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 of the program changes.
- Essentially, the rules of an operational semantics describe the steps taken by an interpreter to execute a given input program.
- Goal: define o.s. for a simplified Lisp-like language.
  - Project: implement this semantics
  - List processing: the ancestor of all functional languages.
  - “Lots of insipid and stupid parentheses”?
- Later: general discussion of operational semantics for imperative languages.
Atoms and S-Expressions

Atoms: grammar used by the interpreter’s lexical analyzer

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

S-Expressions (symbolic expressions)

<S-exp> ::= atom
<S-exp> ::= ( <S-exp> . <S-exp> )

Alternative view: binary trees where leaves are atoms
S-Expressions and Lists

Everything is an S-expression

– The values processed by a program (inputs, outputs, intermediate values) are all S-expressions
– The Lisp program itself is also an S-expression
– The Lisp interpreter takes as input an S-expression (which contains the Lisp program and its input values) and produces an S-expression as output

Lists are special kinds of S-expressions

– The rightmost leaf is NIL
– Written in list notation
– Interpreter input can be in list notation or dot notation (for simplicity, the projects only use list notation for input)
Lists

(A) denotes (A . NIL)

(A B C) denotes (A . (B . (C . NIL)))

(A B C D) denotes (A . (B . (C . (D . NIL))))

() denotes NIL

This is all “syntactic sugar”: the list notation is simply used to help human understanding

More examples:

(((A B) C) is ((A . (B . NIL)) . (C . NIL))

(A B (C D)) is (A . (B . ((C . (D . NIL)) . NIL)))

((A)) is ((A . NIL) . NIL)

(A (B . C)) is (A . ((B . C) . NIL))
Mathematical Functions on S-Expressions

\text{cons}(s1,s2) = (s1 \ . \ s2)
\text{car}( (s1 \ . \ s2) ) = s1
\text{cdr}( (s1 \ . \ s2) ) = s2
\text{car/cdr}: \text{undefined} \text{ for atoms (interpreter should report an error at run time)}

Notes

\text{caar}(x) = \text{car}(\text{car}(x)); \text{cadr}(x) = \text{car}(\text{cdr}(x)); \text{cdar}(x) = \text{cdr}(\text{car}(x)), \text{etc.}
\text{car}(\text{cons}(s1,s2)) = s1; \text{cdr}(\text{cons}(s1,s2)) = s2
\text{cons}(\text{car}(s),\text{cdr}(s)) = s

\text{cons} = \text{construct} \text{ new object}; \text{car} = \text{contents of the address part of register number}; \text{cdr} = \text{contents of the decrement part of register number (IBM 704 computer)}
Special Case: Input is a List

car( (A B C) )=A
car( (A) )=NIL
car( ((A B) C) )=(A B)
car(cdr( (A B C) ))=B

cdr( (A) )=(B C)
cdr( (A B C) )=(B C)
car(cdr( (A B C) ))=B

car gives the first element of the list
cdr gives the rest of the list (i.e., another list)
More Math Functions on S-Expressions

Unary functions $f : S\text{-expr} \rightarrow S\text{-expr}$

Unary predicate functions: atom, int, null

\[ f : S\text{-expr} \rightarrow \{ T, \text{NIL} \} \]

T (true) and NIL (false) are “special” atoms

atom: is the S-exp an atom?

atom( XYZ13 ) = T; atom((A . B)) = NIL

int: is the S-exp a numeric atom (i.e., an integer)?

int(23) = T; int(XYZ) = NIL; int((A B)) = NIL

null: is the S-exr 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, interpreter should report an error)
  - Arithmetic functions
    $\text{plus}(a_1,a_2)$, $\text{minus}(a_1,a_2)$, $\text{times}(a_1,a_2)$
  - Relational functions (produce T or NIL)
    $\text{greater}(a_1,a_2)$, $\text{less}(a_1,a_2)$
  - Equality function “eq” works on a pair of atoms
    If not given atoms, interpreter should report an error;
    $\text{eq}(a_1,a_2) = T$ if $a_1$ and $a_2$ are the same literal atom;
    $T$ if $a_1$ and $a_2$ are numeric atoms with the same value;
    NIL otherwise
Writing Lisp “Programs”

Building blocks are functions
- The simple math functions described earlier
- Other built-in functions and constructs discussed later
- Programmer-defined functions
- All of these are math functions defined over the domain of S-expressions

