Operational Semantics for Lisp

- McCarthy, Lisp 1.5 manual
- Slonneger and Kurtz, Ch 6.1
- Pagan, Ch 5.2
Operational Semantics

- Define the language semantics by describing how the state changes
  - Essentially, we are defining an interpreter

- Goal: define o.s. for a simplified Lisp
  - Project: implement this semantics
  - **LIST** Processing: the ancestor of all functional languages
    - “Lots of Insipid and Stupid Parentheses”?

- Later: general discussion of oper. sem.
Atoms

- Atoms: numbers and literals

<atom> ::= <numeric atom> | <literal atom>

<numeric atom> ::= 
  <numeral> | -<numeral> | +<numeral>

<numeral> ::= <digit> | <numeral><digit>

<literal atom> ::= <letter>
  | <literal atom><letter>
  | <literal atom><digit>

<letter> ::= a | A | b | B | ... | z | Z

<digit> ::= 0 | 1 | 2 | ... | 9
S-Expressions

A grammar for S expressions

\[ <S\text{-exp}> ::= \text{atom} \]

\[ <S\text{-exp}> ::= ( <S\text{-exp}> . <S\text{-exp}> ) \]

- Creation and breaking of S-expressions
  - \( \text{cons}[s_1, s_2] = (s_1 . s_2) \)
  - \( \text{car}[(s_1 . s_2)] = s_1; \text{cdr}[(s_1 . s_2)] = s_2 \)
    - \text{car/cdr: undefined for atoms (i.e. error)}
  - \( \text{caar}[x] = \text{car}[\text{car}[x]]; \text{cadr} = \text{car}[\text{cdr}[x]]; \text{cdar}[x] = \text{cdr}[\text{car}[x]]; \ldots \)
Lists

- List: a special kind of S-expression
- Special atom NIL: denotes the end of a list
  - and several other things
- (s) denotes (s . NIL)
- (s t w) denotes (s . (t . (w . NIL)))
- (s t w z) denotes (s . (t . (w . (z . NIL))))
- ( ) denotes NIL
Examples

\[(A \ B \ C) = (A \ . \ (B \ . \ (C \ . \ \text{NIL})))\]

\[((A \ B) \ C) = ((A \ . \ (B \ . \ \text{NIL})) \ . \ (C \ . \ \text{NIL}))\]

\[(A \ B \ (C \ D)) = (A \ . \ (B \ . \ ((C \ . \ (D \ . \ \text{NIL})) \ . \ \text{NIL})))\]

\>((A)) = ((A \ . \ \text{NIL}) \ . \ \text{NIL})\]

\[(A \ (B \ . \ C)) = (A \ . \ ((B \ . \ C) \ . \ \text{NIL}))\]

\[\text{car}[(A \ B \ C)] = A \]
\[\text{cdr}[(A \ B \ C)] = (B \ C)\]

\[\text{cons}[A; \ (B \ C)] = (A \ B \ C) \]
\[\text{car}[[((A \ B) \ C)] = (A \ B)\]

\[\text{cadr}[(A)] = \text{NIL} \]
\[\text{car}[	ext{cdr}[(A \ B \ C)]] = B\]
More Functions

- Unary functions $f : S\text{-expr} \rightarrow S\text{-expr}$
- Unary predicate functions
  - $f : S\text{-expr} \rightarrow \{ T, \text{NIL} \}$
  - T (true) and NIL (false) are “special” atoms
- “atom” predicate function: is the S-exp an atom?
  - $\text{atom}[\text{XYZ13}] = T$
  - $\text{atom}[(A . B)] = \text{NIL}$
  - $\text{atom}[\text{car}[(A . B)]] = T$
More Functions

- "int" predicate function: is the S-exp a numeric atom (i.e., an integer)?
  - int[23] = T
  - int[XYZ] = NIL
  - int[(A B)] = NIL

- "null" predicate function: is the S-exp the atom NIL?
  - null[NIL] = T  null[()] = T
  - null[(())] = NIL
Binary Functions

- Binary functions
  - \( f : S\text{-expr} \times S\text{-expr} \rightarrow S\text{-expr} \)

- Arithmetic and relational functions
  - binary functions defined only for pairs of numeric atoms (otherwise, report an error)

- Arithmetic functions
  - \( \text{plus}[a1,a2], \text{minus}[a1,a2], \text{times}[a1,a2], \text{quotient}[a1,a2], \text{remainder}[a1,a2] \)

