How To Talk To Computers

by a11ce

```
add 1 1
add 1 multiply 2 3
add 1 multiply 2 3 4
add 1 (multiply 2 3) 4
add 1 1.5 (multiply 2 3 4)
add 1 1.5
multiply 2 3 4
```

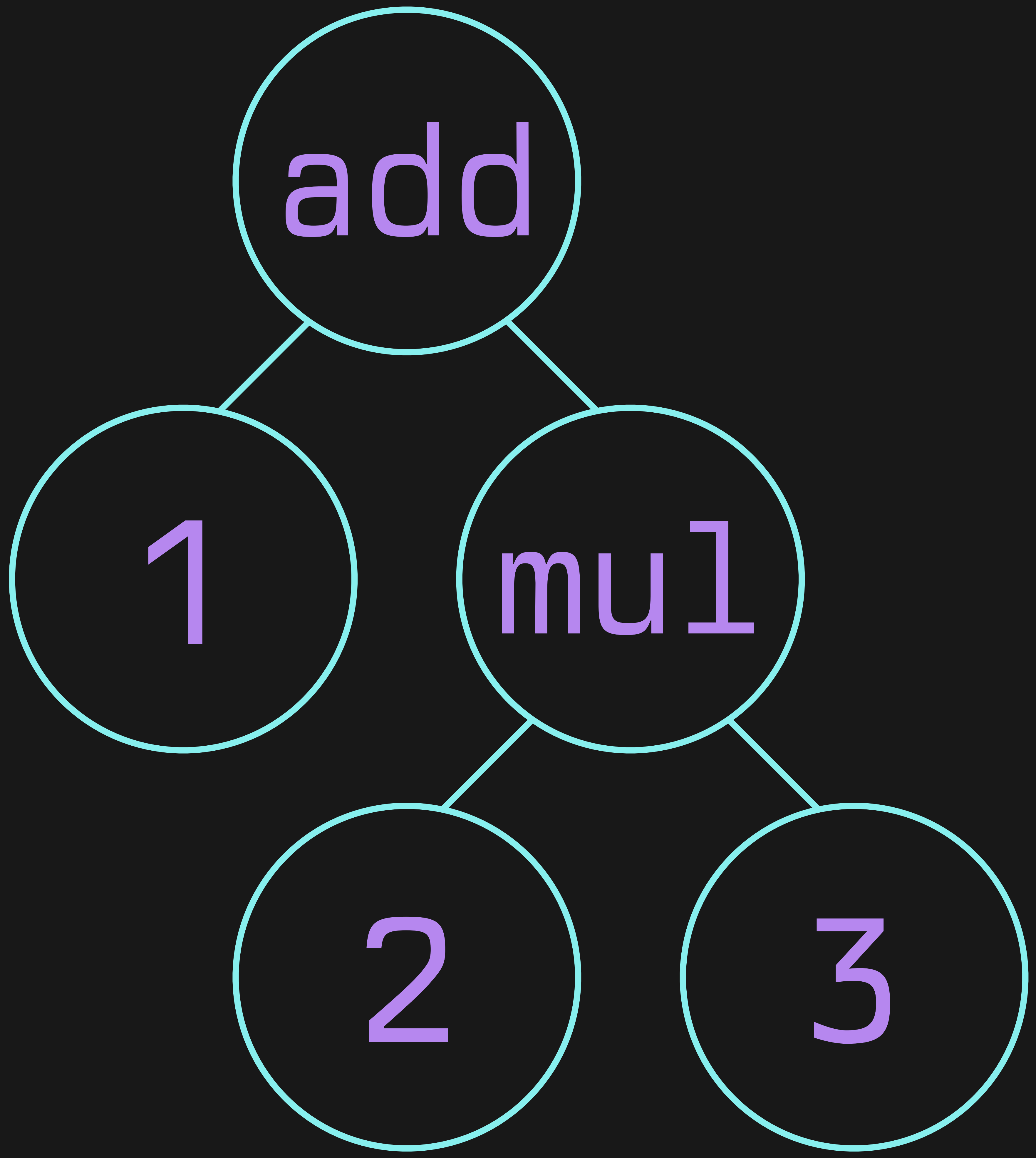
```
(add 1 1.5)
(multiply 2 3 4)
```

```
(add 1 1.5) (multiply 2 3 4)
```