The entire program is a math expression built with the help of such functions (functional programming)
- We “encode” this math expression as an S-expression
- The input to the Lisp interpreter is this encoded form
- The output is the value (i.e., S-expression) produced by evaluating the input math expression
Examples

Math view: `plus(1,2)`
  Interpreter’s input: (PLUS 1 2)
  Interpreter’s output: 3

Math view: `cons( eq(1,2) , eq(3,3) )`
  Interpreter’s input: (CONS (EQ 1 2) (EQ 3 3))
  Interpreter’s output: ( NIL . T )

Math view: apply function `f` : `f(op1, op2, ...)`
  Interpreter’s input: S-expression
    (f S-expr-for-op1 S-expr-for-op2 ...)
  Interpreter’s output:
    `f(eval(S-expr-for-op1), eval(S-expr-for-op2), ...)`
Each operand itself may be complex expression, and needs to be evaluated (by `eval`) recursively
Evaluation of Expressions

Interpreter is a function $\textbf{eval} : \text{S-expr} \rightarrow \text{S-expr}$

Operational semantics: define this math function $\textbf{eval}$

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
Programs can create/execute new code while running
Examples

\((\text{PLUS } (\text{PLUS } 3 5) \ (\text{TIMES } 4 4)) \rightarrow 24\)

The input is the math expression \(\text{plus}(\text{plus}(3,5), \text{times}(4,4))\) written as an S-expression

\((\text{PLUS } 5 \ T)\)

Error, because \(\text{plus}(a1,a2)\) is defined only for numeric atoms

\((\text{EQ } 2 \ (\text{PLUS } 1 1)) \rightarrow T\)
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)

Note: evaluation of (3 4 5) results in an error
  – “Illegal function call”: the interpreter thinks it is a function application, and complains about “3”
  – Remember that (f S-expr-for-op1 S-expr-for-op2 ...) is an encoding of math expression f(op1, op2, ...)

QUOTE is not a function – parameter is not evaluated
  – Do not get eval(S-expr-for-op) but rather S-expr-for-op
  – eval((QUOTE (EQ 1 2)))=(EQ 1 2) but eval((EQ 1 2))=NIL
Examples

Applying function \texttt{atom}

\[(\text{ATOM '}(7 10)) \rightarrow \text{T} \quad \text{(ATOM 7)} \rightarrow \text{T}\]

Applying function \texttt{int}

\[(\text{INT (PLUS 4 5)})) \rightarrow \text{T} \quad \text{(INT (CONS 4 5)})) \rightarrow \text{NIL}\]

Applying function \texttt{null}

\[(\text{NULL NIL}) \rightarrow \text{T} \quad \text{(NULL '}(A)) \rightarrow \text{NIL}\]
\[(\text{NULL (EQ 2 (PLUS 1 1))} \rightarrow \text{NIL}\]

Applying functions \texttt{car/cdr/cons}

\[(\text{PLUS (PLUS 3 5) (CAR (QUOTE (7 8)))}) \rightarrow 15\]
\[(\text{CONS (CAR '}(7 10)) (CDR '}(7 10))) \rightarrow (7 10)\]
Conditional Expressions

(COND (b₁ e₁) (b₂ e₂) ... (bₙ eₙ))

An example of a “special form” – expression that does not follow the normal rules for evaluation; QUOTE is another example

Math notation: [b₁ ➔ e₁ | b₂ ➔ e₂ | ... | bₙ ➔ eₙ]

– Evaluate b₁; if it evaluates to not NIL, evaluate e₁ and use that
– Otherwise, evaluate b₂; if it evaluates to not NIL, evaluate e₂ and use that. Otherwise, evaluate b₃, and so on
– If all bᵢ evaluate to NIL, the value is undefined
– Unlike function applications, we do not evaluate all 2n parameters
  – Do it “on demand”: first b₁, then b₂ then ... then bₖ and finally eₖ
  – This matters: what if some bᵢ has an error or an infinite loop?
    That’s OK if for some earlier bₖ we have eval(bₖ) is defined and is not NIL
Programmer-Defined Functions

**(DEFUN** **F** **(X Y ...) Z)**

Defines for the interpreter a new function F with formal parameters X/Y/... and body Z

- Formals are **distinct literal atoms** (no repetitions); different from names of built-in functions, QUOTE, DEFUN, COND, T, NIL
- F is a **literal atom**: different from names of built-in functions, QUOTE, DEFUN, COND, T, NIL

