CSE 6341: Lisp Interpreter Project, Part 3

Overview

In this project you build a subset of the Lisp interpreter, on top of the parser from Project 2. In Project 2, each input expression was represented internally as a binary tree. In traditional Lisp terminology such binary trees are called *S-expressions*. Given the S-expression representation of an input expression, your interpreter will analyze the structure of the binary tree and will evaluate it to another binary tree. Conceptually, the core component of your interpreter is a mathematical function *eval : S-expression* $\rightarrow$ *S-expression*. Here *S-expression* denotes the set of all possible S-expressions (i.e., binary trees of the form defined in Project 2). Recall from basic mathematics that $f : X \rightarrow Y$ is the notation for a function *f* with domain *X* and range *Y*.

Input and output

In Project 2, for each top-level expression, the corresponding binary tree was pretty-printed. In Project 3, for each top-level expression, you will compute the value of *eval*(s) where s is the binary tree for that input expression. The resulting value is also a binary tree. You will pretty-print this result, but now will use the "list notation" (defined later) rather than the "dot notation" that was used in Project 2.

Lists

We will refer to an S-expression as a *list* if and only if the rightmost leaf node is the special literal atom NIL. If the S-expression is just the leaf node NIL, we will refer to it as an *empty list*. The length of a list s, denoted by *length*(s), is defined as

- ✓ if s is an empty list, *length*(s) = 0
- ✓ otherwise, *length*(s) = 1+*length*(s') where s' is the right subtree of s; note that if s is a list, so is s'

Built-in Lisp functions

Before defining the specifics of the interpreter, let us define several mathematical functions that take as input one or more S-expressions and produce as result an S-expression. Specifically, consider functions *car*, *cdr*, *cons*, *atom*, *int*, *null*, *eq*, *plus*, *minus*, *times*, *less*, and *greater*, defined as follows:

car : S-expression $\rightarrow$ *S-expression* : given a binary tree, produces the left subtree. The function is **undefined** if the input binary tree contains only one node.

cdr : S-expression $\rightarrow$ *S-expression* : given a binary tree, produces the right subtree. The function is **undefined** if the input binary tree contains only one node.

cons : S-expression $\times$ *S-expression* $\rightarrow$ *S-expression* : here $\times$ denotes the Cartesian product of two sets. Given a pair of input binary trees s_1 and s_2, *cons* produces a binary tree whose left subtree is s_1 and right subtree is s_2.

atom : S-expression $\rightarrow$ *{ T, NIL }* : given a binary tree, produces a binary tree with a single node. That node is either the literal atom T, or the literal atom NIL. If the input binary tree has one node, the output tree is the literal atom T. If the input binary tree has more than one node, the output tree is the literal atom NIL.

int : S-expression $\rightarrow$ *{ T, NIL }* : given a binary tree, produces a binary tree with a single node. That node is either the literal atom T, or the literal atom NIL. If the input binary tree has one node and that node is a numeric atom, the output tree is the literal atom T. Otherwise, the output tree is the literal atom NIL.

null : S-expression $\rightarrow$ *{ T, NIL }* : given a binary tree, produces a binary tree with a single node. That node is either the literal atom T, or the literal atom NIL. If the input binary tree has one node and that node is the literal atom NIL, the output tree is the literal atom T. Otherwise, the output tree is the literal atom NIL.

eq : S-expression $\times$ *S-expression* $\rightarrow$ *{ T, NIL }* : given a pair of input binary trees s_1 and s_2, *eq* is **undefined** if s_1 has more than one node or s_2 has more than one node. If both s_1 and s_2 are the same numeric atom, the output tree is the literal atom T. If both is s_1 and s_2 are the same literal atom, the output tree is the literal atom T. Otherwise, the output tree is the literal atom NIL.

plus : S-expression $\times$ *S-expression* $\rightarrow$ *S-expression* : given a pair of input binary trees s_1 and s_2, *plus* is **undefined** if s_1 has is **not** a numeric atom or s_2 is **not** a numeric atom. Otherwise, the output tree is a numeric atom whose value is the sum of the values of the numeric atoms for s_1 and s_2. Binary functions *minus* (for subtraction) and *times* (for multiplication) are defined similarly.

less : S-expression $\times$ *S-expression* $\rightarrow$ *{ T, NIL }* : given a pair of input binary trees s_1 and s_2, *less* is **undefined** if s_1 has is **not** a numeric atom or s_2 is **not** a numeric atom. If the value of the numeric atom for s_1 is smaller than the value of the numeric atom for s_2, the output tree is the literal atom T. Otherwise the output tree is the literal atom NIL. Binary function *greater* (for >) is defined similarly.

