Notes-CSC242

Dr. Qixin Deng
Wabash College

July 18, 2026

Contents

1 Racket Basics

1.1 Getting Started

When you launch DrRacket for the first time, you will see a window divided into two sections: Definitions and Interactions. Racket is part of a family of languages, so you need to select which one to use. Go to the `Languages` menu and choose `Choose Language` to access the language selection dialog. Select `"The Racket Language"` by clicking the first radio button. This action will add the following line at the top of the Definitions window:

```
#lang racket
```

DrRacket will remember this setting, so this line should be the first line in all your Racket programs while using this guide. Note that this line will not be repeated in subsequent examples.

Programs you write in the Definitions window can be saved and opened later using standard file dialogs. You can also hide either the Definitions or Interactions window temporarily, allowing the other to expand and occupy more space. DrRacket supports multiple tabs within a window and can also handle multiple windows.

The `Help` menu provides access to locally-installed documentation through your preferred web browser. This includes an overview found in The Racket Guide and more detailed information in The Racket Reference.

1.2 Programs, Expressions, and Values

In imperative languages, programs typically operate by continuously altering variable values. Conversely, functional programming languages emphasize applying functions (or procedures) to pre-existing values to generate new ones. While Racket includes mutation, it is downplayed; this guide will introduce it and other side effects later to highlight what can be achieved without them.

A Racket program consists of a series of definitions and expressions. When executed, each expression is evaluated sequentially, and the resulting values are displayed in the Interactions window, each on a new line. (Every Racket value has a printed representation, although some may show only limited information.) The Interactions window then offers a prompt (>) for entering additional expressions, which are immediately evaluated. This read-evaluate-print loop (REPL) is a standard and powerful feature of interpreters for functional languages.

The simplest expressions in Racket are direct values. Here are some examples:

```
42
-126765060022822940149670320537 6
3.14159
```

Racket supports exact representations of unbounded integers and rational numbers, along with complex numbers (not covered in this note). The approximation of pi shown above is an inexact number, akin to a double-precision floating point. Inexact numbers are useful when exact representations are either impossible or overly detailed; functions are available to convert between exact and inexact formats.

```
(/ 22 7)
(exact→inexact (/ 22 7))
```

While many programming introductions begin with a "Hello, world!" program, the Racket equivalent is simply a string constant expression:

```
"Hello, world!"
```

An expression can also involve applying a pre-defined function to multiple arguments. This is done by placing an open parenthesis, followed by the function name and the arguments separated by spaces, and closing with a parenthesis. For example:

```
> (max 3 -2 4.1 1/3)
4.1
```

This syntax is consistent for all function applications in Racket. Unlike many languages, Racket does not use infix notation for arithmetic operations. For example, the '+' function can take any number of numerical arguments:

```
> (× (+ 1 2 3) (- 4 5 6) (/ 7 8 9))
-49/12
```

In Racket, parentheses are not used to dictate the order of operations. Adding extra parentheses could alter the expression's meaning, so it's important to use them correctly.

1.3 Definitions

One type of definition in Racket associates a name (or identifier) with a value, potentially derived from an expression. This binding remains constant in the absence of mutation.
Another type of definition allows for the creation of user-defined functions. The function's name is followed by its parameters in parentheses and a single body expression that represents the function's computation. Applying a user-defined function follows the same syntax as using a built-in function.

```
(define x 3)
(define y (+ x 1))

(define (sum-of-squares x y)
  (+ (square x)
     (square y)))

(define (square x) (× x x))

> (sum-of-squares x y)
25
```

While multiple body expressions can be included in a function definition, only the last expression's value is retained. Without side effects, additional expressions are generally unnecessary.

```
(define (example-function x y)
  (+ x y)  ; This value is discarded
  (× x y)) ; Only this value is returned

(define result (example-function 3 4))

>(displayln (format "The result is: ~a" result))
The result is: 12
```

In many imperative languages, types are determined statically, often requiring explicit type annotations. However, Racket employs dynamic typing, where the type of a value is determined at runtime, allowing functions to accept and return different types across various calls.

The concept of scope is illustrated in the example above. For instance, within the `sum-of-squares` definition, the parameter `x` is locally scoped to the function's body. In contrast, the `x` in the definition of `y` refers to the globally scoped `x` defined earlier. This scope extends throughout the program unless a more localized definition supersedes it.