DEFUN occurs only at the top level: cannot be nested in other expressions

- For the project: this is a pre-condition (no need to check it, it is guaranteed to be true)
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 (S1, S2 have no duplicates)

(DEFUN UNI (S1 S2)
    (COND ( (NULL S1) S2)
         ( (NULL S2) S1)
         ( T (COND
                ( (MEM (CAR S1) S2)
                  (UNI (CDR S1) S2) )
                ( T (CONS
                      (CAR S1) (UNI (CDR S1) S2) ))))
  ))
Math Notation

mem(x, list) =
[ null(list) → NIL |
  eq(x, car(list)) → T |
  T → mem(x, cdr(list))
]

uni(s1 ; s2) =
[ null(s1) → s2 |
  null(s2) → s1 |
  T → ( mem(car(s1), s2) → uni(cdr(s1), s2) |
       T → cons(car(s1), uni(cdr(s1), s2)) )
]
Another Example

Given: sorted list X of integers w/ duplicates
Goal: (UNIQUE X) - without the duplicates
unique(x) = ?

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

– Goal: define a Lisp function **myinterpreter**
– Suppose we already had a Lisp interpreter I
  – We want to define **myinterpreter** such that using I to evaluate any S-expression \( E \) is the same as using I to evaluate the S-expression \((\text{myinterpreter (quote } E))\)
– Overall approach: consider \((F e_1 e_2 \ldots)\)
  – Recursively evaluate \(e_1, e_2, \ldots\)
  – “Bind” the resulting values \(v_1, v_2, \ldots\) to the formal parameters \(p_1, p_2, \ldots\) of \(F\): we do this by adding pairs \((p_1 \cdot v_1) (p_2 \cdot v_2) \ldots\) to an association list (a-list)
  – Evaluate the body of \(F\) using the a-list; when we see a parameter \(p_k\), look for pair \((p_k \cdot v_k)\)
    – Remember “env” from Q1 in the first homework?
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 \((F . (\text{param_list} . \text{body}))\) to the d-list

– The only expression with a side effect
  – side effect = changes to “memory” that can be observed by later parts of the program; pure functional programming does not have side effects (no assignments, no global data structures)
  – The d-list is the only “memory” of the program
Top-level Function

\texttt{myinterpreter (exp,d) = eval(exp,NIL,d)}

- Every evaluation starts with no parameter bindings (i.e., the a-list is empty – NIL parameter of eval)
- The function definition list d is the only “surviving” data structure between different invocations of function \texttt{myinterpreter}
  - d accumulates all function definitions

Cleaner alternative for the d-list: Slonneger Ch. 6 (pure functional programming)
Key Function: eval

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

Some helper functions used by \texttt{eval}

\texttt{z}: a list of (x . y) pairs - could be \texttt{a} or \texttt{d}

\texttt{bound(x,z)}: does \texttt{z} contain a pair (x . y)? That is, is \texttt{x} bound to a value in list \texttt{z}?

\texttt{getval(x,z)}: finds the first (x . y) in \texttt{z} and returns \texttt{y}; precondition: \texttt{bound(x,z)} is T
eval

eval(exp,a,d) =

[ atom(exp) \rightarrow \text{exp is an atom} \\
  \text{eq}(exp,T) \rightarrow T \mid T \text{ evaluates to } T \\
  eq(exp,NIL) \rightarrow NIL \mid NIL \text{ evaluates to } NIL \\
  \text{int}(exp) \rightarrow exp \mid \text{numeric evaluates to itself} \\
  \text{bound}(exp,a) \rightarrow \text{getval}(exp,a) \mid \text{literal is looked up in the } a\text{-list} \\
  T \rightarrow \text{undefined!} \mid \text{unbound literal} \\
  T \rightarrow \ldots \text{ next slide} \mid \text{exp must be a list}
**eval**

\[
\text{eval}(\text{exp}, a, d) = \begin{cases} 
\text{atom}(\text{exp}) \rightarrow \ldots & \text{exp is an atom} \\
| T \rightarrow & \text{exp must be a list} \\
[ \text{eq(car}(\text{exp}), \text{QUOTE}) \rightarrow \text{cadr}(\text{exp}) | \\
\text{eq(car}(\text{exp}), \text{COND}) \rightarrow \text{evcon}(\text{cdr}(\text{exp}), a, d) | \\
\text{eq(car}(\text{exp}), \text{DEFUN}) \rightarrow \text{add stuff to d-list} | \\
T \rightarrow \text{apply}(\text{car}(\text{exp}), \text{car}(\text{exp}) \text{ is function name} \\
\text{evlist}(\text{cdr}(\text{exp}), a, d), \\
a, d ) \end{cases}
\]

