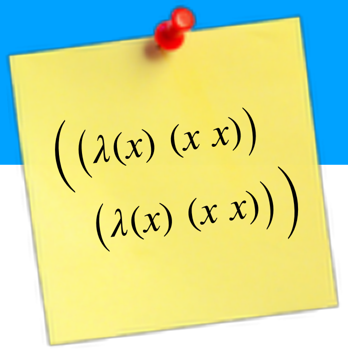
Racket Basics

CIS 352 — Spring 2020
Syracuse University

Racket

- **Dynamically typed**: variables are untyped, values typed
- **Functional**: Racket emphasizes functional style
  - Compositional—emphasizes black-box components
  - Immutability—requires automatic memory management
- **Imperative**: Racket allows data to be modified, in carefully considered cases, but doesn't emphasize "impure" code
- **Object-oriented**: racket has a powerful object system
- **Language-oriented**: Racket is really a language toolkit
- **Homoiconic:** Code is data; the primary data structure of Scheme, and LISP-family languages, is the *linked list*, written as **s-expressions**, & Scheme code is explicitly written as lists.

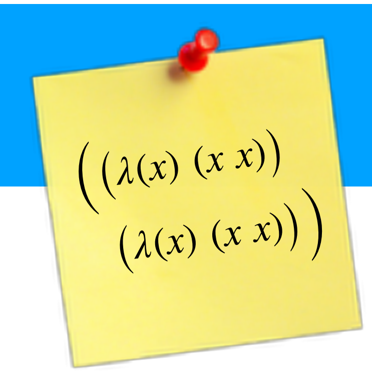
Example



Calculating the slope of a line in Racket

```
(define (calculuate-slope x0 y0 x1 y1)
  (/ (- y1 y0) (- x1 x0)))
```

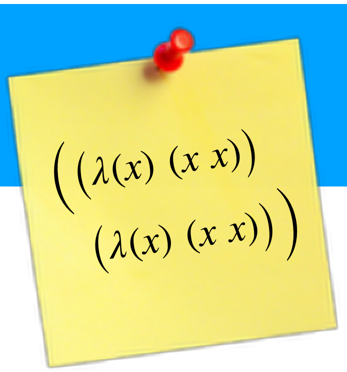
Example




```
(define (calculuate-slope x0 y0 x1 y1)
  (/ (- y1 y0) (- x1 x0)))
```

Prefix notation

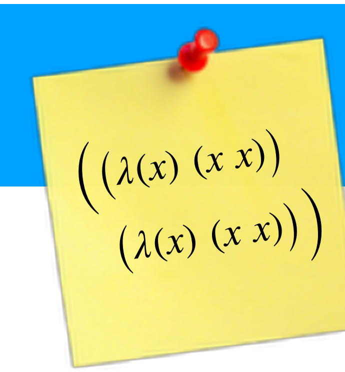
Example



Functions defined via prefix notation, too

```
(define (calculuate-slope x0 y0 x1 y1)
  (/ (- y1 y0) (- x1 x0)))
```

Example




```
(define (calculuate-slope x0 y0 x1 y1)
  (/ (- y1 y0) (- x1 x0)))

(calculate-slope 0 0 3 2)
```

Calls to user-defined functions also in prefix notation

Example

```
(define (calculuate-slope x0 y0 x1 y1)
  (/ (- y1 y0) (- x1 x0)))

(calculate-slope 0 0 3 2)
```

Note: preferred style puts closing parens at end of blocks

Basic Types

- **Numeric tower**. Numeric types gracefully degrade
  - E.g., `(* (/ 8 3) 2+1i)` is `16/3+8/3i`
  - Note that `2+1i` is a **literal** value, as is `2.3`
- **Strings** and **characters** (“`foo`” and `#\a`)
- **Booleans** (`#t` and `#f`) including logical operator (e.g., `or`)
  - Note that operators “short circuit”
- **Symbols** are interned strings `‘foo`
  - Implicitly only one copy of each, unlike (say) strings
- The `#<void>` value (produced by `(void)`)

Exercise

Compute the sum of the following:

- 2/3 and 1.5
- 3+8i and 3i
- 0 and positive infinity (`+inf.0`)

Exercise

Compute the sum of the following:

- **`(+ 2/3 1.5)`**

  `2.1666666666666665` (N.B., result is **inexact)**

- **`(+ 3+8i 3i)`**

  `3+11i`

- **`(+ 0 +inf.0)`**

  `+inf.0`

Forms

- A **form** is a recognized syntax in the language
  - `(if …)`, `(and …)` are forms, but +, `list` refer to functions
  - You can define new forms too! More on this later…
- Scheme prefers to give a small number of general forms.
- The tag just after the open-paren determines the form:
  - (define foo value) — Define a variable
  - (define (foo a0 a1 …) body) — Define a function
  - (if guard e-true e-false), (or e0 e1 …), etc
- Otherwise, by default, each pair of parens is a ***call site.***

Exercise

Define a function that takes an argument, x, and returns:

- x times 2, if x is less than 0
- x times -2 otherwise

Hint: use `(< x y)` for comparison

Exercise

```
(define (f x)
  (if (< x 0)
      (* 2 x)
      (* -2 x)))
```

Exercise

Define a function that takes an argument, x, and returns:

- x divided by 2, if x is even
- x times 3 plus 1, if x is odd

Hint: use `=` and `modulo` to check if x is even/odd

Exercise

```
(define (collatz x)
  (if (= 0 (modulo x 2))
      (/ x 2)
      (+ 1 (* 3 x))))
```

Derived Types

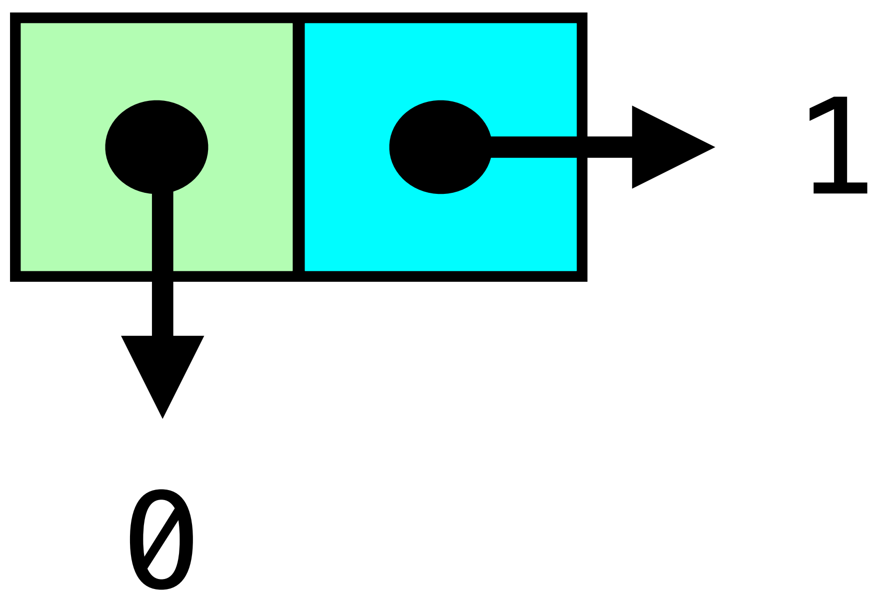
- **S-expressions** (**symbolic** expression)
  - Untyped lists that generalize neatly to trees:

```
(this (is an) s expression)
```

- Computer represents represents these as **linked** structures
  - Cons cells (pairs) of a head and a tail `(cons 1 2)`
- Racket also has **structural** types (defined via structs)
  - Defined via struct; aids robustness
  - We will usually prefer agility of “tagged” S-expressions
- Also an elaborate object-orientation system (we won’t cover)