Lisp programs
Unlike most other languages, in Lisp there is no distinction between "code" and "data". Both are S-expressions. For example, the binary tree corresponding to input string (PLUS (PLUS 5 6) (MINUS 5 20)) is a program. When given to the Lisp interpreter, this program is evaluated to another S-expression: a binary tree containing only one node, which is the numeric atom with value -4. Similarly, if the input to the interpreter is the binary tree corresponding to input string (CONS (PLUS 2 3) (CONS 8 (NULL 5))) the result of evaluating this expression is a binary tree whose dot notation is (5 . (8 . NIL)) and whose list notation is (5 8). Thus, the interpreter can be defined by a function *eval : S-expression* $\rightarrow$ *S-expression*

The definition of function *eval* is shown below. This definition uses functions *car*, *cdr*, etc. defined earlier. If any of the functions *car*, *cdr*, etc. is **undefined** in any of the cases below, function *eval* is also **undefined** in those cases.

- *eval*(T) = T : given a binary tree containing only the literal atom T, the result is that same tree
- *eval*(NIL) = NIL : given a binary tree containing only the literal atom NIL, the result is that same tree
- *eval*(s) = s if *int*(s) = T : given a binary tree containing only a numeric atom, the result is that same tree
- *eval*(s) is undefined if s is an atom and the previous three cases do not apply; note that in the next project we will modify this definition to account for literal atoms that are function parameters

  In all remaining cases listed below, s must be a list and we must have *length*(s) ≥ 2 – otherwise, *eval*(s) is **undefined**. Depending on the value of *car*(s), the following cases apply.

- *car*(s) $\in$ { PLUS, MINUS, TIMES, LESS, GREATER} : in this case *eval*(s) = *plus*(*eval*(s_1),*eval*(s_2)) if s is the binary tree for (PLUS s_1 s_2). The evaluation of (MINUS s_1 s_2), (TIMES s_1 s_2), (LESS s_1 s_2), and (GREATER s_1 s_2) is done similarly, using built-in functions *minus*, *times*, *less*, and *greater* respectively. The value of *eval*(s) is undefined if at least one of the following is true:
  - ✓ *length*(s) $\neq 3$
  - ✓ *eval*(s_1) or *eval*(s_2) is undefined
  - ✓ *eval*(s_1) or *eval*(s_2) is something other than a numeric atom

- *car*(s) = EQ : in this case *eval*(s) = *eq*(*eval*(s_1),*eval*(s_2)) if s is the binary tree for (EQ s_1 s_2). The value of *eval*(s) is undefined if at least one of the following is true:
  - ✓ *length*(s) $\neq 3$
  - ✓ *eval*(s_1) or *eval*(s_2) is undefined
  - ✓ *eval*(s_1) or *eval*(s_2) is something other than an atom

- *car*(s) $\in$ { ATOM, INT, NULL } : in this case *eval*(s) = *atom*(*eval*(s_1)) if s is the binary tree for (ATOM s_1). The evaluation of (INT s_1) and (NULL s_1) is done similarly, using built-in functions *int* and *null*. The value of *eval*(s) is undefined if at least one of the following is true:
  - ✓ *length*(s) $\neq 2$
  - ✓ *eval*(s_1) is undefined

- *car*(s) ∈ { CAR, CDR } : in this case *eval*(s) = *car*(*eval*(s_1)) if s is the binary tree for (CAR s_1). The evaluation of (CDR s_1) is done similarly, using built-in function *cdr*. The value of *eval*(s) is undefined if at least one of the following is true:
  - ✓ *length*(s) ≠ 2
  - ✓ *eval*(s_1) is undefined
  - ✓ *eval*(s_1) is an atom

- *car*(s) = CONS : in this case *eval*(s) = *cons*(*eval*(s_1),*eval*(s_2)) if s is the binary tree for (CONS s_1 s_2). The value of *eval*(s) is undefined if at least one of the following is true:
  - ✓ *length*(s) ≠ 3
  - ✓ *eval*(s_1) or *eval*(s_2) is undefined

- *car*(s) = QUOTE : in this case *eval*(s) = s_1 if s is the binary tree for (QUOTE s_1). Unlike with all other expressions, here we do not evaluate subexpression s_1. The value of *eval*(s) is undefined if *length*(s) ≠ 2.