In Racket, definitions and expressions can be intermingled within a program. An expression can utilize any definitions that precede it, and the body of a function `f` can reference functions defined later in the code, provided they are defined before `f` is invoked.

```
(define a 10)

(define (sum x y)
  (+ x y))

(displayln (sum a 20)) ; Uses 'a' and 'sum' defined above

; Define a new function that references another function defined later
(define (calculate)
  (subtract a 5))  ; Uses 'subtract', which is defined later

; Define the function referenced earlier
(define (subtract x y)
  (- x y))

(displayln (calculate))

(displayln (subtract 20 5)) ; Now, 'subtract' is used after being defined
```

1.4 Style

Racket's uniform syntax offers computational benefits, but adapting to it requires some stylistic adjustments. Racket identifiers are case-sensitive, with a preference for lowercase and hyphens (-) over underscores (_). Arguments are aligned either on the same line or in the same column, and multiple closing parentheses are typically placed on the same line.

DrRacket automatically indents code when the Enter key is pressed, though this can be adjusted manually using the Space, Delete, or Tab keys. Reindentation of selected text or the entire program is also possible via menu options. If the cursor is positioned before an opening parenthesis or after a closing parenthesis, DrRacket highlights the entire sub-expression, making parentheses management easier. You can explore and customize various keyboard shortcuts for editing through the Keybindings entry in the Edit menu.

In Racket, comments begin with a semicolon (;) and extend to the end of the line. For multi-line comments, which can be nested, use `#|` to start and `|#` to end. Racket programs typically feature many small functions that often don't require in-line comments. However, a brief header explaining the purpose and arguments can be beneficial.

1.5 Conditional Expressions

To test numerical equality in Racket, the `=` function is used, which takes two or more numeric arguments and returns `#t` if all are equal, or `#f` otherwise. The Boolean values `#t` and `#f` are commonly referred to as `true` and `false`, respectively.

```
> (= 2 2)
#t

> (= 2 3)
#f

> (= (sqrt 4) 2)
#t

> (= 2 2.0)
#t

> (= 2 2 2)
#t

> (= 2 2 3)
#f

> (= "Pynchon" "DeLillo")
=: contract violation

  expected: number?
  given: "Pynchon"
  argument position: 1st
  other arguments...: "DeLillo"
```

For general equality testing across any two Racket values, the `equal?` function is used. Functions that return Boolean values typically end with a question mark by convention.

```
> (equal? 2 2)
#t

> (equal? "Wallace" "Wallace")
#t

> (equal? 2 "Joyce")
#f
```

The `not` function inverts Boolean values, mapping `#t` to `#f` and vice versa.

```
> (even? 42)
#t

```

```
> (not (even? 43))
#t

> (exact? (sqrt 4))
#t

> (exact? (sqrt 5))
#f

> (boolean? (= 3 5))
#t

> (number? (+ 3 5))
#t

> (string? "hawk")
#t

> (string<=? "hawk" "handsaw")
#f
```

An `if` expression in Racket consists of three subexpressions. The first subexpression (the "test") is evaluated, and if it is false, the third subexpression is evaluated and returned. If the test is true, the second subexpression is evaluated and returned.

```
> (define x -2)
> (if (> x (sqrt 5)) "greater" "not greater")
"not greater"

> (if (positive? x) x (- x))
2
```

It's important to note that the test subexpression in `if` doesn't necessarily need to produce a Boolean value, although it typically does. Unlike functions, `if` is a form, meaning not all of its arguments are evaluated. In Racket, `if` expressions always have a value, and both the second and third subexpressions are mandatory. Unlike in imperative languages, nested `if` expressions are less common in Racket, as the more versatile and readable `cond` expression is often used instead.

A `cond` expression consists of multiple question-answer pairs, each enclosed in square brackets. The questions are evaluated in order, and when one does not return `false`, its corresponding answer is evaluated and returned. The final question can be `else`, in which case its corresponding answer is returned.

```
(define (classify-number x)
  (cond
    [(< x 0) "Negative number"]
    [(= x 0) "Zero"]
    [(and (> x 0) (< x 10)) "Positive number less than 10"]
    [(>= x 10) "Positive number 10 or greater"]
    [else "Unknown"])  ;; The else case for any other scenario
)

> (classify-number 0)
"Zero"
> (classify-number 9)
"Positive number less than 10"
```

Square brackets are used here for readability and can be interchanged with parentheses without altering the program's behavior.