- Relational functions (produce T or NIL)
  - \( \text{greater}[a1,a2]; \text{less}[a1,a2] \)
Equality Function

- Binary predicate function “eq”
- Works on a pair of atoms
  - If not given atoms: error
- eq[a1,a2] = T if a1 and a2 are the same literal atom
- eq[a1,a2] = T if a1 and a2 are numeric atoms with the same value
  - eq[+4,4] = T
- eq[a1,a2] = NIL in all other cases
Writing Lisp “Programs”

- Building blocks are functions
  - The functions described earlier
  - Other “built-in” functions discussed later
  - User-defined functions
  - All of these are mathematical functions defined over the domain of S-expressions

- The entire program is a math expression which uses such functions
  - Constants & function applications; that’s it ...

- We “encode” these math functions and math expressions as S-expressions
Evaluation of Expressions

- Lisp runs in an read-eval-print loop
  - you type an S-expression (the “program”), the interpreter evaluates it, and prints the resulting value
  - The value itself is an S-expression
  - The interpreter is really a unary function $f : S\text{-expr} \rightarrow S\text{-expr}$

- Data vs. code
  - Interpreter for an imperative language: the input is code, the output is data (values)
  - In Lisp: both the code and the data are S-expressions (no clear separation)
Examples

7 → 7   T → T   nil → NIL   () → NIL

(plus (plus 3 5) (times 4 4)) → 24

The input is the math expression
plus[plus[3, 5], times[4, 4]], written as an S-expression

(plus 5 T)
Error, because plus[a1, a2] is defined only for numeric atoms

(eq t nil) → NIL   (EQ NIL NIL) → T
(EQ T T) → T   (EQ +2 (PLUS 1 1)) → T
Quoted Expressions

- Quoted S-expressions
  - e.g. (QUOTE (3 4 5)) or '(3 4 5)
  - The value is the quoted expression itself
    - i.e. the expression is not evaluated further
    - evaluation of '(3 4 5) gives us (3 4 5)
  - Evaluation of (3 4 5) results in an error
    - "Illegal function call": the interpreter treats this as function application, and complains
- For the interpreter,QUOTE is not really a function - no argument evaluation
Examples

Applying function “atom”

(ATOM '(7 . 10)) → NIL       (ATOM 7) → T

Applying function “int”

(INT (PLUS 4 5)) → T         (INT (CONS 4 5)) → NIL

Applying function “null”

(NULL NIL) → T              (NULL ()) → T
(NULL '( )) → T              (NULL '(a)) → NIL
(NULL (EQ 2 (PLUS 1 1))) → NIL
More Examples

(7 . nil) → Error

'(7 . nil) → (7)

(plus (plus 3 5) (car (quote (7 . 8)))) → 15

(CONS (CAR '(7 . 10)) (CDR '(7 . 10)))
→ (7 . 10)
Programmer-Defined Functions

- `(DEFUN  F  (X  Y)  Z)`
- Defines a new function F with formals X and Y and body Z
  - All formals are distinct literal atoms
    - Different from T and NIL
  - F is a literal atom: Different from names of built-in functions, QUOTE, DEFUN, COND
  - Constraints should be checked when DEFUN is processed (do not wait for a call to F)
- One more: DEFUN occurs only at the top level: cannot be nested in other expressions
  - For the project: this is a pre-condition
Conditional Expressions

- \((\text{COND} \ (b_1 \ e_1) \ (b_2 \ e_2) \ \ldots \ (b_n \ e_n))\)
- First evaluate \(b_1\). If not NIL, evaluate \(e_1\) and this is the value to the conditional
- If \(b_1\) evaluates to NIL, evaluate \(b_2\), etc.
- If all \(b\) evaluate to NIL: error
Examples

> (DIFF 5 6)   Error
> (DEFUN DIFF (X Y)
    (COND ((EQ X Y) NIL) (T T)))
Another example: member of a list of atoms
> (DEFUN MEM (X LIST)
    (COND ((NULL LIST) NIL)
          (T (COND ((EQ X (CAR LIST)) T)
                 (T (MEM X (CDR LIST))))))
> (MEM 3 '(2 3 4)) evaluates to T
**List Union**  \((S_1, S_2 \text{ have no duplicates})\)