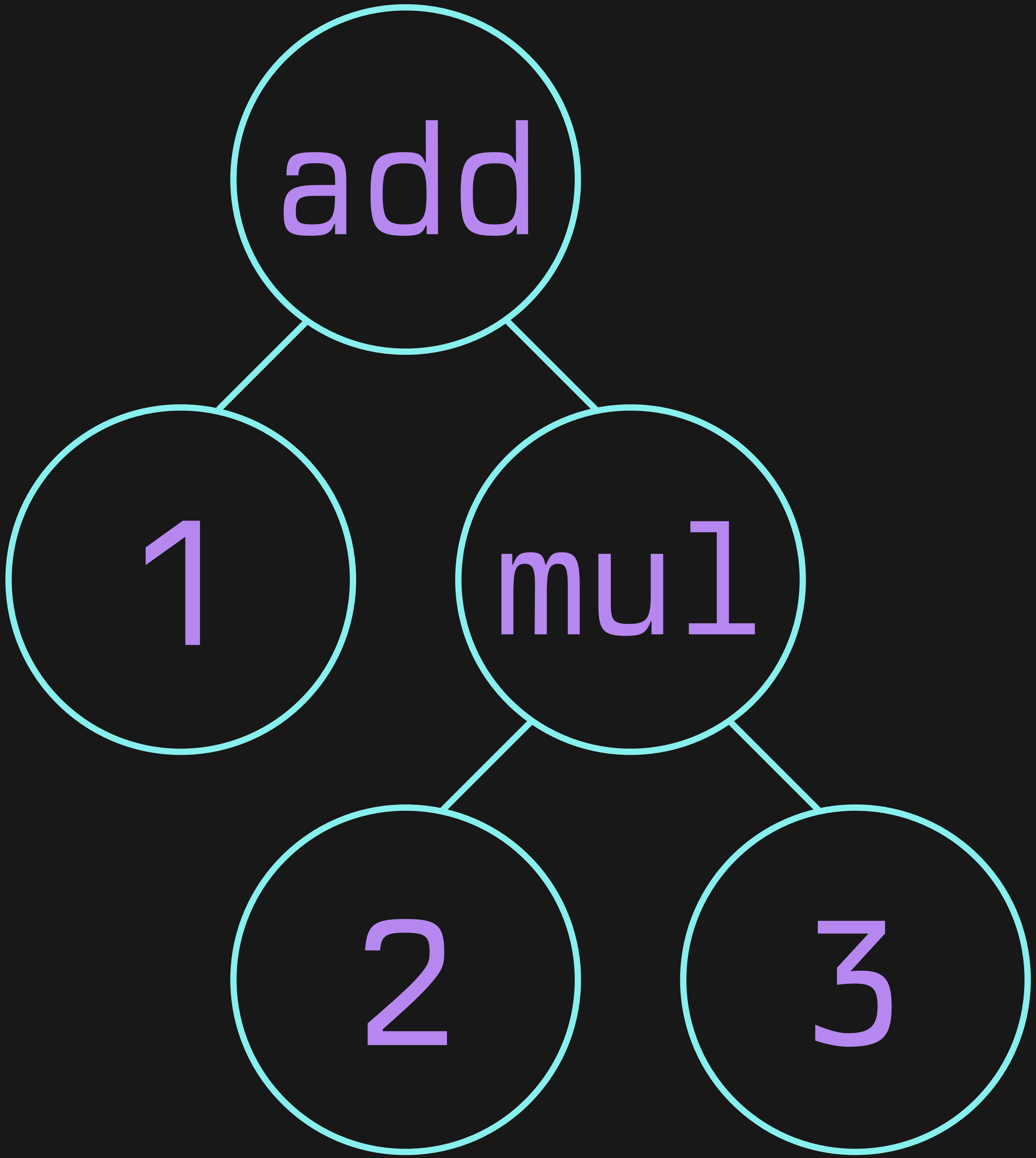
```
add 1 multiply 2 3
```