More complex conditions can be created using `and` and `or`. The `and` expression evaluates its arguments sequentially until one is `false`, in which case the whole expression evaluates to `false`. If all arguments are true, the expression evaluates to the value of the last argument. The `or` expression, on the other hand, evaluates to the first true value it encounters, or `false` if none exist.

```
(define (f x) (if (positive? x) x #f))
```

```
> (and (> 3 4) (< 3 4))
#f

> (and (f 3) (f 4))
4

> (and (f -2) (f 3))
#f

> (or (f -2) (f 3) (f 4))
3
```

1.6 Compound Data

In Racket, two values can be combined using `cons`.

```
> (cons 3 4)
'(3 . 4)

> (cons #t "Lowry")
'(#t . "Lowry")
```

The printed format of a `cons` value changes when it forms a list. A list can be either empty or structured as `(cons v lst)`, where `v` is any Racket value and `lst` is a list. Lists can contain values of different types.

```
> empty
'()

> (cons 3 empty)
'(3)

> (cons true (cons "blue" empty))
'(#t "blue")

> (cons (cons 1 (cons 2 empty)) (cons 3 (cons 4 empty)))
'((1 2) 3 4)
```

A `cons` value can be deconstructed using the functions `car` and `cdr`. If the value is a list, the more intuitive `first` and `rest` can be used instead.

```
> (car (cons 3 4))
3

> (cdr (cons 3 4))
4

> (first (cons true (cons "blue" empty)))
#t

> (rest (cons (cons 1 (cons 2 empty)) (cons 3 (cons 4 empty))))
'(3 4)
```

The quote notation used to display list values can also be utilized in programs, but only for specifying lists whose values are known at the time of writing. The more general `list` function creates a list from its arguments. Racket provides utility functions like `second` through `tenth` to access elements of a list, along with several other useful functions.

```
> (list (+ 1 2) (+ 3 4))
'(3 7)

> (third '(1 2 3 4 5))
3

> (append '(1 2) '(3 4 5))
'(1 2 3 4 5)
```

```
> (list-ref '(6 7 8 9 10) 3)
9

> (length '(6 7 8 9 10))
5

> (take '(1 2 3 4 5) 3)
'(1 2 3)

> (drop '(1 2 3 4 5) 3)
'(4 5)

> (reverse '(1 2 3 4 5))
'(5 4 3 2 1)
```

The expression `'(1 2)` is shorthand for `(quote (1 2))`. Quoting a single identifier results in a symbol, a form of atomic data with constant-time equality checks. Symbols like `'nabokov` and `'lowry` are useful as human-readable, distinct values, such as tags. Generally, quoting an expression starting with an open parenthesis results in all subexpressions being quoted and assembled into a list. Numbers, strings, and values specified with a hash-sign (e.g., `#t` and `#f`) quote to themselves.

The `quote` function provides a concise way to construct nested lists, while `list` is more verbose but allows for evaluation. The `quasiquote` (abbreviated `) offers both benefits, acting like `quote` but allowing subexpressions with `unquote` (abbreviated `,`) to escape to regular evaluation.

```
> '(1 2 ,(length '(1 2 3)) 4 5)
'(1 2 ,(length '(1 2 3)) 4 5)

> `(1 2 ,(length '(1 2 3)) 4 5)
'(1 2 3 4 5)
```

1.7 Modules

A Racket program can be split across multiple source files, with each file beginning with a `#lang` line serving as a module. Modules can share bindings with other modules using the `provide` form to export and the `require` form to import.

The `provide` form, placed at the top level, lists identifiers bound in the module that are available for export. There are variations to export all identifiers, rename bindings, or export all bindings from a structure.

The `require` form specifies which modules or libraries to import, either by using a file path or by referencing installed libraries. For instance, `(require "my-helpers.rkt")` imports from a local file, while `(require net/url)` loads a library for web-related utilities. Options exist to selectively import bindings, rename them, or exclude certain imports to avoid conflicts.

Top-level expressions in a module are evaluated when the module is run from the Definitions window, with results shown in the Interactions window. If a module is imported using `require`, the expressions are evaluated but not printed.

Racket supports submodules, including two useful cases: a `main` submodule runs automatically when the module is executed but not when required, while a `test` submodule can be run from the command line but not when the module itself is run.

While this overview covers common uses of modules, Racket's module system is more versatile. Library documentation typically begins with the necessary `require` expression, and Racket's package system offers additional libraries not installed by default. For further information, see Racket's Package Management documentation.

