These notes show how to define a static empty-list checker. This checker can guarantee the run-time correctness of a program without actually running that program. The checker is designed based on the general concept of abstract interpretation. **You are not responsible for this material for the midterm.**

We will start by defining a revised version of function `eval` for a simple subset of the language used in Project 3. We will then define the notion of a well-typed program. Finally, we will show how to perform static checking for empty lists in any well-typed program.

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)`.

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 the following redefined functions `car`, `cdr`, `cons`, `atom`, `int`, `null`, `eq`, `plus`, and `less`.

- `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`.

The modified `eval` function can be defined based on these modified functions. The only new aspects of this definition are the following. First, `eval(F) = F`. Second, for (COND `(b_1 e_1)` `(b_2 e_2)` … `(b_n e_n)`), if we need to apply `eval` to any `b_i`, the result should be `T` or `F` but nothing else.

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
It is easy to define 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). This can be done based on the following inference rules for the typing relation s : T

Axioms

\[
\begin{align*}
\text{const} & : \text{Nat} & T & : \text{Bool} & F & : \text{Bool} & \text{NIL} & : \text{List(\text{Nat})}
\end{align*}
\]

Note: “const” denotes a numeric atom

Other inference rules

\[
\begin{align*}
\frac{s : \text{List(\text{Nat})}}{(\text{CAR } s) : \text{Nat}} & & \frac{s : \text{List(\text{Nat})}}{(\text{CDR } s) : \text{List(\text{Nat})}} & \frac{s_1 : \text{Nat} \quad s_2 : \text{List(\text{Nat})}}{(\text{CONS } s_1 \ s_2) : \text{List(\text{Nat})}}
\end{align*}
\]

\[
\begin{align*}
\frac{s : T}{(\text{ATOM } s) : \text{Bool}} & & \frac{s_1 : \text{Nat} \quad s_2 : \text{Nat}}{(\text{EQ } s_1 \ s_2) : \text{Bool}} & \frac{s : \text{List(\text{Nat})}}{(\text{NULL } s) : \text{Bool}}
\end{align*}
\]

\[
\begin{align*}
\frac{s : T}{(\text{INT } s) : \text{Bool}} & & \frac{s_1 : \text{Nat} \quad s_2 : \text{Nat}}{(\text{PLUS } s_1 \ s_2) : \text{Nat}} & \frac{s_1 : \text{Nat} \quad s_2 : \text{Nat}}{(\text{LESS } s_1 \ s_2) : \text{Bool}}
\end{align*}
\]

\[
\begin{align*}
\frac{b_1 : \text{Bool} \quad e_1 : T \quad b_2 : \text{Bool} \quad e_2 : T \quad \ldots \quad b_n : \text{Bool} \quad e_n : T}{(\text{COND } (b_1 \ e_1) \ (b_2 \ e_2) \ \ldots \ (b_n \ e_n)) : T}
\end{align*}
\]

In these rules, T denotes any of the three types Nat, Bool, and List(Nat). In the 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 empty-list checker shown below.

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. We will define 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_i in all (CAR s_i) and (CDR s_i). We will 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'

\(\begin{align*}
\text{null'} & \text{ is defined as follows: } \text{null'}(\text{List}[≥0]) = \text{AnyBool; null'}(\text{List}[≥k]) = \text{False for } k ≥ 1. \text{ Input List}[≥0] \text{ means that the list could be empty. As another example, if the input to null'} \text{ is List}[≥9], the result is List}[≥8] \text{ because the list length is reduced by 1.} \\
\text{atom'} & \text{ is defined as follows: } \text{atom'}(\text{True}) = \text{True}, \text{atom'}(\text{False}) = \text{True}, \text{atom'}(\text{AnyBool}) = \text{True}, \text{atom'}(\text{AnyNat}) = \text{True}, \text{atom'}(\text{List}[≥k]) = \text{False} \\
\text{int'} & \text{ is defined as follows: } \text{int'}(\text{True}) = \text{False}, \text{int'}(\text{False}) = \text{False}, \text{int'}(\text{AnyBool}) = \text{False}, \text{int'}(\text{AnyNat}) = \text{True}, \text{int'}(\text{List}[≥k]) = \text{False} \\
\text{eq'}(\text{AnyNat}, \text{AnyNat}) = \text{AnyBool} \\
\text{plus'}(\text{AnyNat, AnyNat}) = \text{AnyNat} \\
\text{less'}(\text{AnyNat, AnyNat}) = \text{AnyBool}
\end{align*}\)