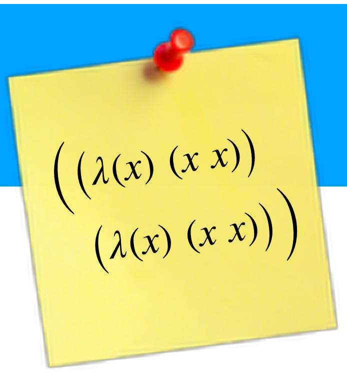
Example

```
(cons 0 1)
```

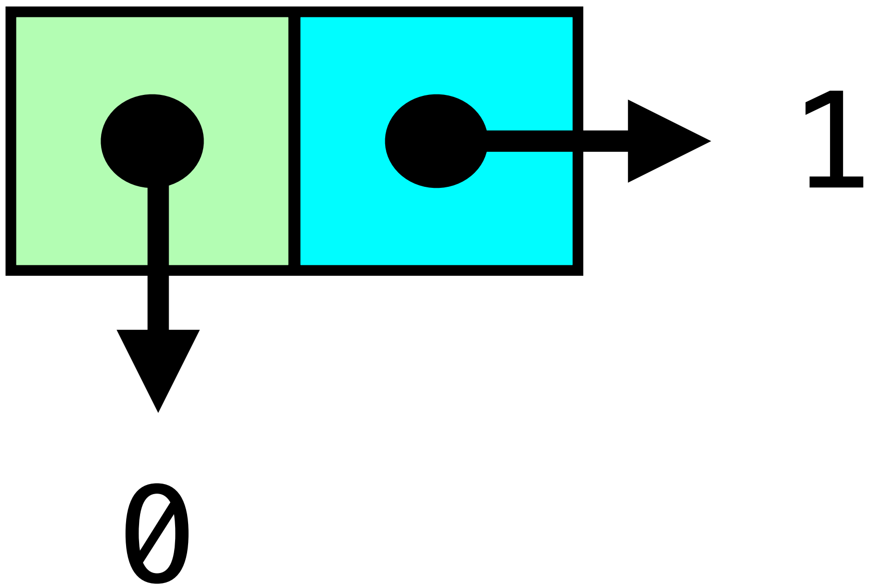


The function **cons** builds a cons cell

Example

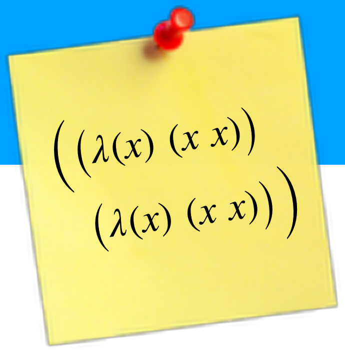


`(car (cons 0 1))` is `0`

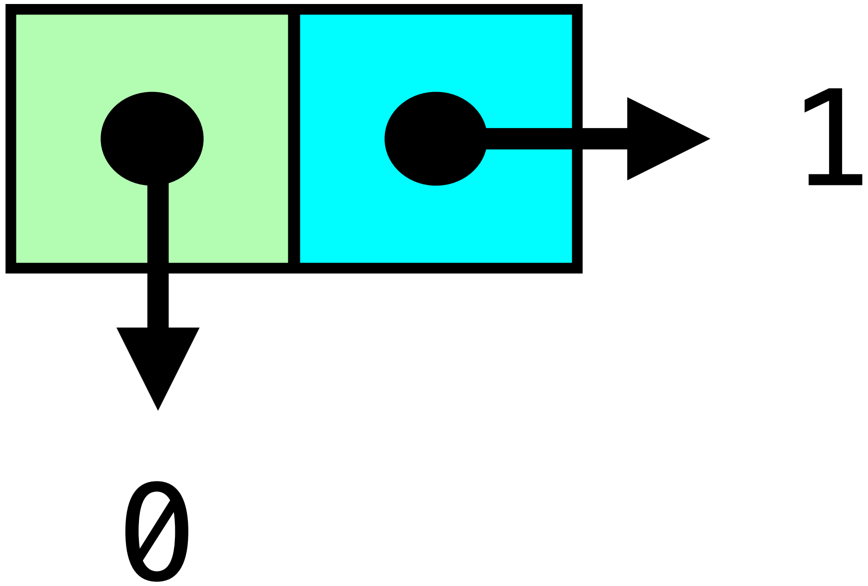


The function **car** gets the left element

Example

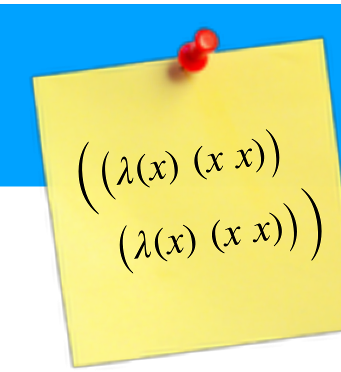


`(cdr (cons 0 1))` is 1

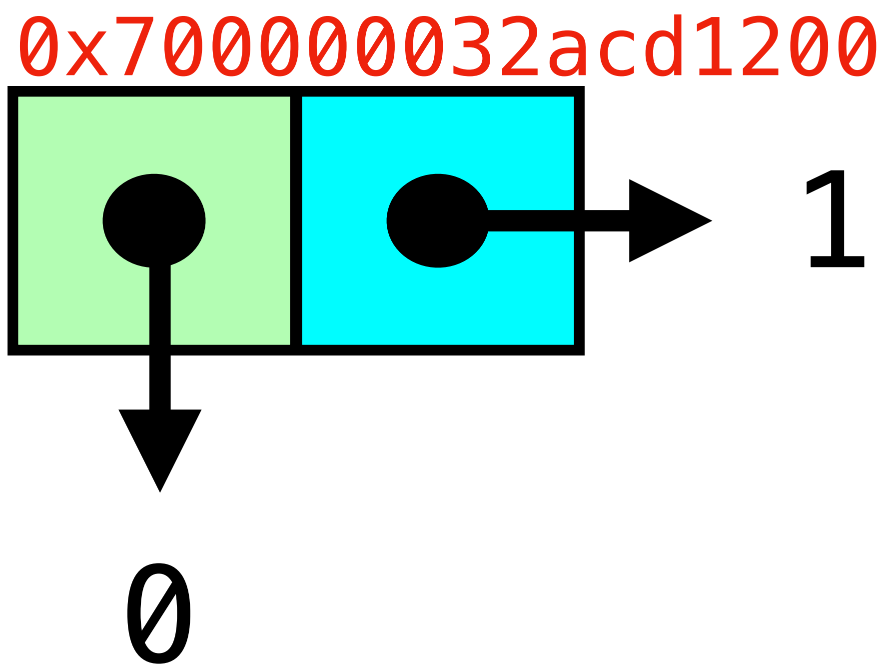


The function **cdr** gets the right element

Example

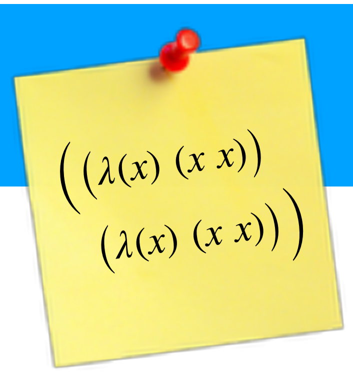



```
(cons 0 1)
```

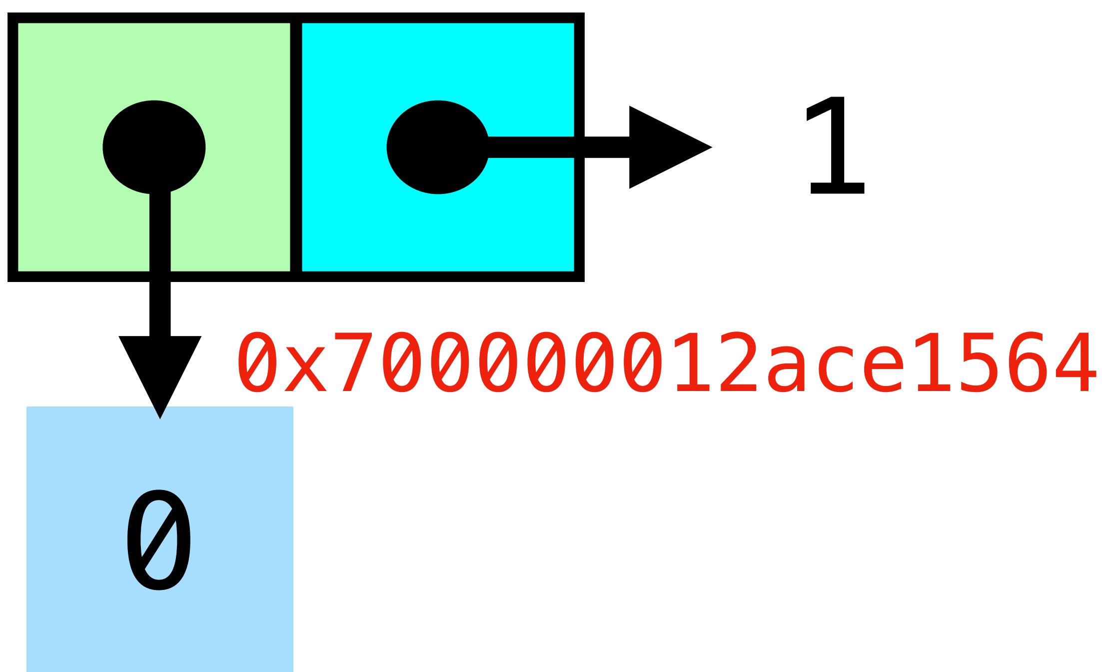


At runtime, each cons cell sits at an **address** in memory

Example

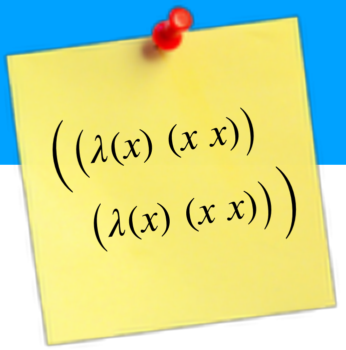



```
(cons 0 1)
```



In fact, numbers are **also** stored in memory locations.
They are thus said to be a “boxed” type

Example




```