1.8 Testing

You can perform on-the-fly testing by entering expressions into DrRacket's Interactions window. If these expressions are included in the Definitions window, they will be evaluated each time the program runs. However, this requires manually checking if the computed values match the expected results. By pairing computed and expected values using an equality predicate, you can identify mismatches by spotting `#f` in the results. String labels can be used to identify each test.

The teaching languages offer a testing framework with a graphical user interface. This framework, although without the GUI, can be accessed by requiring the `test-engine/racket-gui` module. The `check-expect` form can then be used to rewrite the above tests:

A `check-expect` expression is not evaluated immediately, allowing it to be part of the documentation preceding a function definition. All such expressions are evaluated when the following expression is executed:

```
(require test-engine/racket-tests)
;; Define the factorial function
(define (factorial n)
  (if (= n 0)
      1
      (× n (factorial (- n 1)))))


(check-expect (factorial 5) 120)

> (test)
The test passed!
```

This produces a detailed report of any failed tests, including the line number, expected value, and computed value, in a separate window. If the `test-engine/racket-tests` module is required instead, the report is generated in the Interactions window (or written to standard output if using the command-line version of Racket).

The testing modules also include `check-error`, which verifies if its argument expression produces an error when evaluated; `check-within`, which tests whether a computed inexact value falls within a specified tolerance of an expected value; and `check-member-of`, which allows for multiple possible expected values.

```
(require test-engine/racket-tests)

;; A function that causes an error when dividing by zero
(define (safe-divide x y)
  (if (= y 0)
      (error "Division by zero!")
      (/ x y)))

;; A function that returns an inexact square root
(define (approx-sqrt x)
  (sqrt x))

;; A function that returns a random number from 1 to 3
(define (random-number)
  (list-ref '(1 2 3) (random 3)))

;; Test cases

;; check-error: verifies that safe-divide produces an error when dividing by zero
(check-error (safe-divide 10 0))

;; check-within: tests that the square root of 2 is approximately 1.414 within a tolerance of 0.001
(check-within (approx-sqrt 2) 1.414 0.001)

;; check-member-of: checks if the result is one of the specified possible values
(check-member-of (random-number) 1 2 3)
```

These testing modules are primarily intended for beginners. For more advanced testing, the `rackunit` module is recommended, especially when combined with submodules, to effectively test larger projects.

2 Haskell Basics

2.1 Getting Started

Haskell is a functional programming language specifically designed for symbolic computation and list processing tasks. Functional programming, the foundation of Haskell, is centered on the use of mathematical functions. Other well-known languages that embrace the functional programming paradigm include Lisp, Python, Erlang, Racket, F#, Clojure, and others.

In traditional programming, code is written as a series of declarations in a specific syntax, whereas functional programming treats all computations as compositions of distinct mathematical functions.
Now that everything is set up, let's dive in! The Haskell compiler, GHC, includes an interactive interpreter called GHCi, perfect for experimenting with Haskell. Start GHCi by running `ghci` in your command prompt. Let's try a simple calculation to test Haskell's capabilities:

```
ghci> 6 + 3^2 * 4
42
```

Now, what about finding the first 10 even numbers after 42?

```
ghci> take 10 (filter even [43..])
[44,46,48,50,52,54,56,58,60,62]
```

And the sum of these numbers?

```
ghci> sum it
530
```

Note: GHCi has a special `it` variable that remembers the result of the last expression. Congratulations, you've had your first taste of Haskell! Let's move on to running a full program using Haskell compiler.

In your text editor, create a new file named `hello.hs`. Write the following code:

```
main = do
  putStrLn "Hello, everybody!"
  putStrLn ("Please look at my favorite odd numbers: " ++ show (filter odd [10..20]))
```

You can compile this code using GHC to create an executable named `hello`, which you can then run:

```
> ghc hello.hs
> ./hello
Hello, everybody!
Please look at my favorite odd numbers: [11,13,15,17,19]
```

Congratulations, you've just written a simple, polite Haskell program!
GHCi Tip: You can also load your file directly into GHCi to interactively experiment with any functions and definitions you've created. For example:

```
> ghci hello.hs
> main
Hello, everybody!
Please look at my favorite odd numbers: [11,13,15,17,19]
```

2.2 Basic Data Types

2.2.1 Number

Haskell is capable of inferring the type of a number automatically, so there is no need to explicitly declare the type as you would in other programming languages. For example, you can enter a simple expression like `2+2` directly into the GHCi prompt:

```
ghci> 2+2
4
```

In this example, the GHCi compiler interprets the inputs as numbers without requiring type annotations. Next, let's try a more complex calculation to see if the compiler handles it correctly:

```
ghci> 15+(5*5)-40
0
```

The expression returns the expected result of `0`.