For (COND (b₁ e₁) (b₂ e₂) … (bₙ eₙ)) with n ≥ 1, eval' is defined as follows. First, compute eval'(bᵢ) 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'(bᵢ) ≠ AnyBool. Thus, statically we know precisely the run-time values of all bᵢ. Consider the smallest j such that eval'(bᵢ) = True. Compute eval'(eᵢ) and return it; if this eval'(eᵢ) is undefined, eval' is undefined for the entire COND. If all eval'(bᵢ) = False, eval' is undefined for the entire COND.
- Case 2: there is at least one i such that eval'(bᵢ) = AnyBool. In this case we will conservatively assume that any “branch” of the COND could be taken at run time. Compute eval'(eᵢ) 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'(eᵢ) = AnyNat; then return AnyNat
  - Case 2.2: All eval'(eᵢ) = True; then return True
  - Case 2.3: All eval'(eᵢ) = False; then return False
  - Case 2.4: eval'(eᵢ) 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: \( \text{eval}'(e_i) \) are \( \text{List}[\geq k_1] \), \( \text{List}[\geq k_2] \), …, \( \text{List}[\geq k_n] \); in this case \( \text{eval}' \) for the entire expression is \( \text{List}[\geq p] \) where \( p = \min(k_1, k_2, \ldots, k_n) \). For example, if \( n = 3 \) and the values are \( \text{List}[\geq 8] \), \( \text{List}[\geq 2] \), and \( \text{List}[\geq 12] \), the result is \( \text{List}[\geq 2] \).

Note: This handling of COND is not as precise as it could be. For example, consider this expression: \( \text{CAR} \left( \text{COND} \left( \text{T} \ (\text{CONS} \ 8 \ \text{NIL}) \ ((\text{EQ} \ 1 \ 2) \ \text{NIL}) \right) \right) \). When “regular” \( \text{eval} \) is applied to it, the result is 8. However, the abstracted function \( \text{eval}' \) is undefined because \( \text{eval}' \) applied to \( \text{EQ} \ 1 \ 2 \) produces \( \text{AnyBool} \), and thus the abstract evaluation of \( \text{COND} \left( \text{T} \ (\text{CONS} \ 8 \ \text{NIL}) \ ((\text{EQ} \ 1 \ 2) \ \text{NIL}) \right) \) will consider \( \text{List}[\geq 1] \) and \( \text{List}[\geq 0] \) for the two branches of the COND and will “merge” them into \( \text{List}[\geq 0] \). Therefore \( \text{eval}' \) 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 \( \text{eval}'(b_i) \) in order: for example, if \( \text{eval}'(b_1) = \text{True} \), just consider \( \text{eval}'(e_1) \) and ignore the rest of the COND sub-expressions.

Possible refinement

Suppose we wanted to refine this abstract interpreter to distinguish between zero and non-zero numeric values. In real-world static correctness checkers, such analysis is used, for example, to prove the absence of null pointer errors in C programs.

We first need to refine the set of abstract values: instead of just having \( \text{AnyNat} \) we should have \( \text{Zero} \), \( \text{Pos} \), and \( \text{AnyNat} \). The abstracted function \( \text{eval}' \) is changed as follows. First, \( \text{eval}'(0) = \text{Zero} \) and \( \text{eval}'(n) = \text{Pos} \) for any \( n > 0 \). Next, we need to refine the primitive functions as follows:

- \( \text{atom}'(\text{Zero}) = \text{True} \) and \( \text{atom}'(\text{Pos}) = \text{True} \)
- \( \text{int}'(\text{Zero}) = \text{True} \) and \( \text{int}'(\text{Pos}) = \text{True} \)
- \( \text{eq}'(\text{Zero},\text{Zero}) = \text{True} \)
- \( \text{eq}'(\text{Zero},\text{Pos}) = \text{False} \)
- \( \text{eq}'(\text{Pos},\text{Zero}) = \text{False} \)
- \( \text{eq}'(\text{Pos},\text{Pos}) = \text{AnyBool} \)
- \( \text{eq}'(\text{AnyNat},...) = \text{AnyBool} \) and \( \text{eq}'(...,\text{AnyNat}) = \text{AnyBool} \)
- \( \text{plus}'(\text{Zero},\text{Zero}) = \text{Zero} \)
- \( \text{plus}'(\text{Zero},\text{Pos}) = \text{Pos} \)
- \( \text{plus}'(\text{Pos},\text{Zero}) = \text{Pos} \)
- \( \text{plus}'(\text{Pos},\text{Pos}) = \text{Pos} \)
- \( \text{plus}'(\text{AnyNat},...) = \text{AnyNat} \) and \( \text{plus}'(...,\text{AnyNat}) = \text{AnyNat} \)
- \( \text{less}'(\text{Zero},\text{Zero}) = \text{False} \)
- \( \text{less}'(\text{Zero},\text{Pos}) = \text{True} \)
- \( \text{less}'(\text{Pos},\text{Zero}) = \text{False} \)
- \( \text{less}'(\text{Pos},\text{Pos}) = \text{AnyBool} \)
- \( \text{less}'(\text{AnyNat},...) = \text{AnyBool} \) and \( \text{less}'(...,\text{AnyNat}) = \text{AnyBool} \)

In addition, for Case 2 of \( \text{COND} (b_1 \ e_1) (b_2 \ e_2) \ldots (b_n \ e_n) \), earlier we had the following: if all \( \text{eval}'(e_i) = \text{AnyNat} \) then return \( \text{AnyNat} \). This can be refined as follows:

- if all \( \text{eval}'(e_i) = \text{Zero} \) then return \( \text{Zero} \)
- if all \( \text{eval}'(e_i) = \text{Pos} \) then return \( \text{Pos} \)
- in all other cases, return \( \text{AnyNat} \)