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
  - **LIS**t **P**rocessing: 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)) = ((A . NIL) . NIL)

(A (B . C)) = (A . ((B . C) . NIL))

car[(A B C)]=A

cdr[(A B C)]=(B C)

cons[A; (B C)]=(A B C)

car[((A B) C)]=(A B)

cdr[(A)]=NIL

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

More Functions

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

Binary Functions

- Binary functions
  - **f : S-expr × S-expr → S-expr**
- Arithmetic and relational functions
  - binary functions defined only for pairs of numeric atoms (otherwise, report an error)
- Arithmetic functions
  - plus[a1,a2], minus[a1,a2], times[a1,a2], quotient[a1,a2]; remainder[a1,a2]
- Relational functions (produce T or NIL)
  - greater[a1,a2]; 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-expr → S-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

- (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

> (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) ))))
 ))
```

Simplified Math Notation

Recursively-defined math function

```
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

- Sorted list X of integers w/ duplicates
- (UNIQUE X) - without the duplicates

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

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] → *exp is an atom*

[eq[exp, T] → T |

eq[exp, NIL] → NIL |

int[exp] → exp |

bound[exp, a] → getval[exp, a] |

T → Error! (unbound variable)]

| T → ... next slide *exp is a list*

eval (cont)

```
eval[ exp, a, d] =
  [  atom[exp] → …                                  exp is an atom
   | T →                                            exp is a list
     [ eq[car[exp], QUOTE] → cadr[exp]  |
       eq[car[exp], COND] → evcon[cdr[exp],a,d]  |
       eq[car[exp], DEFUN] → add stuff to d-list |
       T → apply[ car[exp],
                  evlist[cdr[exp], a, d],
                  a, d ] ] ]
```

Helper Functions

```
evcon[ x, a, d ] =           x is ( (b1 e1)  (b2 e2)  ... )
  [ null[x] → Error! |
    eval[caar[x], a, d] → eval[cadar[x], a, d]  |
    T → evcon[cdr[x], a, d]

evlist[ x, a, d ] =
  [ null[x] → NIL  |
    T → cons[eval[car[x], a, d],
                  evlist[cdr[x], a, d]   ]   ]
```

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_i . y_i) 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] →

  [eq[f, CAR] → caar[x] |

  eq[f, CDR] → cdar[x] |

  eq[f, CONS] → cons[car[x], cadr[x]] |

  eq[f, ATOM] → atom[car[x]] |

  eq[f, EQ] → eq[car[x], cadr[x]] |

  … INT, NULL, arithmetic … |

  T → … 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] →

[… |

(formal_params . body)

T → eval [cdr[getval[f,d]],

addpairs[car[getval[f,d]], x, a],

d]

bind the formals

] |

T → 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-expr → S-expr**, and this math function defines the semantics