2.2.2 Character

Similar to numbers, Haskell can automatically recognize a character provided as input. You can enter a character within single quotes at the Haskell command prompt. For instance, try the following:

```
ghci> :t 'a'
"a" :: Char
```

The `:t` command displays the type associated with the input. In this case, it identifies `"a"` as a list of characters, or `[Char]`.
If you enter an invalid character without quotes, Haskell will produce an error:

```
ghci> :t a
<interactive>:1:1: Not in scope: 'a'
ghci> a
<interactive>:4:1: Not in scope: 'a'
```

The error message `Not in scope: 'a'` indicates that Haskell does not recognize the input as valid. Haskell uses ASCII encoding to represent characters. For example:

```
ghci> '\97'
'a'

ghci> '\67'
'C'
```

Here, your numeric input is decoded into its corresponding ASCII character.

2.2.3 String

In Haskell, a string is simply a sequence of characters. Although there is no special syntax for strings, they are conventionally represented using double quotes.

For example, consider the string `"wabash.com"`:

```
ghci> :t "wabash.com"
"wabash.com" :: String
```

2.2.4 Boolean

The Boolean data type in Haskell is straightforward, similar to other data types. Below are examples of Boolean operations using values like `True` and `False`:

```
ghci> True && True
True
ghci> True && False
False
```

In these examples, Haskell automatically recognizes `True` and `False` as Boolean values and performs the operations accordingly. However, using lowercase inputs like `true` or `false` will cause an error:

```
ghci> true
<interactive>:9:1: Not in scope: 'true'
```

Haskell cannot interpret `true` as a Boolean value, resulting in an error indicating that the input is out of scope.

2.2.5 List

In Haskell, a list is a versatile and commonly used data type. For example, `[a,b,c]` is a list of characters, which means a list in Haskell is a collection of elements of the **same** data type, separated by commas. You don't need to explicitly declare a list as a list; Haskell automatically infers it based on the syntax. For instance:

```
ghci> [1,2,3,4,5]
[1,2,3,4,5]
```

Haskell's lists are homogeneous, meaning they cannot contain elements of different types. Attempting to mix types in a list will result in an error:

```
ghci> [1,2,3,4,5,'a','b']
<interactive>:17:12: Not in scope: 'a'
```

List comprehension in Haskell allows you to generate lists using mathematical expressions. Consider the following examples:

```
ghci> [x*2 | x <- [1..10]]
[2,4,6,8,10,12,14,16,18,20]

ghci> [x*2 | x <- [1..5]]
[2,4,6,8,10]

ghci> [x | x <- [1..5]]
[1,2,3,4,5]
```

2.2.6 Tuple

Haskell offers a way to group multiple values into a single data structure called a Tuple. While similar to a list, there are key differences between Tuples and Lists. Tuples are immutable, meaning their number of elements cannot be altered at runtime, unlike lists, which are mutable. Furthermore, lists are homogeneous (containing elements of the same type), whereas Tuples are heterogeneous, allowing different types of data within a single Tuple. Tuples are enclosed in parentheses. For example:

```
ghci> (1,1,'a')
(1,1,'a')
```

2.3 Operators

Below is an example for addition operator `+`, the subtraction operator `(-)`, the multiplication operator `(*)`, and the division operator `(/)`:

```
main = do
   let var1 = 2
   let var2 = 3
   putStrLn "The addition of the two numbers is:"
   print(var1 + var2)
   putStrLn "The subtraction of the two numbers is:"
   print(var1 - var2)
   putStrLn "The multiplication of the two numbers is:"
   print(var1 * var2)
   putStrLn "The division of the two numbers is:"
   print(var1 / var2)
```

The output will be:

```
The addition of the two numbers is:
9.0
The subtraction of the two numbers is:
3.0
The multiplication of the two numbers is:
18.0
The division of the two numbers is:
2.0
```

Haskell uses the sequence operator `(..)` to generate lists with a range of values. For example, to print numbers from 1 to 10:

```
main :: IO()
main = do
   print [1..10]
```

The output will be:

```
[1,2,3,4,5,6,7,8,9,10]
```

2.4 Conditional Statements

Decision Making is a feature that allows the programmers to apply a condition in the code flow. The programmer can execute a set of instructions depending on a predefined condition. The general syntax for using the `if-else` conditional statement in Haskell is as follows:

`if <Condition> then <True-Value> else <False-Value>`

- **Condition**: The expression that will be evaluated.
- **True-Value**: The result if the condition is true.
- **False-Value**: The result if the condition is false.