- *car*(s) = COND : in this case *eval*(s), where s is the binary tree for (COND s_1 s_2 … s_n), is defined as follows. The value of *eval*(s) is undefined if at least one of the following is true:
  - ✓ some s_i is not a list
  - ✓ some s_i is a list such that *length*(s_i) ≠ 2

  If these conditions are false, then s is of the form (COND (b_1 e_1) (b_2 e_2) … (b_n e_n)) with $n \geq 1$. Then the evaluation is done with the following sequence of steps
  - ✓ Compute *eval*(b_1). If *eval*(b_1) is undefined, *eval*(s) is also undefined. If *eval*(b_1) ≠ NIL: return *eval*(e_1) if defined; if *eval*(e_1) is undefined, *eval*(s) is also undefined.
  - ✓ Otherwise, compute *eval*(b_2). If *eval*(b_2) is undefined, *eval*(s) is also undefined. If *eval*(b_2) ≠ NIL: return *eval*(e_2) if defined; if *eval*(e_2) is undefined, *eval*(s) is undefined.
  - ✓ Otherwise, compute *eval*(b_3) …
  - ✓ …
  - ✓ Otherwise, compute *eval*(b_n) …
  - ✓ Otherwise, *eval*(s) is undefined

- *eval*(s) is undefined if *car*(s) ∉{ PLUS, MINUS, TIMES, LESS, GREATER, EQ, ATOM, INT, NULL, CAR, CDR, CONS, QUOTE, COND }

Output

As in Project 1, all output goes to UNIX *stdout*. This includes error messages – do not print to *stderr*. For each top-level expression, compute the value of *eval*(s) where s is the binary tree for that input expression. The resulting binary tree should be pretty-printed, followed by newline. The pretty-printing will use the so-called "list notation". Given a binary tree representation, use the following rules.

1. If the tree is a leaf, just print the atom; this includes the case of NIL, which is printed as just NIL.
2. Otherwise, print "(" and follow the chain of right children from root node I_1 to its right child I_2 to its right child I_3 etc. until the last inner node I_n ($n \geq 1$) and its right child I_{n+1} (this last node I_{n+1} is a leaf). For each I_k where $1 \leq k \leq n$, print recursively the subtree rooted at the left child of I_k; if $k<n$, print a space immediately after this.
3. Then, if node I_{n+1} is NIL, print ")". Otherwise, print " . " followed by the atom in I_{n+1} followed by ")". Note the spaces around the dot. For example, if the input is (CONS 2 (CONS 3 (CONS 4 5))), the output would be (2 3 4 . 5)

Invalid input

If *eval*(s) is undefined for some top-level expression s, your interpreter should recognize this and print "ERROR: (some simple explanation)". Immediately after that, the interpreter should exit back to the OS; the rest of the input file will be ignored. Usually the reason *eval*(s) is undefined is that for some subexpression s' of s, *eval*(s') is undefined. As soon as your interpreter discovers a subexpression s' for which *eval* is undefined, it should exit to the OS; the rest of the top-level expression s is irrelevant. The interpreter should not crash on invalid input (no segmentation faults, no uncaught exceptions, etc.). Your score will partially depend on the handling of invalid input and printing error messages as described above.

Project Submission

On or before 11:59 pm, **October 3,** you should submit the following:

- One or more files for the scanner, parser, and interpreter (source code)
- A makefile Makefile such that *make* on *stdlinux* will build your project to executable form.
- A text file called Runfile containing a single line of text that shows how to run the interpreter on *stdlinux*.
- If there are any additional details the grader needs to know in order to compile and run your project, please include them in a separate text file README

Login to *stdlinux* in the directory that contains your files. Then use the following command:

submit c6341aa lab3 Makefile Runfile sourcefile1 sourcefile2 …

Make sure that you submit only the files described above: do not submit files x.o, x.class, x.doc, etc.

Important: every time you execute *submit*, it **erases all files** you have submitted previously. If you need to resubmit, submit all files, not only the changed files. If you resubmit after the project deadline, the grader cannot recover older versions you have submitted earlier – only the last submission will be seen, and it will be graded with a late penalty. If the grader asks you to make some changes, email him/her the modified files rather than resubmitting them through *submit*.

If the time stamp on your electronic submission is **12:00 am on the next day or later**, you will receive 10% reduction per day, for up to three days. If your submission is later than 3 days after the deadline, it will **not** be accepted and you will receive zero points for this project. If the grader has problems with compiling or executing your program, she/he will e-mail you; you must respond within 48 hours to resolve the problem. Please check often your email accounts after submitting the project (for about a week or so) in case the grader needs to get in touch with you.

Academic Integrity

The project you submit must be entirely your own work. Minor consultations with others in the class are OK, but they should be at a very high level, without any specific details. The work on the project should be entirely your own: all design, programming, testing, and debugging should be done only by you, independently and from scratch. Sharing your code or documentation with others is not acceptable. Submissions that show excessive similarities (for code or for documentation) will be taken as evidence of cheating and dealt with accordingly; this includes any similarities with projects submitted in previous instances of this course.

Academic misconduct is an extremely serious offense with **severe** consequences. Additional details on academic integrity are available from the Committee on Academic Misconduct (see *http://oaa.osu.edu/coamresources.html*). I strongly recommend that you check this URL. If you have any questions about university policies or what constitutes academic misconduct in this course, please contact me immediately.