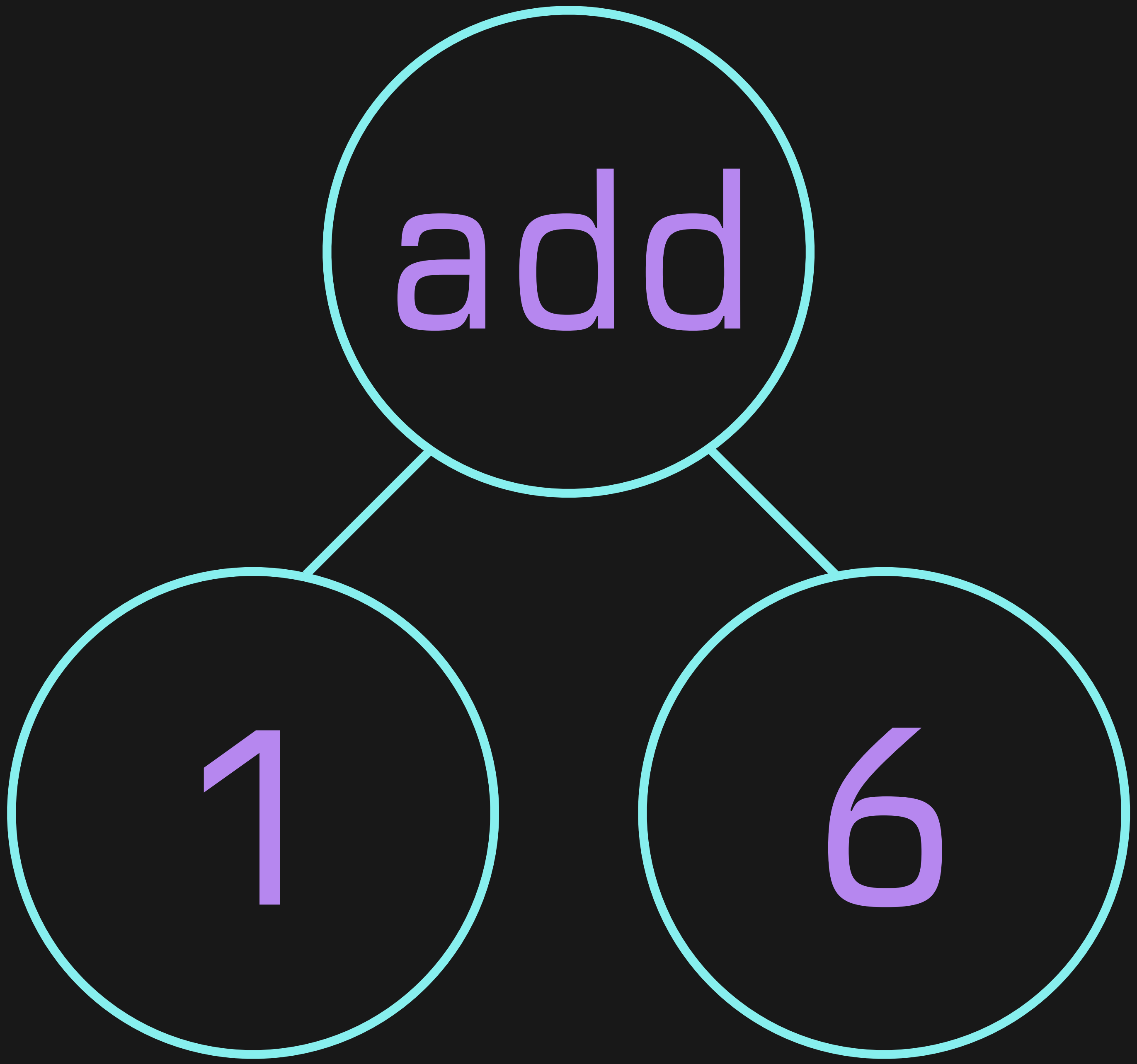
```
(add 1 (multiply 2 3))
```

```
(add 1 (multiply 2 3))
```

