1/6/25 DRAFT: This document may be updated. Later versions supersede earlier ones.

PSI Specification

You will implement an interpreter in C for a Lisp dialect called PSI. This project's purpose is:

- to develop your C skills by having you build a nontrivial program.
- to teach you what high-level languages do in order to execute your code.

PSI Values and Types

Every value in PSI (or "pval") is in one of eight types. Four of them are "base" types, all immutable:

- `boolean`—a value that is either true or false.
  - e.g., `#t`, `#f`
- `number`—a 64-bit signed integer.
  - e.g., `0`, `1`, `-9223372036854775808`
- `string`—a sequence of 8-bit characters.
  - e.g., `""`, `"cat"`, `"I'm Farisa. One S. Not 'Miss Farisa' and certainly not 'Professor' anything."`
- `symbol`—a symbolic value.
  - e.g., `+` (function name), `division-by-zero` (error condition), `if` (special form), `x` (user-defined variable name), `_` (special symbol).

There are three wrapper types that may contain other PSI values:

- `list`—an immutable ordered collection that contains zero-or-more pvals.
  - e.g., `()`, `(2 3)`, `(#t 4 (5 6 7))`
- `cell`—a mutable "reference cell" that contains exactly one pval.
  - like C pointers, they are read/writeable "places" in memory.
- `error`—represents a runtime program fault, and contains exactly one pval.
  - e.g., `$error{division-by-zero}`

Last of all, the `function` type includes both the *builtin* functions that your interpreter will expose, like addition:

```
psi> +
$builtin{+}
psi> (+ 5 6)      ;; apply the builtin function + to args 5, 6
11
```

... and user-defined functions, called closures or lambda-expressions:

```
psi> ((fn (x) (* x x)) 7)
49
psi> (def g (x y) (+ (* x x) (* y y)))
$lambda((x y) (+ (* x x) (* y y)))@g
psi> (g 5 6)
61
```

The Interpreter (or "Read, Eval, Print Loop")

Consider the input and output to the interactive-mode interpreter (or REPL) below:

```
psi> (= (* 6 7) 42)
#t
```

This happens in three stages:

- **read**: the user string `"(= (* 6 7) 42)\n"` is converted into the pval `(= (* 6 7) 42)`
- **eval**: the pval `(= (* 6 7) 42)` is evaluated, resulting in the PSI boolean `#t`
- **print**: the PSI value `#t` is printed as `"#t"` to the console.

This is done in a loop—hence, the name "read, eval, print loop" or REPL.

Parsing and Reading

You can assume that user strings—that is, code to be executed—will never contain the null terminator, `'\0'`, except possibly at the end. When a string is read, it is parsed into tokens of the following kinds:

- **whitespace**: a substring of characters for which `isspace` is true (e.g., spaces, tabs, and newlines.)
- **comment**: a semicolon (`';'`) followed by any number of non-newline characters, followed by a newline (`'\n'`) or end-of-input.
- **boolean literal**: the strings `"#t"` or `"#f"`
- **numeric literal**: strings of digits, optionally prefixed with characters `'+'` or `'-'`.
- **string literal**: the double-quote character `'"'` followed by any number of characters up to the first unescaped double-quote.
- **symbols**: substrings corresponding to legal symbols, as defined below.
- **list markers**: the single characters, `'('` and `')'`
- **shorthand quote**: the character `'\''`.
- **invalid content**: any non-whitespace substring that cannot be tokenized in any category above.

Matching is always maximal—therefore, `"-137"` is always tokenized as a single numeric literal (never two numeric literals, e.g., -13 and 7.)

Consider the user string:

```
"'(= (= (* 7 8) 57) #f)  ; asdf"
```

A full tokenization of this is as follows:

```
Shorthand_Quote
ListL Symbol("=") WS
      ListL Symbol("=") WS
            ListL Symbol("*") WS Number(7) WS Number(8) ListR WS
            Number(57)
      ListR WS
      Bool(false)
ListR
WS
Comment
```

Whitespace and comments are inactive, or nonproductive—they don't generate PSI values. Symbols generate single PSI values, as do literals (boolean, numeric, and string) in the expected way—for example, the `Number(57)` token corresponds to a pval of type `number` with value 57.

When the parser encounters a list-opener token—the character `'('`— it continues parsing pvals and storing them in the list until it reaches the list-closer token, `')'`. Note that, as we see above, lists may contain other lists—to arbitrary depth.

When the reader encounters the shorthand quote, the reader wraps the next pval (call it <x>) in a two-element list, `(quote <x>)`, to exploit the special form quote that halts further evaluation.

Parsing an invalid-content token, or a list closer without a previous matching list opener, triggers an `invalid-token` error. Unclosed lists—such as when parsing `"(+ 1 2"`—and unclosed string literals, as well as unused shorthand quotes, will trigger an `incomplete-parse` error.

Thus, the internal representation of the pval produced by parsing the sequence above will be something like:

```
List([Symbol("quote"),
      List([Symbol("="),
            List([Symbol("="),
                  List[Symbol("*"), Number(7), Number(8)],
                  Number(57)]),
```

```
            Boolean(false)])
    ])
```

The function `parse` takes a string and:

- (1) if the string is empty, returns `#f`.
- (2) if the leading token is inactive, removes it and returns the remainder.
- (3) if the leading token is active, tries to parse one pval.
    - (3a) if successful, returns a 2-element list of the parsed pval and the string's remainder.
    - (3b) if an `invalid-token` error occurs, throws it.
    - (3c) if a list, string literal, or quoted form remains incomplete, throws an `incomplete-parse` error.