(define x 23)
(displayln x)
(set! x 24)
(displayln x)
```

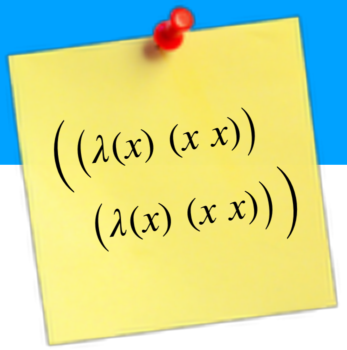
Actually, every Racket variable stores a value in some "box" (i.e., memory location)

`0x700000033dea2280`

x **23**

Prints 23

Example




```
(define x 23)
(displayln x)
(set! x 24)
(displayln x)
```

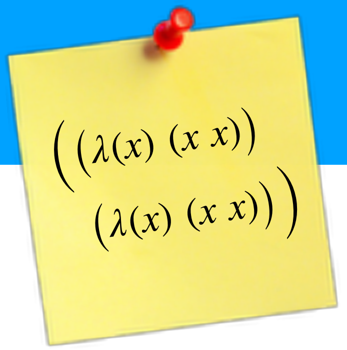
Actually, every Racket variable stores a value in some "box" (i.e., memory location)

`0x700000033dea2280`

x **24**

Changes x's value to 24

Example




```
(define x 23)
(displayln x)
(set! x 24)
(displayln x)
```

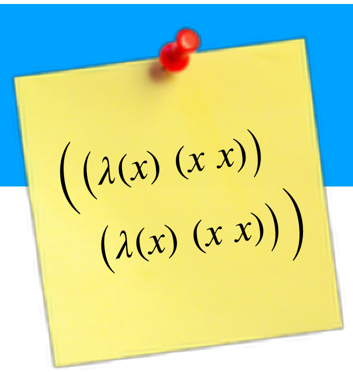
Actually, every Racket variable stores a value in some "box" (i.e., memory location)

`0x700000033dea2280`

x **24**

Now prints 24

Example



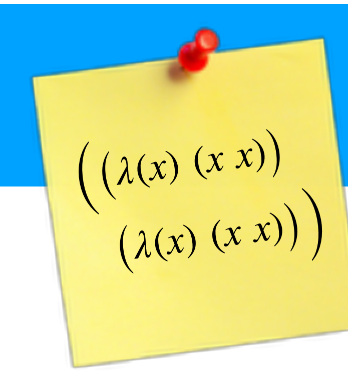

```
(define x (vector 1 2 3))
(vector-set! x 1 0)
x
;; ‘#(1 0 3)
```