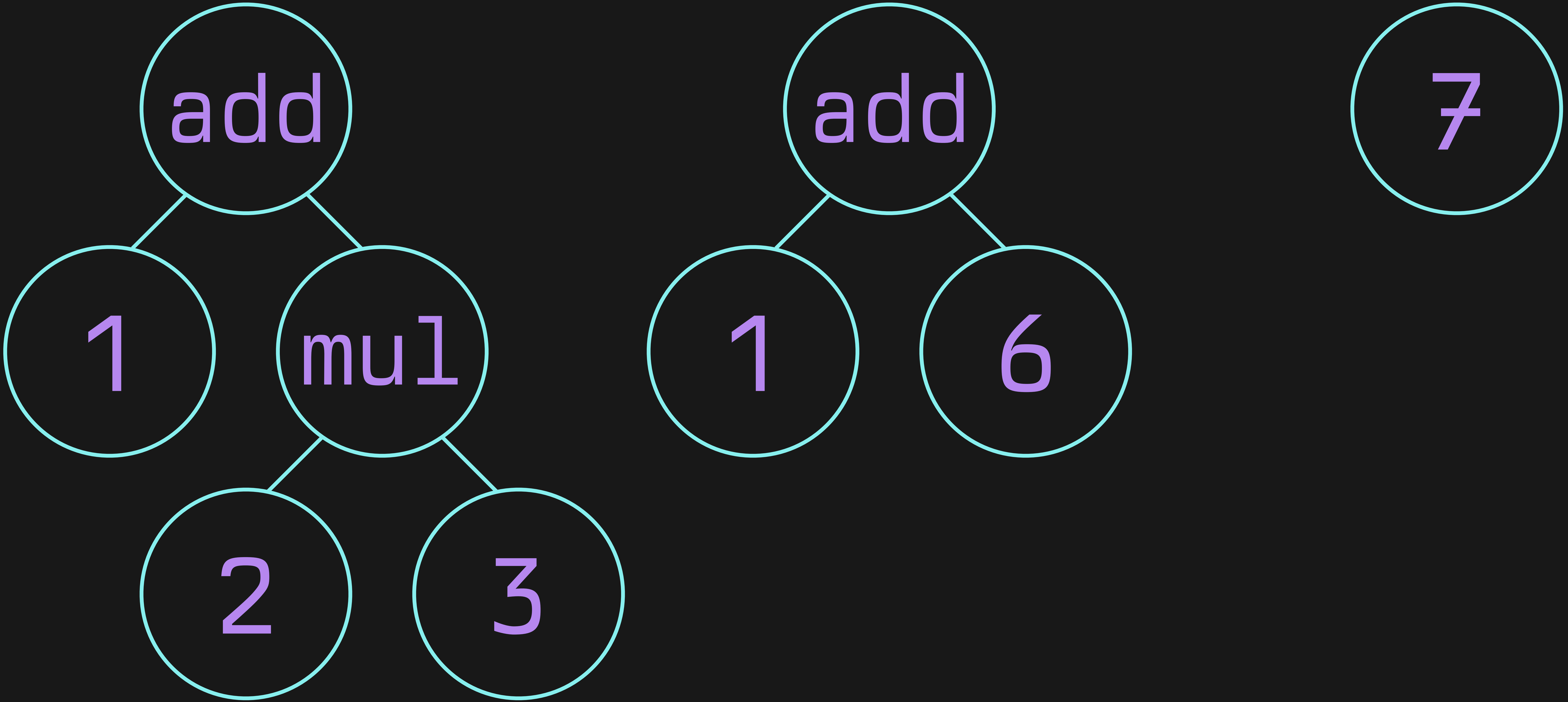


```
(add 1 (multiply 2 3))
```





```
(add 1 (multiply 2 3))
```



```
"(add 1 (multiply 2 3))"
```

```
"(add 1 (multiply 2 3))"
```

split on parens and whitespace,
trim whitespace

```
["(", "add", "1", "(",
"multiply", "2", "3", ")", ")"]
```

```
"(add 1 (multiply 2 3))"
```

split on parens and whitespace,
trim whitespace

```
["(", "add", "1", "(",
"multiply", "2", "3", ")", ")"]
```

start a list at "(", end and add
the list to the parent at ")"

```
["add", "1",
["multiply", "2", "3"]]
```

```
(add 1 (multiply 2 3))
```

```
(add 1 (multiply 2 3))
```

as a string

```
"(add 1 (multiply 2 3))"
```

```
(add 1 (multiply 2 3))
```

as a string

```
"(add 1 (multiply 2 3))"
```

parsed to

```
["add", "1", ["multiply", "2", "3"]]
```

```
(add 1 (multiply 2 3))
```

as a string

```
"(add 1 (multiply 2 3))"
```

parsed to

```
["add", "1", ["multiply", "2", "3"]]
```

written as

```
['add '1 ['multiply '2 '3]]
```

'2

```
'2 -> 2
```

```
  '2 -> 2
'add -> add
```

```
'2 -> 2
'add -> add
```

1. Numbers evaluate to themselves.

```
'2 -> 2
'add -> add
```

1. Numbers evaluate to themselves.
2. Other non-lists evaluate to ??

```
'2 -> 2
'add -> add
```

1. Numbers evaluate to themselves.
2. Other non-lists evaluate to ??

```
{'add: function that adds numbers}
```

```
'2 -> 2
'add -> add
```

1. Numbers evaluate to themselves.
2. Other non-lists evaluate to whatever the environment says the name means.

```
{'add: function that adds numbers}
```

```
['multiply '2 '3]
```

```
['multiply '2 '3]
```

