] \text{ and EVAL}[(* 2 Y)] \]
\[ \text{EVAL}[X] \sim 4 \]
\[ \text{EVAL}[(* 2 Y)] \]
\[ \sim * \text{ of EVAL}[2] \text{ and EVAL}[Y] \]
\[ \text{EVAL}[2] \sim 2 \]
\[ \text{EVAL}[Y] \sim 15 \]
\[ \sim 30 \]
\[ \sim 34 \]

Summary of λ-expressions

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

\[ \text{(LAMBDA (a1 a2 ... an) <form>) where ai are atoms} \]

▶ A λ-expression is used in the following form:

\[ \text{( (LAMBDA (a1 a2 ... an) <form>) v1 v2 ... vn)} \]

▶ Each vi is a value or argument.

▶ Evaluation:

1. Create a new environment
2. Bind each argument ai to its corresponding value vi
3. Evaluate <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
Assign \( X \leftarrow (A B) \) and \( Y \leftarrow (C) \)
Evaluate: \((\text{CONS} (\text{CAR} X) Y)\):
Lookup \text{CONS}
\( \text{EVAL}[(\text{CAR} X)] \)
\( \text{EVAL} X \sim (A B) \)
\( \sim A \)
\( \text{EVAL}[Y] \sim (C) \)
\( \sim (A C) \)

Functions vs. Forms

Functions are abstract mathematical objects with some implementation.

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

Forms are syntactic expressions which get evaluated in a context

- "\((+ X 5)\)" is a form that applies the function + to \( X \) and 5
- "\((\text{CAR} X)\)" is a form that applies the function \text{CAR} to \( X \)
- "\(((\text{LAMBDA} (X) X) 7)\)" is a form which applies \( \lambda \) to 7
More Examples of $\lambda$-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
( (LAMBDA (X) (CAR X)) ' (CONS 'A NIL) )
→CONS
( (LAMBDA (X) (CAR X)) ' (LAMBDA (X) (CAR X)) )
→LAMBDA
( (LAMBDA (X) (CAR X)) (LAMBDA (X) (CAR X)) )
→ undefined --- "LAMBDA" is not function

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 & \( \lambda \)

- The general form of the LET form is:

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

- It is equivalent to the following \( \lambda \)-form:

\[
\text{((LAMBDA (} v_1 \text{... } v_n \text{) } \langle \text{form} \rangle ) } e_1 \text{... } e_n
\]

Examples of LET and \( \lambda \)

\[
\text{(LET ((X 'A)}
\text{ (Y '(B C)))}
\text{ (CONS X Y))}
\]
\[
\rightarrow (A B C)
\]
\[
( (\text{LAMBDA (} X \text{ Y) (CONS X Y)) } 'A ' (B C))
\]
\[
\rightarrow (A B C)
\]
\[
(\text{LET ( (X 'A)}
\text{ (Y ' (B C )))}
\text{ (LET ((} Y 2 \text{))}
\text{ (CONS X Y))})
\]
\[
\rightarrow (A . 2)
\]
\[
(\text{LET ( (} X (\text{CONS (QUOTE A) NIL)} ) )
\text{ (CAR X))}
\]
\[
\rightarrow A
\]
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 →5

(QUOTE B) →B

C →"Error Undefined"

(QUOTE C) →C

(+ 1 2) →3

(QUOTE (+ 1 2)) →(+ 1 2)

(QUOTE (QUOTE A)) →'A
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:

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

Examples of LIST and QUOTE

\[
\begin{align*}
\text{LIST } t \ 5 \ &\rightarrow (t \ 5) \\
\text{LIST } (\text{QUOTE } \text{BAC}) \ &\rightarrow (\text{BAC}) \\
\text{LIST } '\text{PLUS} \ 3 \ 4 \ &\rightarrow (\text{PLUS} \ 3 \ 4) \\
\text{LIST } '(\text{PLUS} \ 3 \ 4) \ &\rightarrow ((\text{PLUS} \ 3 \ 4)) \\
\text{LIST } (\text{PLUS} \ 3 \ 4) \ &\rightarrow (7) \\
\text{CAR } (\text{LIST } '\text{A} \ '\text{B}) \ &\rightarrow \text{A} \\
\text{CDR } (\text{LIST } '\text{A} \ '\text{B}) \ &\rightarrow (\text{B})
\end{align*}
\]
Using LIST to Create \( \lambda \)-Forms

- These expressions create \( \lambda \)-forms:
  
  \[
  \text{(LIST 'LAMBDA '(x) (LIST 'CAR 'x))}
  \quad \rightarrow \quad \text{(LAMBDA (x) (CAR x))}
  \]

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

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

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

- Note the use of the constant \( t \) here to represent the always true condition

- Suppose we wish to write a “lookup” function:

  \[
  \text{(lambda (name) }
  \quad \text{(COND ( (EQ name 'bob ) 'id321) }
  \quad \text{( (EQ name 'russ) 'id452) }
  \quad \text{( (EQ name 'lisa) 'id621) }
  \quad \text{( t } \quad \text{'unknown) ))}
  \]