Examples below:

```
psi> (parse "")                          ; 1
#f
psi> (parse "     (+ 1 (* 2 3)) 4")      ; 2
"(+ 1 (* 2 3)) 4"
psi> (parse "(+ 1 (* 2 3)) 4")           ; 3a
((+ 1 (* 2 3)) " 4")
psi> (parse "#q")                        ; 3b
$error{(invalid-token "#q")}
psi> (parse "(+ 1 2 3")                  ; 3c
$error{(incomplete-parse "(+ 1 2 3")}
```

The function `read` is similar, but parses until it finds exactly one pval and, if it finds one and only whitespace remains, returns it. Otherwise, `read` throws a `value-error`.

String Literals

Interpreting boolean and numeric literals is straightforward, but string literals require escaping, just as in languages like Python and C. For example, the string with contents:

```
« a"b »
```

or, numerically:

```
 {97, 34, 98}
```

needs a representation in PSI that makes clear the fact that the embedded double-quote does ***not*** close out the literal. We use the following backslash-based escaping conventions:

```
rule   0: \0   -> character 0
```

```
rule  10: \n   –> character 10 (newline)
rule  34: \"   –> character 34 (")
rule  92: \\   –> character 92 (\)
rule 255: \xZW –> character 0xZW in hexadecimal ; e.g. \xff –> char 255
```

If we encounter a backslash not fitting any such pattern, it becomes an ordinary backslash. Thus, both of the string literals `"cat\dog"` and `"cat\\dog"` read as the literals for the same 7-character string « `cat\dog` » but the latter representation, the one with correct escaping, is canonical and should be preferred when printing.

We see escaping used below.

```
psi> "a\"b"
"a\"b"
psi> (ord "a\"b")
(97 34 98)
psi> "a\0b"
"a\x00b"
psi> (ord "a\0b")
(97 0 98)
```

Evaluation

To understand evaluation, we first have to understand evaluation. Yes, it's recursive.

The base cases are "self-evaluating" types—booleans, numbers, and strings—for which there is no further interpretation: 3 is 3 is 3.

```
psi> 3
3
psi> #t
#t
psi> "cat"
"cat"
```

The same applies to functions, cells, and error objects—there is no more evaluation to be done on them.

A symbol, however, exists to refer to something else—thus, the interpreter gives it the *associated action* of value lookup.

```
psi> x                 ; nothing yet bound to x -- error
$error{(unbound x)}
psi> (def x 17)        ; create global binding of x to 17
```

```
17
psi> x                  ; x can now be eval'd
17
psi> (* x 2)
34
```

What if you want to use, rather than evaluate, a symbol? Quotation exists for that.

```
psi> (= x 17)            ; true -- a global binding x : 17 exists
#t
psi> (= 'x 17)           ; false -- the symbol x and 17 are different
#f
psi> (= 'x x)
#f
psi> (= 'x 'x)
#t
```

Evaluation, on a quoted form, removes one level of quoting but does nothing else, e.g.:

```
psi> x
17
psi> 'x
x
psi> ''x
'x
psi> '''x
''x
```

The interactive mode supports a special symbol _ that holds the last value successfully returned:

```
psi> (* 2 37)
74
psi> (+ _ 1)
75
psi> (+ _ 1)
76
```

New global bindings can be added with the special form `def`, but there are existing ones created in the interpreter for all of its builtin functions. For example, when we evaluate the symbol +, we get the function object it refers to, because the binding was created at interpreter startup.

```
psi> +
$builtin{+}
```

Lists also have an associated action; for symbols, it's lookup—for lists, it's function application, with one exception: the empty list `()` is self-evaluating.

Most PSI code, as you've seen, is expressed using S-expressions, or arbitrarily nested lists, like so:

```
psi> (+ 1 2 (* 3 4))
15
```

This enables us to represent complex functional expressions without the need for a global order-of-operations—the nesting of the lists makes the ordering precise.

When we want to refer to a list, but not evaluate it, we'll use quotation:

```
psi> '(+ 1 2 (* 3 4))
(+ 1 2 (* 3 4))
```

If we want to force evaluation on a list like the one above, the `eval` function does that.

```
psi> (eval '(+ 1 2 (* 3 4)))
15
```

Code instances that invoke regular function calls are known as *standard forms*. Below are some standard forms:

```
psi> (+ 3 4)
7
psi> (* 5 (+ 6 7))
65
psi> (eval '(- 8 9))
-1
psi> ((if #f + *) 4 5)
20
```

The last of these is unusual, as it's a standard form that contains a nonstandard (or *special*) form—there is no function called `if`, as we'll later discuss.

Standard Form Evaluation

Standard forms are determined by exclusion—if the head of the list is *not* a symbol corresponding either to a known special form (e.g., `if`, `def`, `quote`) and is not a user-defined macro, then it is a standard form by default.

To evaluate a standard form `(e0 e1 ... eN)`:

- evaluate all subexpressions `e0 –> v0` , `e1 –> v1`, ... , `eN –> vN` until either...
- if any argument returns an error, return ("throw") the leftmost one.
- otherwise, if `v0` is not a function, return an `inapplicable-head` error.
- otherwise, apply `v0` to arguments `v1`, ... , `vN` and return that value.

For example, when we evaluate `(* (+ 3 5) 17)`, the process is:

```
#0:  * –> $builtin{*}
#1:  (+ 3 5) –> ...
        #0:   + –> $builtin{+}
        #1:   3 –> 3
        #2:   5 –> 5
        apply($builtin{+}, 3, 5) = 8
#2:  17 –> 17
apply($builtin{*}, 8, 17) = 136
```

thus `(* (+ 3 5) 17) –> 136`

Now consider an error case: `(* (- 7 3) (+ 5 6 #t) (/ 1 0))` is evaluated like so:

```
#0: * –> $builtin{*}
#1: (- 7 3) –> 4
#2: (+ 5 6 #t) –> $error{(type-error + 3 number #t)}
#3: (/ 1 0) –> $error{division-by-zero}        ;; if it were called
```

thus `(* (- 7 3) (+ 5 6 #t) (/ 1 0)) –> $error{(type-error + 3 number #t)}`

We never evaluate `e3`, nor do we apply the outer function, *, because of that error. This is called propagation—errors "flow up" through function calls, the assumption being that the entire computation is corrupted by the error. In this context, we refer to errors being "thrown" like exceptions, but PSI has no special (exceptional) control flow—instead, throwing an error and returning it are identical and the terms will be used interchangeably.