Since Haskell treats code as mathematical expressions, omitting the `else` block will result in an error. The example below demonstrates the use of an `if-else` statement:

```
main = do
   let var = 23
   if var `rem` 2 == 0
      then putStrLn "Number is Even"
   else putStrLn "Number is Odd"
```

The output will be:

```
Number is Odd
```

In Haskell, multiple `if` statements can be chained together by linking each `if` with its corresponding `else` statement. The following example demonstrates how to implement nested `if-else` statements in Haskell:

```
main = do
   let var = 26

   if var == 0
      then putStrLn "Number is zero"
   else if var `rem` 2 == 0
            then putStrLn "Number is Even"
        else putStrLn "Number is Odd"
```

The output will be:

```
Number is Even
```

2.5 Types and Type Class

2.5.1 Built-in Type Class

Haskell is a functional programming language with strict typing, meaning that the data types used throughout an application are known to the compiler at compile time. In Haskell, each statement is treated as a mathematical expression, and the category of this expression is known as its *Type*. A *Type* can be seen as the data type of the expression used during compilation. To explore the type of an expression, the `:t` command is used. A *Type Class* is a set of similar types grouped together.
`Int` is a type representing integer values ranging from `-2147483647` to `2147483647`. The function `fType` below demonstrates its usage:

```haskell
fType :: Int -> Int -> Int
fType x y = x*x + y*y
main = print (fType 2 4)
```

Here, `fType` is defined to take two `Int` values and return an `Int`. The output will be:

```
The addition of the two numbers is:
5
```

`Integer` is not bounded by any specific range, allowing for any length. To see the difference between `Int` and `Integer`, consider the modified example:

```haskell
fType :: Integer -> Integer -> Integer
fType x y = x*x + y*y
main = print (fType 212124454 44545454454554545454454454544545)
```

This code will successfully produce the output:

```
19842975125627933958826446313642976860992103025773740551 41
```

`Float` represents floating-point numbers. Here's an example:

```haskell
fType :: Float -> Float -> Float
fType x y = x*x + y*y
main = print (fType 2.5 3.8)
```

The output will be:

```
20.689999
```

`Double` is a floating-point number with double precision. Example:

```haskell
fType :: Double -> Double -> Double
fType x y = x*x + y*y
main = print (fType 2.56 3.81)
```

The output will be:

```
21.0697
```

`Bool` is a Boolean type with values `True` or `False`. Example:

```
main = do
   let x = True
   if x == False
      then putStrLn "X matches with Bool Type"
   else putStrLn "X is not a Bool Type"
```

The output will be:

```
X is not a Bool Type
```

`Char` represents characters enclosed in single quotes. Here's a modified `fType` function:

```
fType :: Char -> Char
fType x = 'K'
main = do
   let x = 'v'
   print (fType x)
```

The output will be:

```
'K'
```

2.5.2 Type Classes

Type classes provide a way to define behavior that can apply to multiple data types, while the specific data types define the kind of data being used.

The `Eq` type class provides an interface for testing equality between expressions. Any type that needs to compare values for equality must belong to this class. All standard types in Haskell are part of the `Eq` class, and equality is typically checked using the `==` or `/=` operators. For example:

```
main = do
   if 8 /= 8
      then putStrLn "The values are Equal"
   else putStrLn "The values are not Equal"
```

The output will be:

```
The values are not Equal
```

The `Ord` type class provides an interface for ordering values. All types that we've discussed are part of this class. The `Ord` interface is accessed using operators like `>`, `<`, `<=`, `>=`, and the `compare` function. For instance:

```
main = print (4 <= 2)
```

The output will be:

```
False
```

The `Show` type class allows for converting values into strings. Regardless of the input type, the result is always a string. The `show` function is used to invoke this class:

```
main = print (show [1..10])
```

The output will be:

```
"[1,2,3,4,5,6,7,8,9,10]"
```

The `Enum` type class supports sequential or ordered data types. It includes functions like `succ`, `pred`, and works with types like `Bool`, `Char`, etc. Example:

```
main = print (succ 12)
```