Vectors (similar to arrays) are mutable, and give O(1) indexing and updating

In this class, you will not be allowed to use `set!` or `vector-set!` unless explicitly noted

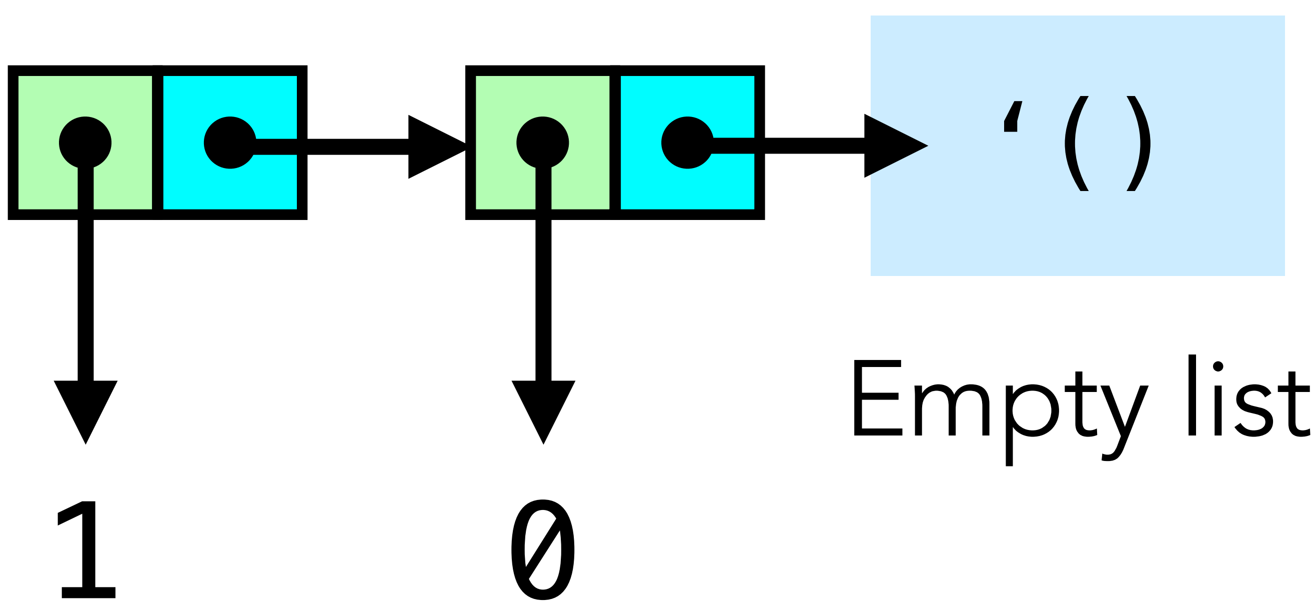
Code that uses `set!` may be denied full credit

Example



Pairs enable us to build **linked lists** of data

```
(cons 1 (cons 0 ‘()))
```



This is how Racket represents lists in memory

Example

Note that in Racket, the following are equivalent

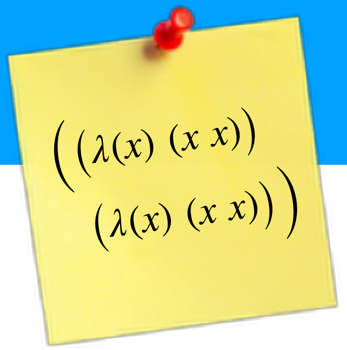
```
(cons 2 (cons 1 (cons 0 ‘())))
‘(2 1 0)
```

But the following is called an **improper list**

```
(cons 2 (cons 1 0))
‘(2 1 . 0)
```