\[
\text{(DEFUN UNI} \ (S_1 \ S_2) \\
\text{\quad (COND} \ ( (\text{NULL} \ S_1) \ S_2) \\
\text{\quad \quad ( (\text{NULL} \ S_2) \ S_1) \\
\text{\quad \quad (T} \ (\text{COND} \ ( (\text{MEM} \ (\text{CAR} \ S_1) \ S_2) \\
\text{\quad \quad \quad \quad \quad \quad \quad \quad \quad (\text{UNI} \ (\text{CDR} \ S_1) \ S_2))} \\
\text{\quad \quad \quad \quad \quad \quad \quad \quad \quad (T} \ (\text{CONS} \ (\text{CAR} \ S_1) \ (\text{UNI} \ (\text{CDR} \ S_1) \ S_2) ))) \\
\text{\quad \quad \quad \quad \quad \quad \quad \quad \quad ))) \\
\text{\quad \quad \quad \quad \quad \quad \quad \quad \quad ))
\]
Simplified Math Notation

\[ \text{mem}[x, \text{list}] = \begin{cases} \null[\text{list}] & \to \text{NIL} \\ \text{eq}[x, \text{car}[\text{list}]] & \to \text{T} \\ \text{T} & \to \text{mem}[x, \text{cdr}[\text{list}]] \end{cases} \]

\[ \text{uni}[s1, s2] = \begin{cases} \null[s1] & \to s2 \\ \null[s2] & \to s1 \\ \text{T} & \to \left[ \text{mem}[\text{car}[s1], s2] \to \text{uni}[\text{cdr}[s1], s2] \mid \text{T} \to \text{cons}[\text{car}[s1], \text{uni}[\text{cdr}[s1], s2]] \right] \end{cases} \]
Another Example

- Sorted list \( X \) of integers w/ duplicates
- \((\text{UNIQUE } X)\) - without the duplicates

\[ \text{unique}[x] = [ \ ? \ ] \]

How should we write this math function as a Lisp program?
Lisp Interpreter Written in Lisp

- Defined as a Lisp function `myinterpreter`
- Suppose we already had an interpreter `I`
  - Conceptually, using `I` to evaluate any S-expression `E` is the same as using `I` to evaluate the S-expression `(myinterpreter (quote E))`
- Overall approach: consider `(F e1 e2 ...)`
  - Recursively evaluate `e1`, `e2`, ...
  - Bind the resulting values `v1`, `v2`, ... to the formal parameters `p1`, `p2`, ... of `F`
    - Add pairs `(p1 . v1) (p2 . v2) ...` to an association list `(a-list)`
  - Evaluate the body of `F` using the `a-list`
Possible Representation of Functions

- `(DEFUN F param_list body)`
- Interpreter maintains an internal list of function definitions (d-list)
- The result of evaluating a DEFUN expression is the addition of a pair `(F . ( param_list . body ) )` to the d-list
- The only expression with a side effect
  - The d-list is the only “global” binding
Top-level Control

\[ \text{myinterpreter } [\exp, \text{d}] = \text{eval}[\exp, \text{NIL}, \text{d}] \]

- Invoked in a read-eval-print loop
- Every evaluation starts with no parameter bindings
- The function definition list \text{d} is the only "surviving" data structure between different invocations of function \text{myinterpreter}
  - \text{d} accumulates all function definitions

- Cleaner alternative: Slonneger Ch. 6
Key Function: eval

- $\text{eval}(exp,a,d)$: evaluates an expression $exp$ based on the current $a$-list $a$ and the current list of function definitions $d$

- Some helper functions
  - $z$: a list of $(x . y)$ pairs - could be $a$ or $d$
  - $\text{bound}(x,z)$: does $z$ contain a pair $(x . y)$?
  - $\text{getval}(x,z)$: finds the first $(x . y)$ in $z$ and returns $y$; precondition: $\text{bound}(x,z)$ is $T$
eval

\[ \text{eval}(\text{exp}, a, d) = \]

- \( \text{atom}(\text{exp}) \rightarrow \) \( \text{exp} \) \text{is an atom}  \\
- \( \text{eq}(\text{exp}, \text{T}) \rightarrow \text{T} \)
  - \( \text{eq}(\text{exp}, \text{NIL}) \rightarrow \text{NIL} \)
  - \( \text{int}(\text{exp}) \rightarrow \text{exp} \)
  - \( \text{bound}(\text{exp}, a) \rightarrow \text{getval}(\text{exp}, a) \)
  - \( \text{T} \rightarrow \text{Error! (unbound variable)} \)