The output will be:

```
13
```

The `Num` type class handles numeric operations and includes types such as `Int`, `Integer`, `Float`, and `Double`. Example:

```
main = do
   print(2 :: Int)
   print(2 :: Float)
```

The output will be:

```
2
2.0
```

2.5.3 Costum Type

Haskell allows defining **custom** types. Here's an example of a user-defined type, Area is the data type, Circle is the constructor:

```
data Area = Circle Float Float Float
surface :: Area -> Float
surface (Circle _ _ r) = pi * r ^ 2
main = print (surface $ Circle 10 20 10)
```

The output will be:

```
314.15927
```

2.5.4 Type Variable

In Haskell, type variables allow functions and data types to be generic, enabling them to work with any type. This tutorial provides an overview of how type variables function in Haskell.

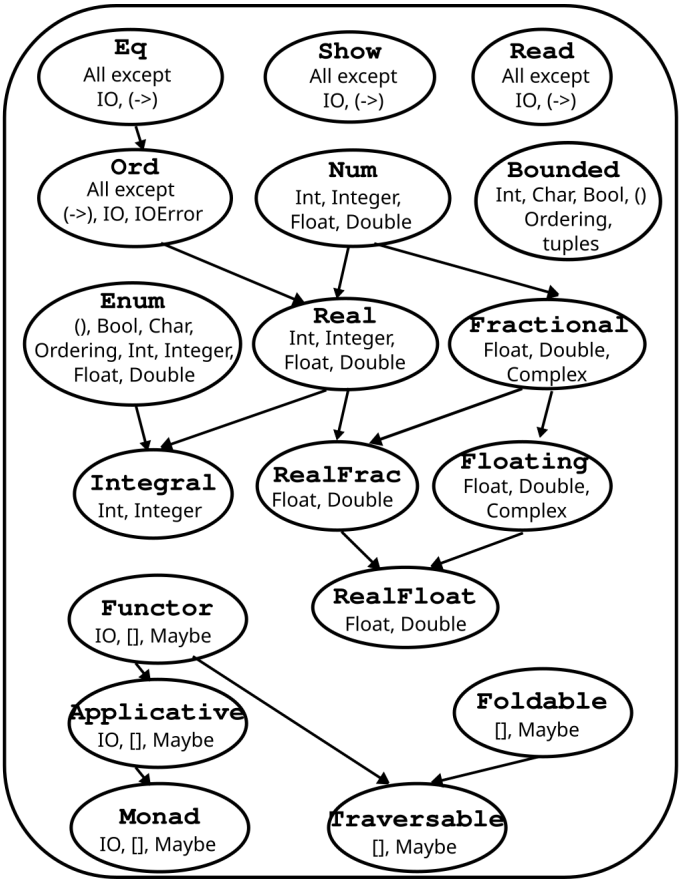
A type variable is a placeholder for any type. It is usually represented by a lowercase letter like `a`, `b`, etc. Type variables enable the creation of functions and data types that are not restricted to a specific type. Consider a function that returns the first element of a list:

```
firstElement :: [a] -> a
firstElement (x:_) = x
```

Here, `[a]` denotes a list of elements of any type `a`. The function `firstElement` works with a list of any type and returns an element of that same type. There are some benefits for using type variables:

- **Reusability**: The same function can operate on lists of integers, characters, or any other type.
- **Type Safety**: Even though the function is generic, Haskell ensures that the types are consistent.

Type variables can be constrained by type classes. For example:




```haskell
add :: Num a => a -> a -> a
add x y = x + y
```

Here, `a` can be any type that is an instance of the `Num` type class, ensuring that `add` works with numeric types.

2.6 Functions

Functions are central to Haskell, a functional programming language. As in other languages, Haskell uses function declarations and definitions.
Function Declaration: Specifies the function name, its arguments, and the return type.
Function Definition: Contains the actual implementation of the function.
For example, consider the following function that adds two integers:

```haskell
add :: Integer -> Integer -> Integer    -- function declaration
add x y =  x + y                        -- function definition

main = do
   putStrLn "The addition of the two numbers is:"
   print(add 2 5)     -- calling the function
```

In this example, the function `add` is declared in the first line and defined in the second line. It takes two integers as input and returns their sum. When executed, the output will be:

```
The addition of the two numbers is:
7
```

