PSI Phase 1

For Phase 1, you will implement a basic calculator with Lisp syntax—it will implement a small subset of PSI. You'll only be implementing a small portion of the PSI Specification in this phase, but I recommend reading the entire document.

An example session is below ($ is the command line prompt):

```
$ ./psi
psi> 1
1
psi> (+ 2 3)
5
psi> (* 4 (+ 5 6))
44
psi> (/ 137 9)
15
psi> (= 18 (* 3 6))
#t
psi> (+ 3 #t)
$error{(type-error + 2 number #t)}
psi> (+ 2 (/ 1 0))
$error{division-by-zero}
psi> (quit)
$
```

You'll need to implement the `bool`, `number`, `symbol`, `function`, `list` and `error` types:

- `number` and `bool`, for computation.
- `symbol` and `list`, to represent user code.
- `function`, for the six builtin functions you'll be implementing.
- `error`, for the error cases—a "wrapper" that contains another value.

For Phase 1, it's sufficient to return an error of the correct type. That is, for `(+ 3 #t)`, you may return `$error(type-error)` if you prefer.

You'll structure your interpreter as an interactive program that is run at the console. In a loop, it is responsible for:

- **reading** input from the user as an S-expression, e.g. "`(+ (- 4 5) (* 6 7))`", and parsing it.
- **evaluating** the S-expression fully, e.g.
  - `(+ (- 4 5) (* 6 7))` → `(+ -1 (* 6 7))`
  - `(+ -1 (* 6 7))` → `(+ -1 42)`

    - `(+ -1 42)` → `41`
- **printing** the final result to the console.

The loop ends when the `quit` function, one of the six you'll implement for this phase, is called.

For the arithmetic operations (+, -, *, /) you only need to handle arity of 2 for now, but you're welcome to handle all arities—as the spec explains, `(* 3 4 5)` → `60` and `(+)` → `0`. You'll throw a type error if a non-numeric operand is given.

You should implement equality (the = function) correctly for the number, symbol, and bool types. Equality is always false across type boundaries, e.g., `(= 1 #t)` → `#f`.

Finally, `quit` has strict arity of 0 and exits the REPL.

You have a lot of flexibility in terms of how you implement this program and its internal data structures. For example, the S-expression tree structure you will be reading—this is known as an abstract syntax tree, but Lisp's concrete syntax *is* its abstract syntax—is a nested list; you can implement it as a linked list, or as a resizable array, or as some other data structure.

The PSi Spec details the functioning of the evaluator. For Phase 1, we only have to consider standard forms, and the evaluator works like so:

- atoms evaluate to themselves; that is `1` → `1`, `#t` → `#t`.
- the empty list also evaluates to itself; `()` → `()`.
- symbols evaluate to their bindings; for now, the only bindings to consider are for the six builtin functions you must implement.
    - a symbol without a binding results in an `unbound` error.
- a general list (`<expr-0>` `<expr-1>` ... `<expr-N>`) is evaluated **recursively** like so:
    - the subexpressions are evaluated in order, left to right;
        - if any evaluate to errors, the first error is the whole evaluation's result
    - otherwise, naming the results of expressions, (`<val-0>` `<val-1>` ... `<val-N>`)...
        - if `<val-0>` is a function, and `<val-1>` ... `<val-N>` match in type and number, call it with those values as arguments.
        - otherwise, throw an `inapplicable-head` error.
    - you are responsible for deleting intermediate computational results once they are no longer reachable—that is, preventing memory leaks.

You may assume that each line of user input will never exceed 4096 characters. When the input line contains a syntax error, you should return an error—in Phase 1, you are not responsible for returning the exact same syntax error as the reference implementation, but you return one and your interpreter should remain operational.

Some other hints:

- You'll probably want to create a tagged union (design pattern, not necessary C `union`) type of some kind for your PSI values.
- You should allocate PSI values on the heap. It's feasible to do Phase 1 using only the stack, but you'll need to use dynamic memory for Phases 2 and 3. This means your functions should probably use pointer-based signatures, e.g.:
    - `pval* pval_number(int64_t n)`
    - `pval* pval_bool(bool b)`
    - `pval* pval_error(pval* x)`
    - `pval* pval_empty_list()`
    - `void pval_add(pval* list, pval* elem)`
    - `void pval_print(pval* pv)`
    - `pval* pval_copy(pval* pv)`
    - `void pval_delete(pval* pv)`
    - `...`
- As long as your implementation is usable, you won't be graded on efficiency.
- For the Phase 1 Check-In—to be released 18 Feb, due 2 Mar—you'll be given 20 lines to feed to your REPL, and you'll be telling me what it returns. You'll also be asked to explain how you checked for memory leaks and how confident you are that none exist.

Start early.