- Again, the constant \( t \) represents a default action
Conditional Forms: COND

▶ The general form of the COND expression is:

```
(COND (CONDITION-1 FORM-1)
       (CONDITION-2 FORM-2)
       :
       (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) )
```

▶ 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, "?"
- 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

- Only code for satisfied conditions is executed
  \[
  \text{(LAMBDA (x y)}
  \text{ (COND ( (= y 0) 'error)}
  \text{ ( t (\ x y)))))
  \]
  \[
  \text{(apply λ '(10 2)) → 2}
  \text{(apply λ '(10 0)) → 'error}
  \]
More examples of \texttt{COND} II

\begin{verbatim}
(COND (t 5)) → 5
(COND (nil 5)) → nil
(COND (t 5)
    (t 6)) → 5
(COND (nil 5)
    (t 6)) → 6
(COND (nil 3)
    (t 1)) → 1
(COND nil) → Error 'nil' should be a list
\end{verbatim}

Boolean Special Forms

- Boolean operators: \texttt{and} and \texttt{or} are implemented as special forms

- Value is value of last argument evaluated

- Both \texttt{and} and \texttt{or} implement short-circuiting
  (evaluate only enough arguments to determine truth value)

- The \texttt{and} special form

\begin{verbatim}
(and (< 6 4) (>= (/ 10 0) 0)) → T
(and (not (= y 0)) (\ x y)) →
    X/Y if y\neq0 otherwise nil
(and (> 6 4) (>= 4 2)) → T
(and 5 6) → 6
(and t (cdr '(1))) → NIL
\end{verbatim}
The or special form

\[(\text{or } (\text{eq } \text{'price } \text{'price})) \rightarrow T\]
\[(\text{or } 5 6 7) \rightarrow 5\]

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

\[(\text{LAMBDA } (x)\]
\[\quad (\text{COND } (X \text{ nil})\]
\[\quad \quad (t t))\)
\[\quad \quad \) \rightarrow t\]
\[(\text{nil) } \rightarrow \text{t}\]
\[\quad \quad \) \rightarrow \text{nil}\]
\[\quad \) \rightarrow \text{nil}\]
\[\quad \) \rightarrow \text{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
(or (atom ’(a b)) (atom ’fred)) → t

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
(or ’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)

( (LAMBDA (fn)
    (fn 1 2)
    ) 'max ) → 2

( (LAMBDA (fn)
    (fn 1 2)
    ) '+ ) → 3

Using FUNCALL in Lisp

By convention, Lisp does not evaluate first argument (unless it is a LAMBDA)

FUNCALL provides the necessary hack

( (LAMBDA (fn)
    (FUNCALL fn 1 2)
    )
    'max ) → 2
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)
  )
)
→ 4
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) \ )
\)

\(\rightarrow 7\)

► Writing helper functions simplifies code and makes it more transparent

Co-reference with FLET

► What does this return?

\[
(FLET \ (\ (\square \ (x) \ (* \ x \ x)) \\
(\sos \ (x \ y) \ (+ \ (\square \ x) \ (\square \ y))))
\]

\[(\sos \ 3 \ 4)
\]

\(\rightarrow 25\)

Dr. B. Price & Dr. R. Greiner CMPUT 325 - Lisp Basics
Local Shadowing of Global Functions

- Can also redefine system functions locally

(FLET ((max (x y) (+ x y))
)
(max 3 4) )
→ 7

- Redefine local functions in terms of system definition

(FLET ((max (x y) (- (max x y)))
)
(max 3 4) )
→ -4

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

\[(\text{FLET ( (\text{foo (x) (IF (= x 0)
0
(foo (- x 1)))
)\n)})
(foo 1))\]
→ ** Error: foo undefined

LABELS Permits Self-Reference

▶ Here

\[(\text{LABELS ( (\text{foo (x) (IF (= x 0)
0
(foo (- x 1)))
)\n)})
(foo 1))\]
→ 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**
  - Atom [constant or variable]: nil, 5, X
  - Function application: \((\text{fn } f_1 \ldots f_n)\)
  - Quoted expr: \((\text{QUOTE } s)\)
  - Cond expr:
    \[
    (\text{COND } ((c_1 f_1) \\
                  \vdots \\
                  (c_n f_n)))
    \]
- **A FUNCTION is**
  - Primitive atomic: CONS, CAR, EQ, ...
  - \(\lambda\)-expression: \((\text{LAMBDA } (v_1 \ldots v_n) \langle \text{form} \rangle)\)
  - Variable (evaluating to Function)
  - Labeled \(\lambda\)-expression: FLET or LABEL

Impure Lisp I

- To facilitate marking of your code we ask you to be impure:
  - Conceptually, define global named functions in Lisp
  - Environment with
    \[
    (\text{SETF } f \ ' (\text{LAMBDA } (x) (- x)))
    (\text{FUNCALL } f \ 1) \rightarrow -1
    \]
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

(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))