QUOTE, COND, and DEFUN are not like normal functions and they need special processing (they are “special forms”)
Helper Functions

\[ evcon(x,a,d) = \quad x \text{ is } ( (b1 e1) \ (b2 e2) \ ... ) \]
\[
\begin{array}{ll}
\text{null}(x) \rightarrow \text{Error!} & | \\
\text{eval}(	ext{caar}(x),a,d) \rightarrow \text{eval}(	ext{cadar}(x),a,d) & | \\
T \rightarrow \text{evcon}(	ext{cdr}(x),a,d) & \\
\end{array}
\]

\[ evlist(x,a,d) = \quad x \text{ is a list of actual parameters} \]
\[
\begin{array}{ll}
\text{null}(x) \rightarrow \text{NIL} & | \\
T \rightarrow \text{cons}(	ext{eval}(\text{car}(x),a,d),
\quad \text{evlist}(	ext{cdr}(x),a,d)) & \\
\end{array}
\]

Before \textbf{apply} is called, the list of actual parameters is evaluated and the list of results is used as input to \textbf{apply}
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
- If car(exp) is COND, cdr(exp) should be a list with at least one element; every element of that list is a list of length 2
- Details in Project 3 and 4 specifications; please read very carefully
Key Function: apply

apply(f,x,a,d): applies a function $f$ on a list $x$ of actual parameter values

Helper function addpairs
- $z$: a list of $(x . y)$ pairs – the current association list
- addpairs(xlist,ylist,z): a new list, w/ pairs $(x_i . y_i)$ followed by the contents of $z$
- Example: addpairs( (p q) , (1 2) , ( (r . 4) ) ) = ( (p . 1) (q . 2) (r . 4) )
- Precondition: size of xlist = size of ylist
Applying a Function

apply(f,x,a,d) = \( x \) is list of actual parameter values

[ atom(f) \rightarrow \textit{first, built-in functions} \\
  [ eq(f, CAR) \rightarrow \text{caar}(x) | \\
    eq(f, CDR) \rightarrow \text{cdar}(x) | \\
    eq(f, CONS) \rightarrow \text{cons}(\text{car}(x), \text{cadr}(x)) | \\
    eq(f, ATOM) \rightarrow \text{atom}(\text{car}(x)) | \\
    eq(f, EQ) \rightarrow \text{eq}(\text{car}(x), \text{cadr}(x)) | \\
    \ldots \text{INT, NULL, arithmetic} \ldots | \\
    T \rightarrow \ldots \text{next slide} \textit{next, user-defined functions} ]

Need error checking – e.g. CAR takes one parameter; CONS takes two parameters; EQ takes two atoms; etc.
Function Application

\[
\text{apply}(f, x, a, d) = \\
\begin{cases} 
\begin{align*}
\text{atom}(f) & \rightarrow \\
\quad \begin{align*}
\cdots & \rightarrow \\
T & \rightarrow \text{eval} \left( \text{cdr} \left( \text{getval}(f, d) \right) \right) ; \\
\text{addpairs} \left( \text{car} \left( \text{getval}(f, d) \right), x, a \right) ; \\
d & 
\end{align*}
\end{align*}
\end{cases}
\end{align*}
\]

T \rightarrow \text{Error!} \]  

*here cannot have non-atomic expr in (expr ...),*  
*but this is possible in real Lisp and in all other functional languages*

*Error checking: number of formal parameters is equal to number of actual parameters (precondition of addpairs)*
Dynamic Scoping

(DEFUN F (X) (PLUS X Y))
(DEFUN G (Y) (F 10))
(DEFUN H (Y) (F 20))

(G 5) → 15  F’s Y is resolved to G’s Y
(H 5) → 25  F’s Y is resolved to H’s Y
(G (H 5)) → 35

**Dynamic scoping**: name Y inside F can be resolved only at run time (i.e. dynamically) depending on who calls F

**Static scoping**: names can be resolved without running the program (as in C, C++, C#, Java, Scheme, Haskell)

Remember the type-checking example from attribute grammars (static scoping)