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
  - LISP 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-exp> ::= atom
  - <S-exp> ::= ( <S-exp> . <S-exp> )
- Creation and breaking of S-expressions
  - cons[s1,s2] = (s1 . s2)
  - car[(s1 . s2)] = s1; cdr[(s1 . s2)] = s2
  - car/cdr: undefined for atoms (i.e. error)
  - caar[x] = car[car[x]]; cadr = car[cdr[x]]; cdar[x] = cdr[car[x]]; ...

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 . NIL)))
- ((A B) C) = ((A . (B . NIL)) . (C . NIL))
- (A B (C D)) = (A . ((B . (C . (D . NIL))) . NIL))
- (()) = ((A . NIL) . NIL)
- (A B . C) = (A . ((B . C) . NIL))
- car[(A B C)]=A; cdr[(A B C)]=B
- cons[A; (B C)]=A B C; car[[(A B) C]]=A B; cdr[(()]=NIL; car[cdr[[(A B) C]]]=B
More Functions

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

More Functions

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

Binary Functions

- Binary functions
- 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)
  - $\text{greater}[a1,a2]$, $\text{lessthan}[a1,a2]$

Equality Function

- Binary predicate function "eq"
- Works on a pair of atoms
  - If not given atoms: error
  - $\text{eq}[a1,a2] = \text{T}$ if $a1$ and $a2$ are the same literal atom
  - $\text{eq}[a1,a2] = \text{T}$ if $a1$ and $a2$ are numeric atoms with the same value
    - $\text{eq}[+4,4] = \text{T}$
    - $\text{eq}[a1,a2] = \text{NIL}$ 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 a 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
    - $f : S\text{-expr} \to S\text{-expr}$
- 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
\text{plus}[\text{plus}[3,5],\text{times}[4,4]], written as an S-expression

(plus 5 T)
Error, because \text{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. \text{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 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

• (COND (b1 e1) (b2 e2) ... (bn en))
• First evaluate b1. If not NIL, evaluate e1 and this is the value to the conditional
  • If b1 evaluates to NIL, evaluate b2, etc.
  • If all b evaluate to NIL: error
Examples

\[(\text{DIFF 5 6})\] Error
\[(\text{DEFUN DIFF (X Y)}\)
\[\text{(COND ((EQ X Y) NIL) (T T))})\]
Another example: member of a list of atoms
\[(\text{DEFUN MEM (X LIST)}\)
\[\text{(COND ((NULL LIST) NIL)}\]
\[\text{(T (COND ((EQ X (CAR LIST)) T)}\]
\[\text{(T (MEM X (CDR LIST))))}))\]
\[(\text{MEM 3 (2 3 4)}\) evaluates to T

List Union \((S_1, S_2\) have no duplicates)

\[(\text{DEFUN UNI (S1 S2)}\)
\[\text{(COND ((NULL S1) S2)}\]
\[\text{(NULL S2) S1)}\]
\[\text{(T (COND ((MEM (CAR S1) S2)}\]
\[\text{(UNI (CDR S1) S2))}\]
\[\text{(T (CONS (CAR S1) (UNI (CDR S1) S2)))))}}\]

Simplified Math Notation

\[\text{mem}[x, \text{list}]= \text{Recursively-defined math function}\]
\[\text{null}[\text{list}] \rightarrow \text{NIL} |\]
\[\text{eq}[x, \text{car}[\text{list}]] \rightarrow \text{T} |\]
\[\text{T} \rightarrow \text{mem}[x, \text{cdr}[\text{list}]] \]
\[\text{uni}[s_1, s_2] =\]
\[\text{null}[s_1] \rightarrow s_2 |\]
\[\text{null}[s_2] \rightarrow s_1 |\]
\[\text{T} \rightarrow \text{[ mem}[\text{car}[s_1], s_2] \rightarrow \text{uni}[\text{cdr}[s_1], s_2] |\]
\[\text{T} \rightarrow \text{cons}[\text{car}[s_1], \text{uni}[\text{cdr}[s_1], s_2]] ]\]

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 \textit{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
    \((\text{myinterpreter} \text{ (quote } E))\)
- Overall approach: consider \((F e_1 e_2 ... )\)
  - Recursively evaluate \(e_1, e_2, ... \)
  - Bind the resulting values \(v_1, v_2, ...\) to the formal
    parameters \(p_1, p_2, ...\) of \(F\)
  - Add pairs \((p_1, v_1) (p_2, v_2) ...\) to an association list
    \((a\text{-list})\)
  - Evaluate the body of \(F\) using the \(a\text{-list}\)

Possible Representation of Functions

- \((\text{DEFUN } F \text{ param\_list body})\)
- \(\text{Interpreter maintains an internal list of function definitions (d-list)}\)
- \(\text{The result of evaluating a DEFUN expression is the addition of a pair}\)
  \((F . (\text{param\_list . body }) )\) to the d-list
- \(\text{The only expression with a side effect}\)
  - \(\text{The d-list is the only "global" binding}\)
Top-level Control

myinterpreter [exp,d] = eval[exp, NIL, d]
- Invoked in a read-eval-print loop
- Every evaluation starts with no parameter bindings
- The function definition list d is the only "surviving" data structure between different invocations of function myinterpreter
  - d accumulates all function definitions
- Cleaner alternative: Slonneger Ch. 6

Key Function: eval

- 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
  - bound[x,z]: does z contain a pair (x . y)?
  - getval[x,z]: finds the first (x . y) in z and returns y; precondition: bound[x,z] is T

eval

eval[ exp, a, d ] =
[ atom[exp] \rightarrow \text{exp is an atom} ]
[ eq[exp, T] \rightarrow T ]
[ eq[exp, NIL] \rightarrow NIL ]
[ int[exp] \rightarrow exp ]
[ bound[exp, a] \rightarrow getval[exp, a] ]
| T \rightarrow \text{Error! (unbound variable)} ]
| T \rightarrow \ldots \text{next slide} ]
| exp is a list

eval (cont)

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

Helper Functions

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

evlist[ x, a, d ] =
\begin{align*}
\text{null[x] } \rightarrow \text{NIL} & \\
T \rightarrow \text{cons[eval[car[x], a, d], evlist[cdr[x], a, d]]} & 
\end{align*}

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 \((x, y)\) 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
[ atom[f] \rightarrow
 [ \text{eq}[f, \text{CAR}] \rightarrow \text{caar}[x] | \text{eq}[f, \text{CDR}] \rightarrow \text{cdar}[x] | \text{eq}[f, \text{CON}S] \rightarrow \text{cons}[\text{car}[x], \text{cdr}[x]] | \text{eq}[f, \text{ATOM}] \rightarrow \text{atom}[\text{car}[x]] | \text{eq}[f, \text{EQ}] \rightarrow \text{eq}[\text{car}[x], \text{cdr}[x]] | ... \text{INT, NULL, arithmetic ...} | T \rightarrow ... \text{next slide} \]

Function Application (cont)

apply\[ f, x, a, d \] = ; x is list of actual params
[ atom[f] \rightarrow
 [ \text{eval} [\text{cdr}[\text{getval}[f, d]],
 addpairs[\text{car}[\text{getval}[f, d]], x, a],
 d ] \rightarrow \text{bind the formals}
 T \rightarrow Error! ] \text{More error checking?}

Dynamic Scoping

(defun f (x) (plus x y))
(defun g (y) (f 10))
(defun h (y) (f 20))
(g 5) \rightarrow 15
(h 5) \rightarrow 25
(g (h 5)) \rightarrow 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 \( f : \text{S-expr} \rightarrow \text{S-expr} \), and this math function defines the semantics