Dot indicates a cons cell of a left and right element

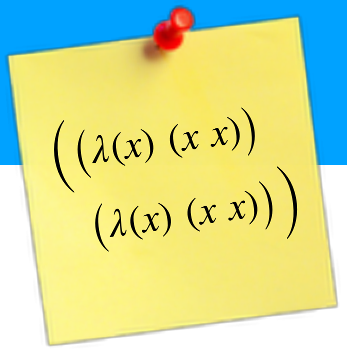
Example



Also can build **compound** expressions

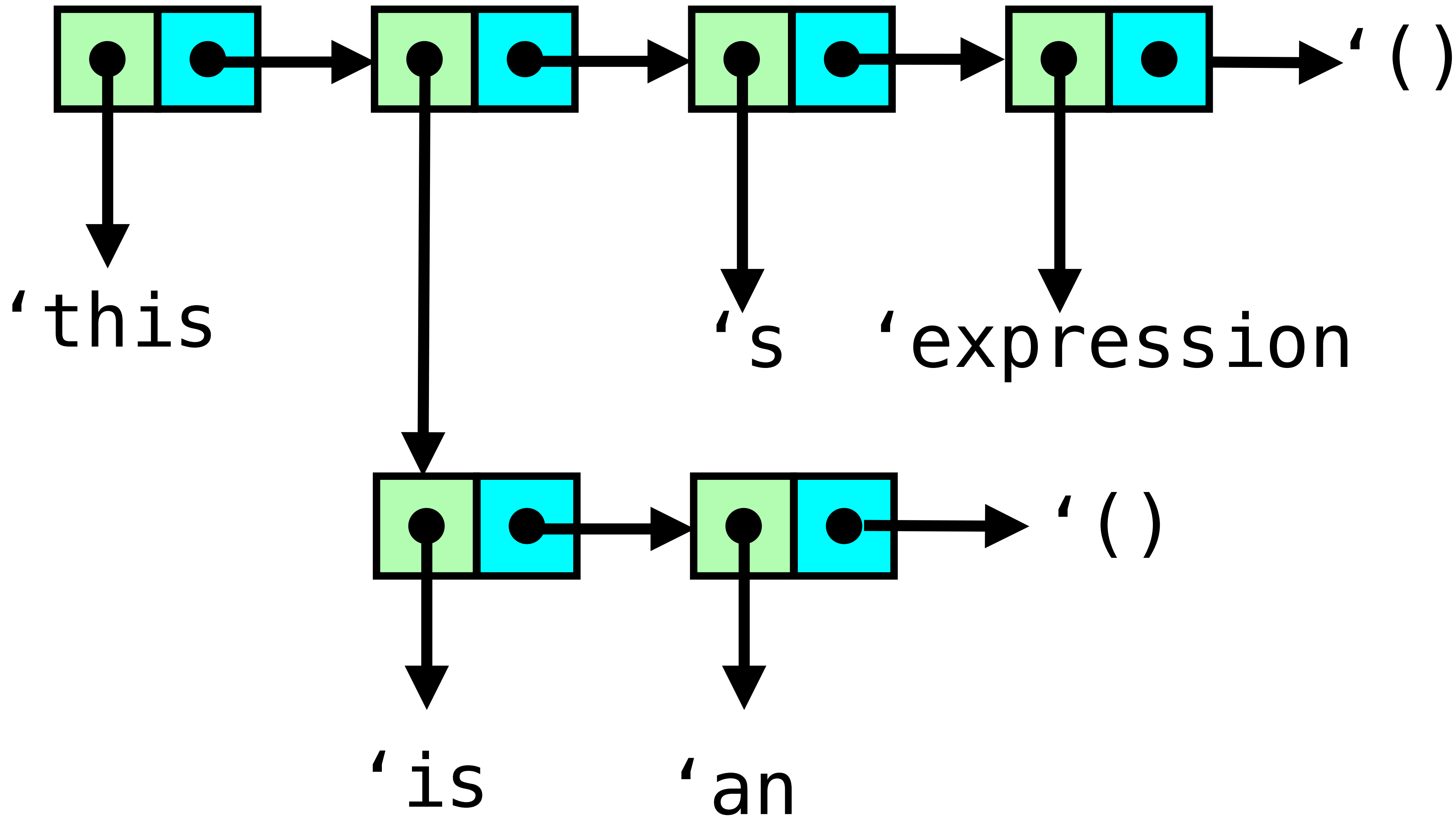
```
‘(this (is an) s expression)
```

Example

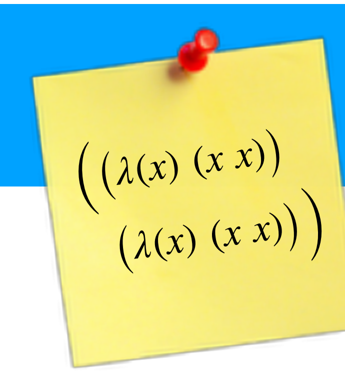


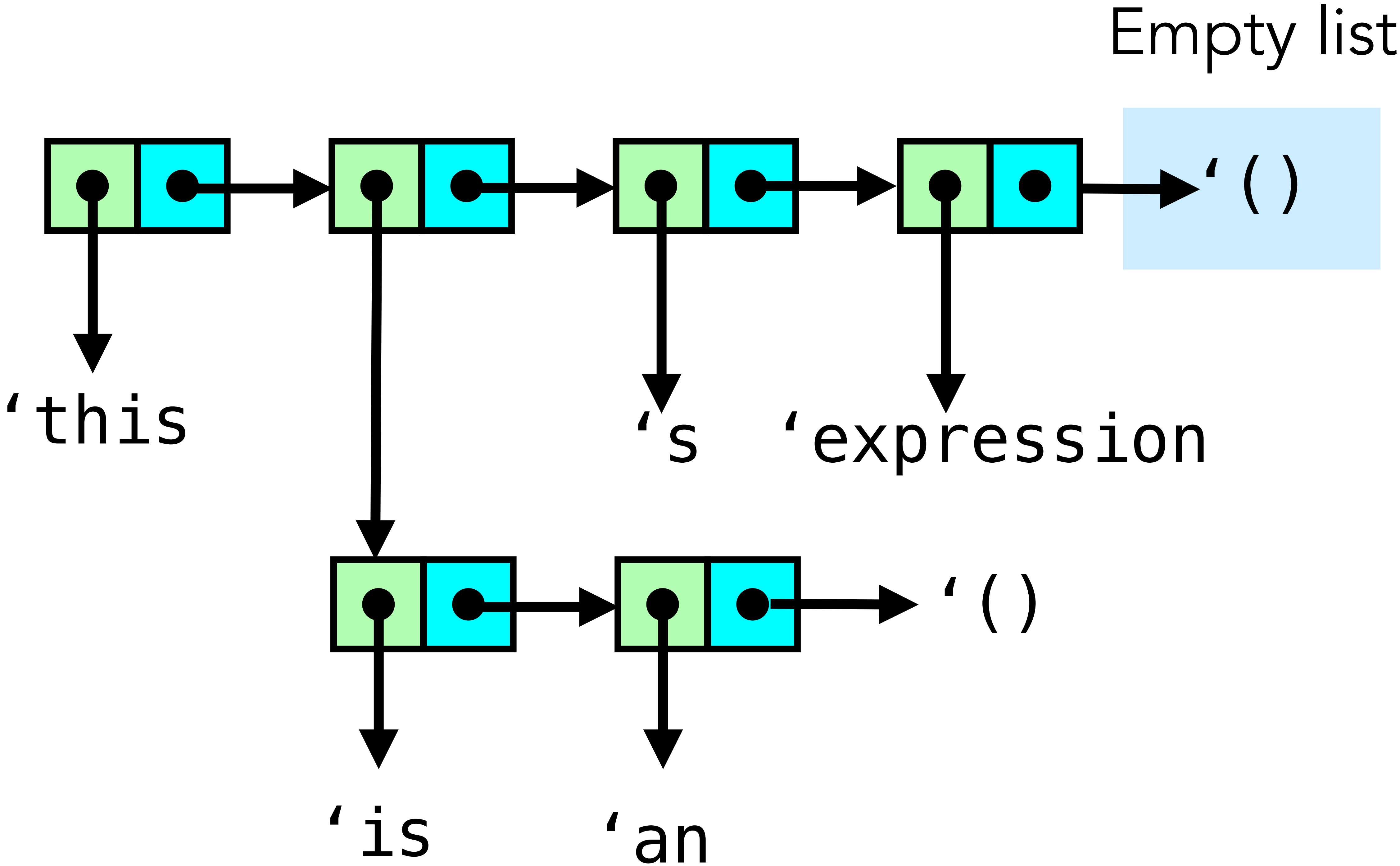
Also can build **compound** expressions

`'(this (is an) s expression)`

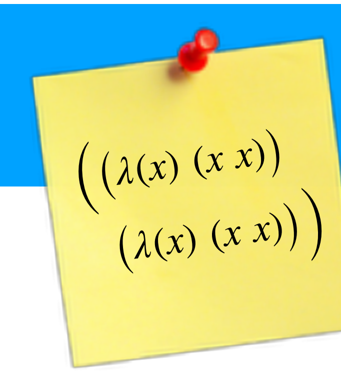


Example

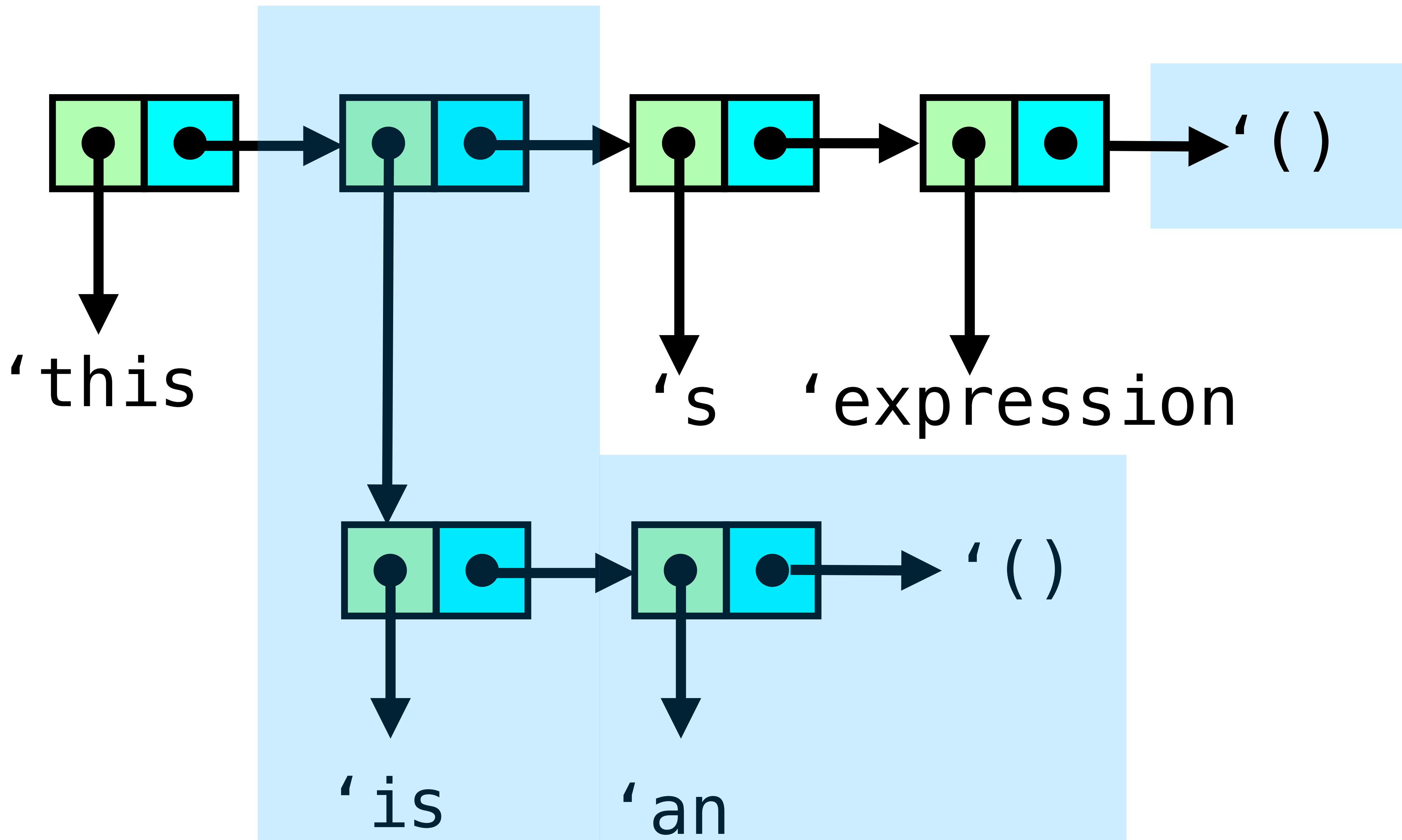




Example



Note link to compound subexpression



Exercise

Draw the cons diagram for the following…

- (cons 0 (cons 3 4))
- Is this a list? If not, what is it?
- (cons 0 (cons 3 (cons 4 ‘())))
- Is this a list? If not, what is it?

Exercise

Draw the cons diagram for the following…

- (cons 0 (cons 3 4)) — **Drawn on board**
- Is this a list? If not, what is it?
- **No, not a list, but an improper list, no empty list at end**
- (cons 0 (cons 3 (cons 4 '()))) — **Drawn on board**
- Is this a list? If not, what is it?
- **Yes, this is a list**

Binding and identifiers

- Identifiers refer to their most proximate syntactic binding
  - I.e., Racket is **statically scoped;** more later…
