CSE 6341: Lisp Interpreter Project, Part 5

Overview

In this project you will consider a slightly modified version of the language from Project 3 (i.e., without user-defined functions). The goal is to create a *static typechecker* and a *static empty-list checker* that allow us to guarantee the run-time correctness of a program without actually running that program.

We will start by defining a revised version of function *eval* which was used in earlier projects. You do **not** need to implement this version of *eval* – it is needed only to define the properties of the two static checkers. One significant restriction is that for any s, the value of *eval*(s) can only be (1) a number, (2) a boolean value, or (3) a list of numbers. That is, we will allow only values of three types: *Nat*, *Bool*, and *List(Nat)*.

Input

We will consider a very limited subset of possible Lisp behaviors. The only constructs we will use are T, NIL, CAR, CDR, CONS, ATOM, INT, NULL, EQ, PLUS, LESS, and COND. In addition, we will use the literal atom F to represent boolean value "false". NIL will be used only to represent an empty list of numbers. Thus, the values of type *Bool* are two literal atoms: T and F.

Revised definitions of built-in Lisp functions

Consider functions *car*, *cdr*, *cons*, *atom*, *int*, *null*, *eq*, *plus*, and *less*. These functions we defined in Project 3, but here we will redefine them as follows:

car : S-expression $\rightarrow$ *S-expression* : given a list of numbers, produces the first number from the list. The function is **undefined** if (1) the input list is empty, or (2) the input is not a list of numbers.

cdr : S-expression $\rightarrow$ *S-expression* : given a list of numbers, produces a list containing all but the first element, and preserves the order of elements. The function is **undefined** if (1) the input list is empty, or (2) the input is not a list of numbers.

cons : S-expression $\times$ *S-expression* $\rightarrow$ *S-expression* : given a number s_1 and a list of numbers s_2, produces a new list in which s_1 is prepended to s_2. The function is **undefined** if (1) s_1 is not a number, or (2) s_2 is not a list of numbers.

atom : S-expression $\rightarrow$ *{ T, F }* : given an input value, produces either T or F. If the input is a number or a boolean value, the output is T. If the input is a list of numbers, the output is F. The function is **undefined** if the input is not a number, a boolean value, or a list of numbers.

int : S-expression $\rightarrow$ *{ T, F }* : given an input value, produces either T or F. If the input is a number, the output is T. If the input is a boolean value or a list of numbers, the output is F. The function is **undefined** if the input is not a number, a boolean value, or a list of numbers.

null : S-expression $\rightarrow$ *{ T, F }* : given an input value, produces either T or F. If the input is not a list of numbers, the function is **undefined**. If the input is an empty list of numbers (i.e., NIL), the output is T. If the input is a non-empty list of numbers, the output is F.

eq : S-expression $\times$ *S-expression* $\rightarrow$ *{ T, F }* : given input values s_1 and s_2, produces either T or F. If s_1 and s_2 are the same number, the result is T. If s_1 and s_2 are different numbers, the result is F. The function is **undefined** if s_1 is not a number or s_2 is not a number.

plus : S-expression $\times$ *S-expression* $\rightarrow$ *S-expression* : given input values s_1 and s_2, *plus* is **undefined** if s_1 is not a number or s_2 is not a number. Otherwise, the result is a number whose value is the sum of the values of s_1 and s_2.

less : S-expression $\times$ *S-expression* $\rightarrow$ *{ T, F }* : given input values s_1 and s_2, *less* is **undefined** if s_1 is not a number or s_2 is not a number. If $s_1 < s_2$ the result is T; otherwise, it is F.

Modified rules for evaluation

The modified definition of function *eval* is shown below. Again, you do **not** have to implement this function. The definition uses the modified 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. For example, if *eval*(s) = *plus*(*eval*(s_1),*eval*(s_2)) and *plus* is undefined for inputs values *eval*(s_1) and *eval*(s_2), the value of *eval*(s) is also undefined.

- *eval*(T) = T
- *eval*(F) = F
- *eval*(NIL) = NIL
- *eval*(s) = s whenever *int*(s) = T
- *eval*(s) is undefined if s is an atom and the previous four cases do not apply

  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 first element of s, the following cases apply.