The policy above is called eager evaluation; even if a function's arguments are irrelevant, they are all evaluated. For example, the function `ignore` defined below never uses its argument, but error propagation still occurs:

```
psi> (def ignore (x) 0)
$lambda({x} 0)@ignore
psi> (ignore (/ 1 0))
$error{division-by-zero}
```

There are language constructs that can't be implemented using eager functions. Conditional execution (`if`) is one of them—after determining the truth value of the condition, we want to evaluate only one of the two branches, thus avoiding unwanted and potentially unsafe computations. A standard form or ordinary function, due to eager evaluation, can't do this.

Consider, with `x` bound to `0`, the form `(if (!= x 0) (/ 1 x) "Nope, not doing that.")`

```
#0: recognized as special form "if" (evaluate #2 or #3 based on condition #1)
#1:  (!= x 0) –> ...
        #0:   != –> <builtin !=>
        #1:   x  –> 0
        #2:   0  –> 0
        apply(<builtin !=>, 0, 0) = #f
#2: ; not evaluated -- condition false
#3: "Nope, not doing that." –> "Nope, not doing that."
```

... thus, the entire form returns `"Nope, not doing that."`

The `try` special form breaks the rule that errors must propagate:

```
psi> (try (+ 7 14))
(#t 21)
psi> (try (/ 1 0))
(#f division-by-zero)
```

The first element of the 2-element list is whether evaluation succeeded (returned a non-error value) and the second, on success, is that value—on failure, it is the pval inside the error object.

The `quote` form breaks the rule of recursive evaluation by telling the evaluator, when it reaches a quoted list or symbol, to remove the quote but—as if it had reached a self-evaluating value, go go no further:

```
psi> x          ; (def x 17) above
17
psi> (quote x)
x
psi> 'x         ; identical shorthand for (quote x)
x
psi> (+ x 5)
22
psi> '(+ x 5)   ; shorthand for (quote (+ x 5))
(+ x 5)
```

Symbols

Symbols' names, like strings, are sequences of characters, but subject to restrictions. Only the following characters can occur in a symbol:

- upper- and lowercase alphabetical characters
- the digit characters: `0123456789`
- `+`, `-`, `*`, `/`, `%`, `=`, `<`, `>`, `!`, and `:`
- in special cases, `&`, `_`, and `$`

No symbol may begin with a digit character, and any symbol beginning with a non-alphanumeric character must contain only non-alphanumeric characters.

Thus, symbols like `x1`, `++`, `a<B<c`, and `once-more-with-rizz` are legal; `+d`, `-e`, `7c` and `6+` are not. `-19` is legal—as a numeric literal—but not as a symbol.

Furthermore, the characters _ and & are only permitted in the symbols _ and &, which have special meanings. The `$` "reader-reject" character is legal in symbols, but only those produced by a specific function—`new-symbol`, whose purpose is to generate a symbol that can exist nowhere else in the program. Thus, the reader must treat it as illegal if it occurs in user code.

Equality

It is legal to compare values of different types for equality, but the value is always `#f`—PSI has no notion of cross-type equality; two values in different types are always unequal.

```
psi> (= 1 #t)
#f
psi> (= "a" 'a)
#f
```

Values in the base types—`bool`, `number`, `string`, `symbol`—are equal if and only if they are identical:

```
psi> (= "x" "x")
#t
```

Lists are equal if they have the same length and contain equal elements at each position. Thus, these five types have *value equality*. They are equal if they hold the same value.

```
psi> (= '(/ 1 0) '(/ 1 0))
#t
```

You can consider the error type to have value equality, but propagation (= is a function) makes the matter irrelevant.

```
psi> (= (/ 1 0) (/ 1 0))
$error{division-by-zero}
```

Reference cells, on the other hand, do not use value but *object equality*—they are equal only if they are the same cell. Whether they hold the same value or not is irrelevant—they might not in the future.

```
psi> (def a (cell 0))
$cell{0}@0x600001c14140
psi> (def b (cell 0))
$cell{0}@0x600001c1c1c0
psi> (def c a)
$cell{0}@0x600001c14140
psi> (= a b)    ; same value, different cell
#f
psi> (= a c)    ; two names, one object
#t
```

Functions also use object equality—they are the same only if they are the same object. Both cases below show functions that behave identically for all inputs, but the interpreter cannot be expected to derive arbitrary equivalencies (in fact, it's mathematically impossible.)

```
psi> (= (fn (x) x) (fn (x) x))
#f
psi> (= (fn (& xs) (eval (cons '+ xs))) +)
#f
```

Printing

The function `str` can be used to convert values to strings, and `print` can be used to print them to the console.

```
psi> (str 35)
"35"
psi> (str #f)
"#f"
psi> (print 35)
35
#t
```

The last example *prints* `35` to the console—it does I/O—and then *returns* `#t`, because every PSI function must return something. The interleaving of the interactive mode shows both, but if `(print 35)` were part of an intermediate computation, or if we were in a noninteractive mode, only the `35` would be printed, like so:

```
psi> (do (print 36) (print 37) "Done.")
36
37
"Done."
```

Both `str` and `print` add quotes and escaping to a string, producing a canonical PSI literal. If you want to write a string verbatim to the console, use `output` instead.

```
psi> (str "cat")
"\"cat\""
psi> (print "cat")
"cat"
#t
psi> (output "cat" "\n")
cat
#t
```

Symbols, as we've seen, print as unquoted string-like sequences. Since only a restricted set of characters are legal in symbols, C `printf` can be used on them—unlike with PSI strings, in which 0 is a valid character—as long as the buffer is null-terminated.

```
psi> (str 'a)
"a"
```

Lists are converted into strings using the canonical form—single spaces separating the elements, regardless of how they were represented in user code (since whitespace is thrown away.)

```
psi> (str '(1  2  3
4 ))
"(1 2 3 4)"
```

Cells, which are newly created using the function `cell`, have an output syntax of:

`$cell{<value>}@<address>`:

```
psi> (cell 8)
$cell{8}@0x600001698340
```

Addresses are nondeterministic—you will never be held responsible for attaining specific values. For this reason, cells are not readable—there's no way for the reader to guarantee that a specific memory address is available, so a "cell literal" wouldn't make sense.

