

PSI Phase 2

For Phase 2, you will add some features to your Phase 1 “calculator” to give it: memory, list processing, conditional execution, strings, and basic I/O.

Comparison Functions

To start, add the comparison functions `<`, `<=`, `>=`, `>`, and `!=`. These, as well as the existing math functions (`+`, `-`, `*`, `/`, `=`) should be extended to all the arities supported in the PSI spec:

```
psi> (+)
0
psi> (+ 5 6 7)
18
psi> (= 3 3 4)
#f
psi> (< 3 4 5)
#t
```

You will also add a `not` function (arity 1) that returns `#t` if the argument is `#f`, and `#f` otherwise.

```
psi> (not #t)
#f
psi> (not 1)
#f
psi> (not #f)
#t
```

In other words, everything that isn’t `#f` is “truthy” in PSI.

The ordered comparisons are strictly numerical and throw type errors if given non-number arguments, but `=` and `!=` have cross-type semantics—equality across types is always false, but it is not a type error.

```
psi> (= 0 ())
#f
psi> (!= 0 ())
#t
```

Creating Bindings Using def

Code is evaluated in an *environment* that contains *bindings*. In Phase 2, you only have to worry about one global environment. New bindings are introduced into it using the `def` special form:

```
psi> (def a 7)
7
psi> a
7
psi> (+ a 4)
11
```

This def is **not** a function; we call it a *special form*, and we'll see others. The interpreter knows, when the head of a program is def, not to call a function but to do something different. Notably, a has not been bound yet when the def is used—the name is taken as-is, not evaluated (evaluating a symbol looks up its binding, but it has not been bound yet.)

You can rebind a symbol as you wish, but you can't rebind the builtin function's names. It is also an error to evaluate a symbol that has no bindings.

```
psi> (def a (+ a 6))
13
psi> (+ a 4)
17
psi> (def + 5)
$error{(protected-symbol +)}
psi> b
$error{(unbound b)}
```

The first line above shows us that old bindings can participate in the #2 subform of a def; the (unevaluated) symbol at #1 is used for the new name—in this case, the old binding (to 7) is deleted—and you are responsible for freeing unreachable memory.

Also, the `_` symbol is always bound to the last non-error return value:

```
psi> (+ 3 4)
7
psi> (/ 1 0)
$error{division-by-zero}
psi> (+ _ 1)
8
```

Conditional Execution

The `if` special form introduces conditional execution. This could not be implemented as a function, since functions evaluate all arguments (even if they do not use them.) It works as follows: The #1 subform is always evaluated. If the result is anything other than #f, the #2 subform is evaluated; if it's #f, the #3 subform is evaluated. If there is no #3 subform, the default value is #f.

```
psi> (if (= 3 3) 1 2)
1
psi> (if (= 3 4) 1 2)
2
psi> (if (= 3 4) 1)
#f
psi> ((if (= 3 4) + *) 7 8)
56
```

An example of why this is useful is below:

```
psi> (if (= 3 3) 6 (/ 1 0))
6
```

If we had implemented `if` as a function, this would return an error—however, since `(= 3 3)` returns `#t`, the `(/ 1 0)` is never executed.

Lists—head, tail, and cons

You've already used lists to represent code, but now you will develop the ability to build lists and take them apart using:

- `head`, which returns the initial element of a list.
- `tail`, which returns everything but the initial element.
- `cons`, which creates a new list whose head is the first element and whose tail is the second.

```
psi> (def l (cons 1 (cons 2 ())))
(1 2)
psi> (head l)
1
psi> (tail l)
(2)
psi> (cons 3 (tail (tail l)))
(3)
```

The `head` and `tail` functions throw `value-error` if given empty lists and type errors if given nonlists; `cons` throws a type error if the #2 argument is not a list.

```
psi> (head 1)
$error{(type-error head 1 list 1)}
psi> (head ())
$error{(value-error head ())}
```

```
psi> (tail ())
$error{(value-error tail ())}
psi> (cons 1 31)
$error{(type-error cons 2 List 31)}
```

If x is a nonempty list, it will always be the case that $(\text{cons} (\text{head } x) (\text{tail } x))$ equals x . Lists can hold PSI values of all types except the error type—they are heterogeneous.

Quotation

There is a special form called quote and there is a shorthand using the single-quote character; you'll support both. The effect of quote is to tell the evaluator *not* to look up a symbol or treat a list as a function application:

```
psi> (def x 17)
17
psi> x
17
psi> (quote x)
x
psi> 'x
x
psi> (quote (quote x))
'x
psi> (quote (+ 3 4))
(+ 3 4)
psi> '(+ 3 4)
(+ 3 4)
psi> (quote (quote (+ 3 4)))
'(+ 3 4)
```

With this covered, note that *computing* a list (or symbol) is different from *evaluating* it. Below are three ways to compute the list $(1 \ 2 \ 3)$; however, evaluating it would cause an error because 1 is not applicable—that is, not a function.

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

Strings

You will also support a `string` type. Please see the Spec for full detail; however, for Phase 2, you won't be tested on the unusual cases—just get the basics working. Strings in PSI are *not* null-terminated; they are sequences of 8-bit numbers that represent the characters in the ASCII encoding. You will implement `ord`, which converts a string into a list of numbers, and `chr`, which converts a list of numbers into a string.

```
psi> "cat"  
"cat"  
psi> (ord "cat")  
(99 97 116)  
psi> (chr '(99 97 116))  
"cat"  
psi> (chr '(65 10 66))  
"A\nB"
```

I/O and the type function

You will implement the functions `input` and `output`, which have “side effects,” meaning they do things other than compute values. The `input` function (arity 0) takes a line of input from the user and returns it, without the newline, as a string:

```
psi> (input)  
Console input here.  
"Console input here."
```

The `output` function has arity 1 and only takes strings. It prints the given string to the console and then returns `#t`. Below, the printing action and return both occur in standard output, but this will not always be the case—in Phase 3, there will be cases in which we don't print return values. Still, `output` *always* prints to stdout.

```
psi> (output "Hello\n\nworLd!\n")  
Hello  
  
world!  
#t
```

Last of all, the `type` function returns, as a symbol, the type of its single argument.

```
psi> (type #t)  
bool
```

```
psi> (type 1)
number
psi> (type "1")
string
psi> (type 'x)
symbol
psi> (type '(1))
list
psi> (type +)
function
```