- First element of s ∈ { PLUS, LESS, EQ, CONS } : in this case *eval*(s) = *plus*(*eval*(s_1),*eval*(s_2)) if s is (PLUS s_1 s_2). The evaluation of (LESS s_1 s_2), (EQ s_1 s_2), and (CONS s_1 s_2) is done similarly. 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

- First element of s ∈ { CAR, CDR, ATOM, INT, NULL } : in this case *eval*(s) = *car*(*eval*(s_1)) if s is (CAR s_1). The evaluation of (CDR s_1), (ATOM s_1), (INT s_1), and (NULL s_1) is done similarly. The value of *eval*(s) is undefined if at least one of the following is true:
  - ✓ *length*(s) ≠ 2
  - ✓ *eval*(s_1) is undefined

- First element of s is COND : in this case s is (COND s_1 s_2 … s_n) with n≥1. Function *eval*(s) 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≥1. Then the evaluation is done with the following sequence of steps:
  - ✓ Compute *eval*(b_1). If *eval*(b_1) is undefined or is not a boolean value, then *eval*(s) is undefined
  - ✓ if *eval*(b_1) = T, then *eval*(s) = *eval*(e_1); if *eval*(e_1) is undefined, so is *eval*(s)
  - ✓ if *eval*(b_1) = F, apply the same logic to b_2: if *eval*(b_2) is undefined or is not a boolean value, then *eval*(s) is undefined; if *eval*(b_2) = T, then *eval*(s) = *eval*(e_2) if *eval*(e_2) is defined, and undefined otherwise
  - ✓ if *eval*(b_1) = *eval*(b_2) = F, apply the same logic to b_3: if *eval*(b_3) is undefined or is not a boolean value, …
  - ✓ …
  - ✓ if *eval*(b_1) = *eval*(b_2) = … = *eval*(b_{n-1}) = F, apply the same logic to b_n: if *eval*(b_n) is undefined or is not a boolean value, …
  - ✓ if *eval*(b_1) = *eval*(b_2) = … = *eval*(b_{n-1}) = *eval*(b_n) = F, then *eval*(s) is undefined

- *eval*(s) is undefined if the first element of s ∉ { PLUS, LESS, EQ, CONS, CAR, CDR, ATOM, INT, NULL, COND }

This definition of *eval* is more restrictive than the one from Project 3. For example, expressions such as (EQ 5 T), (NULL 5), and (COND ((CONS 4 NIL) T) (T 5)) will evaluate successfully in your Project 3 implementation, but will not be accepted by this definition of *eval*.

Typing relation

Our first goal is to define and implement a static typechecker that guarantees the absence of certain type-related run-time errors (i.e., cases where *eval*(s) is undefined because a subexpression is of the wrong type). The type checking should be done based on the following inference rules for the typing relation s : T

Axioms

const : *Nat* | T : *Bool* | F : *Bool* | NIL : *List(Nat)*

Note: "const" denotes a numeric atom

Other inference rules

$$\frac{s : List(Nat)}{(CAR\ s) : Nat} \qquad \frac{s : List(Nat)}{(CDR\ s) : List(Nat)} \qquad \frac{s_1 : Nat \quad s_2 : List(Nat)}{(CONS\ s_1\ s_2) : List\ (Nat)}$$

$$\frac{s : T}{(ATOM\ s) : Bool} \qquad \frac{s_1 : Nat \quad s_2 : Nat}{(EQ\ s_1\ s_2) : Bool} \qquad \frac{s : List(Nat)}{(NULL\ s) : Bool}$$

$$\frac{s : T}{(INT\ s) : Bool} \qquad \frac{s_1 : Nat \quad s_2 : Nat}{(PLUS\ s_1\ s_2) : Nat} \qquad \frac{s_1 : Nat \quad s_2 : Nat}{(LESS\ s_1\ s_2) : Bool}$$

$$\frac{b_1 : Bool \quad e_1 : T \quad b_2 : Bool \quad e_2 : T \quad \ldots \quad b_n : Bool \quad e_n : T}{(COND\ (b_1\ e_1)\ (b_2\ e_2)\ \ldots\ (b_n\ e_n)) : T}$$

In these rules, *T* denotes any of the three types *Nat*, *Bool*, and *List(Nat)*. In the last rule, we must have $n \geq 1$. In that last rule all occurrences of *T* refer to the same type – that is, there is one instance of the rule in which all *T* are replaced with *Nat*, another instance of the rule in which all *T* are replaced with *Bool*, etc.