Errors, likewise, are not readable but print in a specific, canonical way regarding their enclosed value:

```
$error{<value>}
```

```
psi> (error '(new-error-class "1 = 2"))
$error{new-error-class "1 = 2"}
```

A function's printing format depends on whether it is a builtin or a user-defined function—a closure. For the former, it is:

```
$builtin{<name>}
```

For the latter, it is:

```
$lambda{<arglist> <body>}[@<name>]
```

For example, see below:

```
psi> +
$builtin{+}
psi> (fn (x) x)
$lambda{(x) x}
psi> (fn this (x) x)
$lambda{(x) x}@this
```

Error Types

A `division-by-zero` error—the error object wraps the symbol—is thrown by / or % whenever a zero divisor or modulus is used.

```
psi> (/ 1 0)
$error{division-by-zero}
```

An `(unbound <sym>)` error—the error object wraps a list—is thrown when an attempt is made to evaluate an unbound symbol.

An `(arity-error <fn-sym> <expected> <observed>)` is thrown when a function is given an unacceptable number of arguments; `<expected>` is usually a list of form `(<cmp> <number>)`. See below:

```
psi> (-)         ; requires at least one argument
$error{(arity-error - (>= 1) 0)}
psi> (% 4 5 6 7) ; requires exactly two arguments
$error{(arity-error % (= 2) 4)}
```

A `(type-error <fn-sym> <position> <exp-type> <value>)` is thrown whenever a function is given a value of an unacceptable type.

```
psi> (% 15 "cat")
$error{(type-error % 2 number "cat")}
```

This indicates that the error was in the #2 position, that a `number` was expected, but that `"cat"`, was found.

A `value-error` is thrown when the value—possibly the argument, possibly an interior value—is in the correct type, but illegal for the function. Here are two examples:

```
psi> (head ())                  ; the empty list has no first element
$error{(value-error head ())}
psi> (chr '(99 97 116 256))     ; 256 is not a valid ASCII code
$error{(value-error chr 256)}
```

The `inapplicable-head` error is thrown when a non-function occurs in the #0 position of a standard form—below, 1 is not a function and cannot be “applied” to the arguments 2 and 3.

```
psi> (1 2 3)
$error{inapplicable-head}
```

A `protected-symbol` error is thrown when we try to create a new global binding for a builtin function:

```
psi> (def + 48)
$error{(protected-symbol +)}
```

Environments

Symbol lookups occur in the context of an *environment*, which is a key–value data structure whose keys are symbols and whose values can be arbitrary PSI values. When the interpreter starts, the *global environment* contains only the builtin functions, like so:

```
{+ : $builtin{+}, - : $builtin{-}, quit: $builtin{quit}, ... }
```

The special form `def` adds bindings to the global environment, so that after:

```
psi> (def a 5)
5
psi> (def b (+ a 7))
12
psi> (* a b)
60
```

our global environment is:

```
{a : 5, b : 12, + : $builtin{+}, ... , _ : 60}
```

Recall that _ is the special symbol bound to the most recent successful computation.

New bindings on the same name in the same environment replace the old ones. When an object is unreachable from any environment, ***it must be deleted***. The names of the builtin functions cannot be rebound in the global environment—however, creating new local bindings for them is allowed.

```
psi> ((fn (+) (+ 5 6)) *)          ; this is OK
30
psi> (def + *)                     ; but this is not
$error{(protected-symbol +)}
```

Local environments can be created using the `let` special form, which creates a temporary local binding—below, `a : 1`—that is available when evaluating the body of the form—the `(+ a 6)`—but disappears once we exit the `let`-form.

```
psi> (let a 1 (+ a 6))
7
psi> a
$error{(unbound a)}
```

You can think of this in terms of stack frames. The `let`-form creates a new environment in which `{a : 1}`, whose parent is the global environment, and "pushes" it on the stack. When we later evaluate `(+ a 6)` we have to do two lookups—a is found in the local environment, but + is not, so we must consult the global environment (the local one's parent) to find its binding. Upon exiting the `let`-form, we "pop" the stack frame and return to the global environment.

When bindings conflict, we always use the most local binding. Consider the case below:

```
psi> (def b 5)
5
psi> (do (print b)
```

```
          (let b 7 (do (print b) (let b 9 (print b)) (print b)))
          (print b))
5
7
9
7
5
#t
```

When we're inside the innermost `let`-form, our environment looks like this.

```
e2 = {b : 9} with parent e1
e1 = {b : 7} with parent e0
e0 = {b : 5, + : $builtin{+}, ... } with no parent
```

Thus, a lookup of `b` is going to return 9. However, when we exit that inner let form, we return to environment `e1`, and therefore will print 7.

Functions and Closures

The builtin functions—except `eval`—have no environmental dependencies; they work the same in all environments. However, user-defined functions may depend on their environments—for example, `(fn (a) (+ a b))` creates a *closure* that must know which `a` and which `b` to add together.

The case of `a`, a parameter, we handle using local variables, exactly as is done with `let`, above. A function call creates a new local environment. In fact, `(let <var> <expr> <body>)` and `((fn (<var>) <body>) <expr>)` are always equal—if you implement macros, you may legally define `let` in this way.