- Can create local bindings with the `let` form:
  - ```
    (let ([x 0]
          [y 1])
      body)
    ```
    (square brackets are the same as parens)

    x is bound to 0, y to 1, in body
  - Note that y cannot reference x! Otherwise you want "sequential let", the `let*` form
    - ```
      (let ([x 23]
            [y (* 2 x)]
        (+ y 2))
      ```
      undefined variable x!

Exercise

What is the value of the following expression?

```
(let ([a 1]
      [b 2])
  (let ([b 3]
        [c 4])
    (+ a b c)))
```

Exercise

What is the value of the following expression?

```
(let ([a 1]
      [b 2])
  (let ([b 3]
        [c 4])
    (+ a b c)))
```

8

The second definition of b ***shadows*** the first b. At the point where + is invoked on three values, b is bound most proximately to 3.

Exercise

What is the value of the following expression?

```
(let ([a 1]
      [b 2])
  (let ([b 3]
        [c (+ a b)])
    c))
```

Exercise

What is the value of the following expression?

```
(let ([a 1]
      [b 2])
  (let ([b 3]
        [c (+ a b)])
    c))
```

Although the second definition of b *shadows* the first b, when defining c, the value of b is still 2!

The new binding only takes effect in the body of the let form.

Exercise

Use `let*` to evaluate the following mathematical expression (without simplifying it), where x is 4:

$$\big((x * 2) + (x * 2) + (x * 2)\big)^2$$

Exercise

Use `let*` to evaluate the following mathematical expression (without simplifying it), where x is 4:

$$\left((x*2)+(x*2)+(x*2)\right)^2$$

```
(let* ([x 4]
       [y (* x 2)]
       [z (+ y y y)])
  (* z z))
```

Exercise

What does the following code compute?

```
(define (foo x) 1)

(let* ([f foo]
       [f (f 2)])
  (* f (let ([f 3])
         (+ f (foo f)))))
```

Exercise

What does the following code compute?

```
(define (foo x) 1)

(let* ([f foo]
       [f (f 2)])
  (* f (let ([f 3])
         (+ f (foo f)))))
```

4

Exercise

For each variable use within the following code, identify the variable's proximate binder

```
(define (foo x) 1)

(let* ([f foo]
       [f (f 2)])
  (* f (let ([f 3])
         (+ f (foo f)))))
```

Exercise

For each variable use within the following code, identify the variable's proximate binder

```
(define (foo x) 1)

(let* ([f foo]
       [f (f 2)])
  (* f (let ([f 3])
         (+ f (foo f)))))
```