These rules eliminate many cases for which *eval* is not defined. For example, (PLUS), (PLUS 3 4 5), and (PLUS T 5) are not well typed and will be rejected by the typechecker. However, not every well typed program is guaranteed to execute without run-time errors. For example, (CAR NIL) is well typed but *eval* is not defined for it. A similar example is (COND (F 5) (F 6)).

Not all executable programs are well typed. In particular, the last rule forces all expressions e_i to have the same type. This is needed to be able to do static type checking, but it is not really part of the definition of *eval*. For example, (COND (F 5) (T NIL)) would execute successfully at run time (*eval* will produce NIL) but will not be accepted by the type checker. This trade-off is typical for static analyses.

Note: this typechecker can be easily defined using abstract interpretation, as done for the second checker.

Static checking for potentially-empty lists

For this language it is possible to prove statically that a program will not fail with empty-list errors for *car* and *cdr*. We can do this by computing a **static lower bound on the length of a run-time list**. You will implement a static checker that (1) computes such a lower bound for each expression of type *List(Nat)*, and (2) checks that this lower bound is greater than zero for parameters s_1 in all (CAR s_1) and (CDR s_1). **Assume that the input to this checker is a well typed program**.

A precise definition of the static empty-list checker can be formulated as **abstract interpretation**. With this definition, the implementation of the checker can be easily obtained by modifying slightly the implementation of the interpreter from Project 3. The modified semantics is based on a set of **abstract values** { *True*, *False*, *AnyBool*, *AnyNat*, *List[≥0]*, *List[≥1]*, *List[≥2]*, … }. Here *List[≥k]* is an abstraction of any run-time list with length ≥ *k*. The **abstracted function** *eval'* is defined as follows:

- *eval'*(T) = *True*
- *eval'*(F) = *False*
- *eval'*(NIL) = *List[≥0]*
- *eval'*(s) = *AnyNat* when s is a numeric atom
- *eval'*(s) is undefined if s is an atom and the previous four cases do not apply
- For all other cases, *eval'*(s) is defined similarly to *eval*(s), but using abstracted functions *car'*, *cdr'*, *cons'*, *null'*, *atom'*, *int'*, *eq'*, *plus'*, *less'*
  - ✓ *car'* is defined as follows: *car'*(*List[≥0]*) is **undefined**; *car'*(*List[≥k]*) is *AnyNat* for *k=1,2,…*. For *List[≥0]*, the function is undefined because the list could be empty. If the abstract interpreter (i.e., static checker) observes an expression whose value is undefined, it will report an error – it will statically report that there may be a possible run-time error.
  - ✓ *cdr'* is defined as follows: *cdr'*(*List[≥0]*) is **undefined**; *cdr'*(*List[≥k]*) is *List[≥(k-1)]* for *k=1,2,…*. Input *List[≥0]* means that the list could be empty. As another example, if the input to *cdr'* is *List[≥9]*, the result is *List[≥8]* because the list length is reduced by 1.
  - ✓ *cons'* is defined as follows: *cons'*(*AnyNat*, *List[≥k]*) is *List[≥(k+1)]* because the list length increases by 1
  - ✓ *null'* is defined as follows: *null'*(*List[≥0]*) is *AnyBool*; *null'*(*List[≥k]*)=*False* for *k*≥1. Input *List[≥0]* means that the list could be empty or non-empty, and "regular" function *null* may produce T or F. As another example, if the input to *null'* is *List[≥9]*, the result is *False* because statically we can assert that the list cannot be empty.
  - ✓ *atom'* is defined as follows: *atom'*(*True*) = *True*, *atom'*(*False*) = *True*, *atom'*(*AnyBool*) = *True*, *atom'*(*AnyNat*) = *True*, *atom'*(*List[≥k]*) = *False*
  - ✓ *int'* is defined as follows: *int'*(*True*) = *False*, *int'*(*False*) = *False*, *int'*(*AnyBool*) = *False*, *int'*(*AnyNat*) = *True*, *int'*(*List[≥k]*) = *False*
  - ✓ *eq'*(*AnyNat*,*AnyNat*) = *AnyBool*
  - ✓ *plus'*(*AnyNat*,*AnyNat*) = *AnyNat*
  - ✓ *less'*(*AnyNat*,*AnyNat*) = *AnyBool*