However, a function might be defined—a *function object* (or "closure") created—in a different environment from the one where it is used. Consider, for example:

```
psi> (def f (let b 1 (fn (a) (+ a b))))
$lambda{(a) (+ a b)}
psi> (f 7)
8
```

In order to work properly, `f` must retain access to the `b` that exists locally (due to the `let`-form) when it is created, even though this becomes inaccessible to the rest of the program on exit from the `let`-form. Thus, a function object needs to be able to *close over*—to preserve—its creation-time environment. Therefore, in defining semantics for user-defined functions, we need to differentiate between creation time and use (or call) time.

Closure Creation

The `fn` special form has two arities, `fn/2` and `fn/3`:
`(fn <arglist> <body>)`
`(fn <name> <arglist> <body>)`

They are identical except for the fact that one creates a name for the closure; the other one doesn't. When invoked, this special form:

- checks that the name, if provided, is a `symbol`—if it isn't one, raise a `type-error`.
- checks that the argument list (or "arglist") is legal:
    - it must be a `list`,
    - all elements must be `symbol`s,
    - there must be no duplicates, and
    - if & is included, it must be the second-to-last symbol;
    - if any of these properties are violated, throw an `arglist-error`.
- If no errors occur, create a function object (closure) storing:
    - the name, if one was provided.
    - the argument list.
    - the body—can be any PSI value.
    - a "frozen"—it will never change—copy of the creation-time environment.

The frozenness of a function's copied environment means that future bindings will never change its behavior. See below:

```
psi> (def b 6)
6
psi> (def f (fn () b))    ; closure copies full env., captures binding
$lambda{() b}
psi> (f)
6
psi> (def b 7)            ; global environment now binds b to 7
7
psi> (f)                  ; f's version of b unchanged
6
```

Closure Application

At call time, closures create temporary environments whose parents are the "frozen" copies of the ones in which they were defined—that is, a new frame is pushed. For each parameter, the variable name is bound to the value of the argument. Thus, in the case below:

```
psi> ((fn (x) (+ x 1)) (+ 8 9))
18
```

... the body is evaluated in an environment whose top frame includes the binding `{x : 17}`.

If there is no & symbol in the argument list, the function's arity is strictly the number of symbols in it. Thus, `(fn (x y z) 0)` takes exactly 3 arguments and throws an `arity-error` otherwise. However, if an & symbol—only legal as the second-to-last symbol in an arglist—is included, then the last symbol is a "rest" parameter that binds to a list of optional arguments.

For example, `(fn (x y & rest) rest)` accepts all arities 2+:

```
psi> ((fn (x y & rest) rest) 1 2)
()
psi> ((fn (x y & rest) rest) 1 2 3 4 5)
(3 4 5)
psi> ((fn (x y & rest) rest) 1)
$error{(arity-error (>= 2) 1)}
```

Unless a closure is a macro—more on that later—it is a standard form, so all of the subforms are evaluated. When f is a closure, to evaluate `(f v1 v2 ... vN)`:

- transition from the existing environment (call it `e*`) to the closure's creation-time environment, `e`.
- push a new frame—that is, transition into a local environment whose parent is `e`.
- to that environment, add bindings `x1 = v1, ... , xN = vN` where the x's are the variable names.
    - if there is a rest parameter `xR`, bind it to the list `(xK+1 ... xN)` where `K` is the number of mandatory arguments per the arglist.
- if the closure is named, also add a binding of its name to the function—that is, a reference to itself—in the environment.
    - this enables named functions to call themselves; that is, recursion.
- evaluate the function's body in the new environment.
- return the result, and restore the old environment `e*`.

Memory Usage

Note that closures can "capture" bindings that would otherwise be deallocated. Here's an interesting example:

```
psi> (def counter (let a (cell 0) (fn (b) (:= a (+ (! a) b)))))
$lambda{(b) (:= a (+ (! a) b))}
psi> (counter 5)
5
psi> (counter 7)
12
```

```
psi> (counter 11)
23
```

The closure has exclusive access to this reference cell—therefore, the cell that would otherwise be deleted as unreachable must be kept around. However, if the closure is ever deleted, the reference cell must also be deleted.

When counter was created, the environment created for the returned closure looked like this:

```
e1 = {a : $cell{0}@0x....beef5000} with parent e0
e0 = {+ : $builtin{+}; ...  } with no parent (global).
```

When counter is applied to 5, it evaluates the closure's body in an environment like so:

```
e2      = {b : 5} with parent e1_copy.
e1_copy = {a : $cell{0}@0x....beef5000} with parent e0_copy.
e0_copy = {+ : $builtin{+} } with no parent (global).
```

Recursion

Named functions can refer to themselves:

```
psi> (def factorial (fn this (x) (if (= x 0) 1 (* x (this (- x 1))))))
$lambda{(x) (if (= x 0) 1 (* x (this (- x 1))))}@this

psi> (factorial 5)
120
```

The case above is unusual—the interior name differs from the symbol it is bound to in the global environment—but legal. The inner name `this` is bound in the function's use-time environment, but never available outside.

Reference Counting

Warning: Although closures create full copies of environments, there are two kinds of objects you cannot copy:

- `cells`—a copy of a cell may diverge from the original; they cannot be considered the same.
- closures—as closures' copied environments may contain other closures, naive copying results in exponential memory usage.

Therefore, when a copy request is made for a pval of either type, you must pass the original pointer. That, on its own, isn't unsafe, although it defeats the purpose of copying—a

uniquely-owned copy of an object can always safely be deleted when the owner itself is deleted. Since `cell`s and closures are not copied—the original pointer is given out instead—there is no unique owner. The object can't be safely deleted unless *all* its owners have relinquished it (typically, by being deleted themselves.)

The easiest solution is for an uncopyable object to keep a reference count, which is initialized to 1 when the object is created. When a copy is requested, the count is incremented; when deletion is requested, it is decremented, but deletion only occurs if the count reaches zero. This is a primitive form of garbage collection and it is good enough for this course's project.