- \( \text{T} \rightarrow \ldots \text{next slide} \)

\( \text{exp} \) \text{is a list}
eval (cont)

eval \[ \text{exp, } a, \text{ and } d \] =

[ \text{atom[exp]} \rightarrow \ldots \text{exp is an atom} \\
| \text{T } \rightarrow \text{exp is a list} \\
\quad \text{[ eq[car[exp]], QUOTE } \rightarrow \text{cadr[exp]} | \\
\quad \text{eq[car[exp]], COND } \rightarrow \text{evcon[cdr[exp], a, d]} | \\
\quad \text{eq[car[exp]], DEFUN } \rightarrow \text{add stuff to d-list} | \\
\quad \text{T } \rightarrow \text{apply[car[exp],} \\
\quad \text{evlist[cdr[exp], a, d],} \\
\quad \text{a, d ] ] ] ]}
Helper Functions

\[ \text{evcon}(x, a, d) = \begin{cases} \text{x is } ((b1 e1) (b2 e2) \ldots) \\ [\ null[x] \rightarrow \text{Error!} ] \\ \ \text{eval}[\text{caar}[x], a, d] \rightarrow \text{eval}[\text{cadar}[x], a, d] \end{cases} \]

\[ \begin{cases} \text{T} \rightarrow \text{evcon}[\text{cdr}[x], a, d] \end{cases} \]

\[ \text{evlist}(x, a, d) = \begin{cases} \text{null}[x] \rightarrow \text{NIL} \\ \begin{cases} \text{T} \rightarrow \text{cons}[	ext{eval}[\text{car}[x], a, d], \text{evlist}[\text{cdr}[x], a, d]] \end{cases} \end{cases} \]
Error Checking

- Not shown, but must be there
- If car[exp] is QUOTE, cdr[exp] should be a list with a single element
- If car[exp] is DEFUN, cdr[exp] should be a list with exactly three elements
  - Literal atom (function name)
  - List of distinct literal atoms (params)
  - Arbitrary expression (body)
- If car[exp] is DEFUN, exp cannot be a nested expression (but no need to check this for the project)
Key Function: apply

- **apply**[f,x,a,d]: applies a function f on a list of actual parameters x

- Helper function addpairs
  - z: a list of (x . y) pairs - the current association list
  - addpairs[xlist,ylist,z]: a new list, w/ pairs (xᵢ . yᵢ) followed by the contents of z
    - addpairs[‘(p q), ‘(1 2), ‘(r . 4) ] = ( (p . 1) (q . 2) (r . 4) )
  - Precondition: size of xlist = size of ylist
**Function Application**

apply\[ f, x, a, d \] = ; \( x \) is list of actual params

\[
[ \ \text{atom}[f] \rightarrow \\
[ \ \text{eq}[f, \text{CAR}] \rightarrow \text{caar}[x] \mid \\
\text{eq}[f, \text{CDR}] \rightarrow \text{cdar}[x] \mid \\
\text{eq}[f, \text{CONS}] \rightarrow \text{cons}[\text{car}[x], \text{cadr}[x]] \mid \\
\text{eq}[f, \text{ATOM}] \rightarrow \text{atom}[\text{car}[x]] \mid \\
\text{eq}[f, \text{EQ}] \rightarrow \text{eq}[\text{car}[x], \text{cadr}[x]] \mid \\
... \text{INT, NULL, arithmetic} ... \mid \\
T \rightarrow \text{... next slide} \] \]

*What about error checking?*

*user-defined fun*
Function Application (cont)

apply[ f, x, a, d ] = ; x is list of actual params

[ atom[f] \rightarrow

[ ... \mid

T \rightarrow \text{eval} [ \text{cdr[ getval[f,d] ]},

\text{addpairs[ car[ getval[f,d] ], x, a],}

\text{d} ] \rightarrow \text{bind the formals]

] \mid

T \rightarrow \text{Error!] More error checking?}
Dynamic Scoping

(defun f (x) (plus x y))
(defun g (y) (f 10))
(defun h (y) (f 20))

(g 5) → 15
(h 5) → 25
(g (h 5)) → 35
Observations

- Function bodies and function applications are similar to "normal" expressions
  - No difference between "values" and "code"

- The interpreter description defines an operational semantics for the language
  - Tells us how to "operate" on a given program
  - The interpreter is really a unary math function $f : S\text{-expr} \rightarrow S\text{-expr}$, and this math function defines the semantics