- For (COND (b_1 e_1) (b_2 e_2) … (b_n e_n)) with *n*≥1, *eval'* is defined as follows. First, compute *eval'*(b_i) for 1≤*i*≤*n*. If any one of these is undefined, *eval'* is undefined for the entire COND expression. Otherwise, consider two mutually-exclusive cases:
  - ✓ Case 1: for all 1≤*i*≤*n* we have *eval'*(b_i) ≠ *AnyBool*. Thus, statically we know precisely the run-time values of all b_i. Consider the smallest *j* such that *eval'*(b_j) = *True*. Compute *eval'*(e_j) and return it; if this *eval'*(e_j) is undefined, *eval'* is undefined for the entire COND. If all *eval'*(b_i) = *False*, *eval'* is undefined for the entire COND.
  - ✓ Case 2: there is at least one *i* such that *eval'*(b_i) = *AnyBool*. In this case we will conservatively assume that any "branch" of the COND could be taken at run time. Compute *eval'*(e_i) for 1≤*i*≤*n*. If any one of these is undefined, *eval'* is undefined for the entire COND. Otherwise, the following subcases apply:
    - Case 2.1: All *eval'*(e_i) = *AnyNat*; then return *AnyNat*
    - Case 2.2: All *eval'*(e_i) = *True*; then return *True*
    - Case 2.3: All *eval'*(e_i) = *False*; then return *False*
    - Case 2.4: *eval'*(e_i) are some combination of *True*, *False*, and *AnyBool* values but Cases 2.2 and 2.3 do not apply; then return *AnyBool*
    - Case 2.5: *eval'*(e_i) are *List[≥k_1]*, *List[≥k_2]*, …, *List[≥k_n]*; in this case *eval'* for the entire expression is *List[≥p]* where *p* = min(*k_1*, *k_1*, … , *k_n*). For example, if *n*=3 and the values are *List[≥8]*, *List[≥2]*, and *List[≥12]*, the result is *List[≥2]*

Note: This handling of COND is not as precise as it could be. For example, consider this expression: (CAR (COND (T (CONS 8 NIL)) ((EQ 1 2) NIL))). When "regular" *eval* is applied to it, the result is 8. However, the abstracted function *eval'* is undefined because *eval'* applied to (EQ 1 2) produces *AnyBool*, and thus the abstract evaluation of (COND (T (CONS 8 NIL)) ((EQ 1 2) NIL)) will consider *List[≥1]* and *List[≥0]* for the two branches of the COND and will "merge" them into *List[≥0]*. Therefore *eval'* of the CAR expression would be undefined and the static checker will report a potential "CAR-on-empty-list" run-time error. It is possible to improve the precision by considering *eval'*(b_i) in order: for example, if *eval'*(b_1) = *True*, just consider *eval'*(e_1) and ignore the rest of the COND sub-expressions. Such improvements are beyond the scope of this project.

Output

For each top-level expression, apply the typechecker to see if the expression is well typed. If it is not, print an error message "TYPE ERROR: …." to *stdout* and exit to the OS. If it is, run the empty-list checker. If it fails – that is, *eval'* is undefined for that input – print an error message "EMPTY LIST ERROR: …" to *stdout* and exit to the OS. If the top-level expression passes both checks, pretty-print the expression to *stdout* using the approach from Project 3.

Project Submission

On or before 11:59 pm, **November 21 (Tuesday)**, you should submit the following:

- One or more files for the interpreter (source code)
- A makefile Makefile such that *make* on *stdlinux* will build your project to executable form
- A text file 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 lab5 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.