`__start.psi` (Optional)

Almost all PSI functionality will require native support from the implementation—and, because this is a C course, it must be written in C. However, you may include a `__start.psi` file for PSI code that will run on interpreter startup. For example, if you implement closures correctly, you won't need to implement the `list` function as a builtin, as you can include, in your `__start.psi`:

```
(def list (& xs) xs)
```

The function will `print` differently (i.e., as a closure, rather than `$builtin{list}`) but it will work identically, and this is acceptable.

Furthermore, if you implement macros, you do not need to build in the special forms `and`, `or`, and `let`. Macros are provided for each.

Macros (Optional)

Macros give the users a way to create their own special forms—if the special form is a code transformation that can be written as a PSI function, then a macro can be used.

For example, let's say that you'd like to add a `while/2` construct to PSI, to be able to write programs like the following:

```
;; while.psi

;; equivalent to the C program
;; int counter = 0;
;; int sum     = 0;
;; while (counter < 10) {
;;     counter += 1;
;;     sum     += counter;
;; }
;; printf("%d\n", sum);
```

```
(def counter (cell 0))  ; PSI values are immutable, must use cells
(def sum (cell 0))

(while (< (! counter) 10)
  (do (:= counter (+ 1 (! counter)))
      (:= sum (+ (! sum) (! counter)))))

(print (! sum))

;; ---- end of while.psi
```

We can't write this `while` as a regular function, because standard forms eagerly evaluate their subforms—if you supplied `(< (! counter 10))` as a function argument, it would be evaluated once; we want to pass this *as code* to our `while`-form.

Here's how we write that:

```
psi> (macro while (c body) (list 'loop (list 'if c (list 'do body #t))))
$macro{(c body) (list 'loop (list 'if c (list 'do body #t)))}@while
```

A macro is backed by a closure, but it works on evaluated subforms to transform code before evaluation. Consider the case below:

```
psi> (let a (cell 1) (while (< (! a) 10) (print (:= a (* 2 (! a))))))
2
4
8
16
#t
```

The operation of `while`, in fact, happens in two stages:

Stage 1: Macroexpansion—the closure is applied to the unevaluated subforms:

```
#0 -- symbol while recognized as macro
#1 = (< (! a) 10)
#2 = (print (:= a (* 2 (! a))))

mex = (list 'loop (list 'if #1 (list 'do #2 #t)))
    = (loop (if (< (! a) 10)
                (do (print (:= a (* 2 (! a)))) #t)))
```

Stage 2: Evaluation—the result of the macroexpansion is evaluated:

```
(loop (if (< (! a) 10)
          (do (print (:= a (* 2 (! a)))) #t))) -> <OUT: 2\n4\n8\n16\n> #t
```

The following macros implement `and`, `or`, and `let` for you—if you implement macros, you can include these in your `__start.psi` rather than coding them directly.

```
(macro and (& x)
  (if (= x ()) #t
    (if (= (tail x) ())
      (head x)
      (list 'if (head x)
                  (cons 'and (tail x))
                  #f))))

(macro let (& x)
  (if (= (% (length x) 2) 0)
    (error (list 'arity-error 'let 'odd (length x)))
    (if (= (tail x) ()) (head x)
      (if (= (type (head x)) 'symbol)
        (list (list 'fn (list (head x))
                (cons 'let (tail (tail x))))
                (head (tail x)))
        (error (list 'type-error 'let 1 'symbol (head x)))))))

(macro or (& x)
  (if (= x ()) #f
    (let y (new-symbol)
      (list
        (list 'fn (list y)
          (list 'if y y (cons 'or (tail x))))
        (head x)))))
```

Glossary

PSI exposes 33 functions and 11 special forms—most, you will implement directly in C; a couple of these can be implemented in your `__start.psi`. So long as there is no degradation of functionality, you won't be penalized for doing so.

All builtin functions are capable of throwing `arity-error` and `type-error` if given arguments outside of their constraints—therefore, those are not mentioned here as possible error types.

+ (or "plus")

arity: 0+
input types: number
return type: number

Returns the sum of its arguments.

```
(+)           –> 0
(+ 2 3)       –> 5
(+ 5 7 11 13) –> 36
```

`-` (or “minus”)
arity: 1+
input types: number
return type: number

Arity 1: Returns the negation of its first argument.
Arity 2+: Subtracts arguments #2, #3, ...; all from #1.

```
(- 2 3) –> -1
```

`*` (or “times”)
arity: 0+
input types: number
return type: number

Returns the product of its arguments; `(*)` is 1.

```
(* 2 3) –> 6
```

`/` (or “div”)
arity: 1+
input types: number
return type: number
can throw: `division-by-zero` if denominator is 0.

Integer division; start with argument #1, then divide by #2, then #3, and so on.

In PSI, integer division always rounds fractional results down (i.e., to negative infinity.)

```
(/  7  3) –>  2
(/ -2  3) –> -1
(/ -7 -3) –>  2
(/  2 -3) –> -1
```

% (or “mod”)
arity: 2
input types: number
return type: number

Returns the remainder when #1 is divided by #2. Always returns the positive modulus.

```
(% -1 13) –> 12
```

=
arity: 0+
input types: any
return type: bool

Returns `#t` if #1, #2, ... are all equal; `#f` otherwise. Always `#t` at arity 0, 1.

```
(= 3 3) –> #t
```

!=
arity: 0+
input types: any
return type: bool

Identical to `(not (= #1 ...))`. Returns `#f` if #1, #2, ... are all equal; `#t` otherwise. Always `#f` at arity 0, 1.

```
(!= 3 3) –> #f
```

<, **<=**, **>**, **>=** (as $cmp)
arity: 0+
input types: number
return type: bool

Returns `#t` if the comparisons {#1 $cmp #2, #2 $cmp #3, ...} all hold. Otherwise, `#f`. Always `#t` at arity 0, 1.

```
(< 3 3)   –> #f
(<= 3 3)  –> #t
(< 3 4 5) –> #t
(< 3 4 4) –> #f
```

!
arity: 1
input type: cell

return type: any

Returns the current contents of a mutable reference cell.

```
;; (def c (cell 5))
(! c) -> 5
```

:=
arity: 2
input types: #1 cell, #2 any
return type: any

Writes a new value into a reference cell, and returns the written value.

```
(:= c 6) -> 6        ;; (! c) will henceforth return 6
```

and
special form
arity: 0+
input type: any
return type: any

Evaluates subforms, from left to right, until one returns `#f`—in this case, returns `#f`. If all are true, returns the last value; always returns `#t` on arity 0.