3. For a list, eval each of the elements...

```
[multiply 2 3]
```

```
['multiply '2 '3]
```

3. For a list, eval each of the elements...

```
[multiply 2 3]
```

Then call the first element (a function) with the other elements as arguments

```
6
```

1. Numbers evaluate to themselves.
2. Other non-lists evaluate to whatever the environment says the name means.
3. For a list, eval each of the elements then call the first element (a function) with the other elements

```
['add '1 ['multiply '2 '3]]
```

1. Numbers evaluate to themselves.
2. Other non-lists evaluate to whatever the environment says the name means.
3. For a list, eval each of the elements then call the first element (a function) with the other elements

```
['add '1 ['multiply '2 '3]]
 [add 1 ['multiply '2 '3]]
```

1. Numbers evaluate to themselves.
2. Other non-lists evaluate to whatever the environment says the name means.
3. For a list, eval each of the elements then call the first element (a function) with the other elements

```
['add '1 ['multiply '2 '3]]
 [add 1 ['multiply '2 '3]]
   [add 1 [multiply 2 3]]
```

1. Numbers evaluate to themselves.
2. Other non-lists evaluate to whatever the environment says the name means.
3. For a list, eval each of the elements then call the first element (a function) with the other elements

```
['add '1 ['multiply '2 '3]]
 [add 1 ['multiply '2 '3]]
   [add 1 [multiply 2 3]]

          [add 1 6]
```

1. Numbers evaluate to themselves.
2. Other non-lists evaluate to whatever the environment says the name means.
3. For a list, eval each of the elements then call the first element (a function) with the other elements

```
['add '1 ['multiply '2 '3]]
[add 1 ['multiply '2 '3]]
[add 1 [multiply 2 3]]
[add 1 6]
7
```

```
(if (whatever)
 (print "yes")
 (print "no"))
```

```
{'add: function that adds numbers
 'print: print}
```

```
(if (whatever)
 (print "yes")
 (print "no"))
```

```
{'add: function that adds numbers
 default: the interpreter's def!}
```

1. Numbers (and strings and stuff) evaluate to themselves.
2. Other non-lists evaluate to whatever the environment says the name means.
3. For a list, eval each of the elements then call the first element (a function) with the other elements

```
(if (whatever)             "yes"
 (print "yes")              "no"
 (print "no"))     <yes or no again>
```

```
{'add: function that adds numbers
 default: the interpreter's def!}
```

1. Numbers (and strings and stuff) evaluate to themselves.
2. Other non-lists evaluate to whatever the environment says the name means.
3. For a list,
   1. If the first element evaluates to a special form, do the special form thing
   2. Otherwise, eval each of the elements then call the first element (a function) with the other elements

```
(if (whatever)
 (print "yes")
 (print "no"))
```

```
{'if: special: checks the condition
  THEN evaluates only one branch}
```

1. Numbers (and strings and stuff) evaluate to themselves.
2. Other non-lists evaluate to whatever the environment says the name means.
3. For a list,
   1. If the first element evaluates to a special form, do the special form thing
   2. Otherwise, eval each of the elements then call the first element (a function) with the other elements

```
(define x (add 1 2))
(print x)
```

{'define: special: evaluates the second arg, adds it to the env named as the first arg}

1. Numbers (and strings and stuff) evaluate to themselves.
2. Other non-lists evaluate to whatever the environment says the name means.
3. For a list,
   1. If the first element evaluates to a special form, do the special form thing
   2. If the first element evaluates to a lambda object, do the lambda thing
   3. Otherwise, eval each of the elements then call the first element (a function) with the other elements

```
(define y (lambda (x)
                (add 1 x)))
(print (y 2))
```

{'lambda: special: evaluates to an object that stores the lambda body and the parameter names. when called as a function, creates a temporary environment with the parameter names resolving to the values that it was applied to, evaluates the body with that environment}

1. Values (numbers/strings/etc) evaluate to themselves.
2. Names (other non-lists) evaluate to whatever the environment says the name means.
3. For a list,
   1. If the first element evaluates to a special form, do the special form thing
   2. If the first element evaluates to a lambda object, do the lambda thing
   3. Otherwise, eval each of the elements then call the first element (a function) with the other elements

1. Values (numbers/strings/etc) evaluate to themselves.
2. Names (other non-lists) evaluate to whatever the environment says the name means.
3. For a list, (this part is usually called apply)
   1. If the first element evaluates to a special form, do the special form thing
   2. If the first element evaluates to a lambda object, do the lambda thing
   3. Otherwise, eval each of the elements then call the first element (a function) with the other elements

THIS IS LISP!!!!!!