Must be implemented as a special form or macro due to short-circuiting.

```
(and)            -> #t
(and 1 2)        -> 2
(and #f (/ 1 0)) -> #f
```

cell
arity: 1
input type: any
return type: cell

Returns a mutable reference cell that can be read using `!` and written-to using `:=`.

```
(cell 5) -> $cell{5}@0x6000....
```

chr
arity: 1
input type: list—all elements numbers between 0 and 255, inclusive.
return type: string

Returns a string whose contents, byte by byte, correspond to the given integer list.
`(chr '(99 97 116)) –> "cat"`

`cons`
arity: 2
input types: #1 any, #2 list
return type: list

“Cons”-tructs a list whose first element (“head”) is #1 and whose tail is #2.

```
(cons 3 '(4 5)) –> (3 4 5)
```

`def`
special form
Arity: 2 or 3—`(def <name> <arglist> <form>)`
input types: {`name` : symbol, `arglist` : arglist, `form` : any}
return type: any
can throw: `protected-symbol` if an attempt to redefine a protected symbol (builtin name) occurs.

`def/2`: evaluates `form –> v`. If an error occurs, throws it. Otherwise, creates a binding, in the global environment, of `name` to `v`. Returns `v`.

`def/3`: `(def name arglist form)` is identical to `(def name (fn name arglist form))`.

`do`
special form
arity: 0+
input types: any
return type: any

Evaluates the subforms in order from left-to-right until either an error is thrown—in this case, throw it—or all have been evaluated. If all have been successfully evaluated, returns the result of the last form. In the arity-0 case, returns `#t`.

```
(do 1 2 (+ 5 6))                    –> 11
(do (print 1) (print 2) (print 3)) –> <OUT: 1\n2\n3\n> #t
(do (print 1) (/ 1 0) (print 3))   –> <OUT: 1\n> $error{division-by-zero}
```

`error`
arity: 1
input type: any
return type: error

Returns an error object wrapping the given pval. Funny enough, also creates an error object when its arity specification is violated—you can't truly lose with this one.

```
(error 'no-sir-i-don-t-like-it) —> $error{no-sir-he-doesn-t-like-it}
```

`fn`
special form
arity: 2 or 3—(`fn [<name>] <arglist> <body>`)
input types: {`name`: symbol, `arglist`: arglist, `body`: any}
return type: function
can throw: `arglist-error` if arglist is not a list, has non-symbol elements, has duplicate symbols, or has & in a position other than second-to-last.

Captures (deep-copies) the creation-time environment and returns a closure object with the name, arglist, body, and environment as specified.

```
(fn () 0)                —> $lambda{() 0}
((fn () 0))              —> 0
((fn (x) x) 7)           —> 7
((fn (x & xs) xs) 3 4 5) —> (4 5)
((fn f (x)
   (if (= x 0) 0
     (+ x (f (- x 1)))))
  5)                     —> 15
```

`get-file`
arity: 1
input type: string
return type: string
can throw: `(bad-filename <string>)`

Returns contents of the file with filename #1 as a string.

```
(get-file "very-boring.r34") —> <File IO> "Nothing. Why, what'd you expect?"
```

`head`
arity: 1
input type: list
return type: error
can-throw: `value-error` if #1 is empty.

Returns the initial element of a nonempty list. Throws `value-error` if the list is empty.

```
(head '(1 2 3)) —> 1
```

`if`
special form
arity: 2 or 3—`(if <condition> <then-form> [<else-form>])`
input type: any
return type: any

Always evaluates the `condition` form; if it generates an error, throws it. If the value is `#f`, evaluates and returns `else-form`; otherwise, evaluate and return `then-form`. In the arity-2 case, behaves as if `else-form` were given as `#f`.

```
(if #f 1)         –> #f
(if #t 1 2)       –> 1
((if #f + *) 3 4) –> 12
```

`input`
arity: 0
return type: string

Prompts the user for a line of input—returns it, with the terminal newline removed.

```
(input) –> <IN: qxz\n> "qxz"
```

`let`
special form
arity: odd—`(let name-1 expr-1 ... name-N expr-N body)`
input types: all names must be symbols
return type: any

If `N` = 0, evaluate `body –> v` and return v.

Otherwise: from left to right, evaluate `expr-N –> vN` and, on success, bind `vN` to `name-N` in a local environment. (On an error, throw the first error.) These expressions may have backward dependencies; for example, `expr-2` must be evaluated in an environment where the binding `name-1 : v1` is available. If the same symbol is bound twice, the later binding shadows the earlier one. Once the bindings are all available, evaluate `body` in that local environment, return the result, and transfer back to the original environment.

```
(let a 5 (* a a)) –> 25
(let a 3
     a (* a a)
     a (* a a)
     a)           –> 81
```

Note—The following are legal transformations:

```
(let body)                      <->   body
(let n1 e1 n2 e2 ... body)      <->   (let n1 e1 (let n2 e2 ... body))
(let n1 e1 body)                <->   ((fn (n1) body) e1)
```

These transformations mean that you can implement general `let` in terms of `let/3`, and that you can skip it altogether if you implement macros. Implementing `let` this way may alter some of your error messages, but you are not graded on error messages in this course—only that errors occur when they should and are of the correct type.

`list`
arity: 0+
input type: any
return type: list

Creates and returns a list of the arguments passed in.

```
(list)         -> ()
(list 1 #t "") -> (1 #t "")
```

`load`
arity: 1
input type: string
return type: number

Evaluates the contents of file #1 as S-expressions. Returns an error if any have been thrown—otherwise, returns the number of S-expressions evaluated.

```
(load "r34.psi") -> 0     ;; it turns out there isn't
```

`loop`
special form
arity: 1
input type: any
return type: `#t`

Evaluates subform over and over until it throws an error—and then throws it—or returns `#f`; on successful exit, returns `#t`.

```
(loop #f) -> #t
(loop #t) -> <never returns--infinite loop>
; (def c (cell 0))
(loop (if (< (! c) 4)
```

```
(print (:= c (+ (! c) 1))))) -> <OUT: 1\n2\n3\n4\n> #t
```

macro
special form
arity: 3—`(macro name arglist body)`
input type: {`name`: symbol, `arglist`: arglist, `body`: any}
return type: function

Creates a closure for a macro with the provided name, arglist, and body. This will operate like a special form. When the macro is invoked, it will be run on the subforms—not the evaluated arguments—to produce translated code, which will then be evaluated.

not
arity: 1
input type: any
return type: bool

Returns `#t` if #1 is `#f`, and `#f` otherwise.

```
(not 0) -> #f
```

new-symbol
arity: 0
return type: symbol

Generates a guaranteed unique symbol that will be used nowhere else—necessary for hygienic macros. The `$` character is a “reader-reject” character—the reader will never generate a symbol using it, but this function can create such symbols.

```
(new-symbol) -> g$137
(new-symbol) -> g$138
```

or
special form
arity: 0+
input type: any
return type: any

Evaluates subforms, from left to right, until one returns a true (non-`#f`) value—then returns it—or throws an error—then throws it. If all are false, or arity is 0, returns `#f`.

```
(or)             -> #f
(or #f 2)        -> 2
(or 13 (/ 1 0))  -> 13
```

ord
arity: 1
input type: string
return type: list
can throw: `value-error`, if list elements not numbers between 0 and 255, inclusive.

Returns the ASCII codes of the characters, one by one, in a string. Does not add a null terminator.

```
(ord "CAT") -> (67 65 84).
```

output
arity: 1
input type: string
return type: `#t`

Prints a string verbatim to the console.

```
(output "CAT") -> <OUT: CAT> #t
```

parse
arity: 1
input type: string
return type: `#f` or `string` or 2-element list
can-throw: `invalid-token` on unparseable token; and `incomplete-parse` if the string is a possible prefix of a valid pval, but not completed—for example, `"(+ 1 2"`.

Returns `#f` when given an empty string. Otherwise, parses either an inactive (i.e., whitespace or comment) token—then returns the remainder of the string—or a single pval and, if successful, returns a 2-element list containing the parsed pval and the remainder of the string.

```
(parse "")       -> #f
(parse "     1") -> "1"


(parse "(def f (x) (* x x)) (def g (y z) (+ y (f z)))") ->
  ((def f (x) (* x x)) " (def g (y z) (+ y (f z)))")
```

print
arity: 0+
input type: any
return type: `#t`

Prints `(str #1)` ; `" "` ; `(str #2)` ; ... ; `(str #N)` ; `"\n"` to the console.

```
(print 1 #f "CAT") –> <OUT: 1 #f "CAT"\n> #t
```

put-file
arity: 2
input type: string
return type: `#t`

Opens the file named #1—does not append, creates the file if it does not exist—and writes the contents of #2, then closes it.

```
(put-file "i-swear-it-s-nothing.txt" "Chapter 46 ...") –> <File IO> #t
```

quit
arity: 0
return type: (none)

Exits the interpreter with return code 0.

```
(quit) –> <EXIT 0>    ;; and James Joyce finishes "that sentence."
```

quote
special form
arity: 1
input type: any
return type: any

Returns the *unevaluated* form—during evaluation, one layer of quoting is removed.

```
(quote x)                 –>   x
(quote (quote x))         –>  'x
(quote (quote (quote x))) –> ''x
;; for contrast
x                         –> <look up x, or throw error if unbound>
```

read
arity: 1
input type: string
return type: any
can throw: `invalid-token`, `incomplete-parse` from parsing; also `value-error`.

Reads (parses) a string for exactly one pval, and returns it. Throws `value-error` if there are no pvals, or if there are non-whitespace characters remaining after the parsing.

```
(read "(+ 1 2)") -> (+ 1 2)

(read "")    -> $error{(value-error read "")}
(read "1 2") -> $error{(value-error read "1 2")}
(read "$")   -> $error{(invalid-token "$")}
```

str
arity: 0+
input type: any
return type: string

Returns a PSI string containing the canonical, readable representation of readable PSI values. Should return some useful string for non-readable types.

```
(str 3)     -> "3"
(str #t)    -> "#t"
(str "cat") -> "\"cat\""
```

tail
arity: 1
input type: list
return type: symbol
can-throw: `value-error` if #1 is empty.

Returns the tail—everything but the initial element—of the list.

```
(tail '(1 2 3) -> (2 3))
```

try
special form
arity: 1
input type: any
return type: 2-element list (#0 bool, #1 any)

Evaluates its subform. If it successfully returns a value v, returns `(#t v)`; on an error, returns `(#f e)` where `e` is the pval stored inside the error object.

```
(try (/ 15 5)) -> (#t 3)
(try (/  1 0)) -> (#f division-by-zero)
```

type
arity: 1
input type: any

return type: symbol (`bool`, `number`, `string`, `symbol`, `list`, `cell`, `function`)

Returns the type of the given argument.

```
(type 5)      –> number
(type "this") –> string
(type type)   –> function
```