Pattern matching is a technique used to simplify code by matching specific types of expressions. It serves as an alternative to `if-else` statements and can be applied to any type class. Pattern matching is a form of dynamic polymorphism, allowing different methods to be executed depending on the argument list at runtime. Consider the following example, which calculates the factorial of a number using pattern matching:

```haskell
-- Define a function that describes a list
describeList :: [a] -> String --[a] represents a list of elements of any type a.
                              --The a is a type variable, meaning it can be any type (e.g., Int, Char, etc.).
describeList [] = "The list is empty."
describeList [x] = "The list has one element."
describeList [x, y] = "The list has two elements."
describeList (x:xs) = "The list has more than two elements."

main :: IO ()
main = do
    print (describeList [])         -- Output: "The list is empty."
    print (describeList [1])        -- Output: "The list has one element."
    print (describeList [1, 2])     -- Output: "The list has two elements."
    print (describeList [1, 2, 3])  -- Output: "The list has more than two elements."
```

The output will be:

```
"The list is empty."
"The list has one element."
"The list has two elements."
"The list has more than two elements."
```

Guards provide a way to test specific properties of an expression, similar to pattern matching but more readable. Although pattern matching is preferred, guards are functionally different and can resemble `if-else` statements.

```haskell
-- Define a function that classifies a number as positive, negative, or zero
classifyNumber :: Int -> String
classifyNumber n
    | n > 0     = "Positive"
    | n < 0     = "Negative"
    | otherwise = "Zero"

main :: IO ()
main = do
    print (classifyNumber 5)    -- Output: "Positive"
    print (classifyNumber (-3)) -- Output: "Negative"
    print (classifyNumber 0)    -- Output: "Zero"
```

The output will be:

```
"Positive"
"Negative"
"Zero"
```

The `where` clause allows you to simplify complex expressions by breaking them into smaller parts. This is particularly useful when dealing with intricate mathematical expressions.

```haskell
-- Function to calculate the roots of a quadratic equation
quadraticRoots :: Float -> Float -> Float -> (Float, Float)
quadraticRoots a b c = (root1, root2)
  where
    discriminant = b^2 - 4*a*c
    root1 = (-b + sqrt discriminant) / (2*a)
    root2 = (-b - sqrt discriminant) / (2*a)

main :: IO ()
main = do
    let (r1, r2) = quadraticRoots 1 (-8) 6
    putStrLn ("The roots are: " ++ show r1 ++ " and " ++ show r2)
```

The output will be:

```
The roots are: 7.1622777 and 0.8377223
```

3 Programs and Programming Languages

What is a program? Typically, we see programs as either active entities that perform tasks on our computers or as the source code that instructs the computer on what to do. However, it's crucial to remember that code is not the end goal; it's a way to communicate our intentions to the computer.

A programming language is more than just a tool for writing code—it's a true language, designed to be precise, unambiguous, and perfectly understood by machines. Unlike natural languages, which are often ambiguous and nuanced, programming languages must be clear and rigid to function correctly. This strictness can be challenging for beginners, but it allows us to harness the computational power of modern technology.

The goal of this course, then, is to stop taking programming languages for granted; to go deeper, from users of programming languages to understanding the design and implementation of these languages.

3.1 Imperative Programming

Imperative programming languages, such as Python, Java, and C++, are focused on instructing the computer on **what steps to perform** and **in what order** these steps should be executed. This paradigm is characterized by the use of statements that alter the program's state as it runs.

3.1.1 Side Effects

A **side effect** refers to any external action performed by a function, aside from returning a value. Side effects can include:

1. **Mutating an external variable**: Changing the value of a variable that exists outside the function.
2. **Printing to the screen**: Displaying information to the user.
3. **Writing file or network I/O operations**: Interacting with external files or network resources.

3.1.2 Pure vs Impure Functions

- **Pure Function**: A function is considered pure if it has no side effects. The return value of a pure function depends solely on its input values, and it does not modify any external state.
- **Impure Function**: Conversely, an impure function may involve side effects, thus its return value might depend on both its inputs and the mutable state outside the function.

Consider the following Python code, which calculates the nth Fibonacci number and tracks the maximum Fibonacci number computed so far:

```python
# External variable
counter = 0

def accumulate_sum(numbers):
    global counter   # Access external variable
    total = 0
    for n in numbers:
        total += n
        counter += 1   # Changes external state each loop iteration
    return total
```

This code demonstrates key aspects of imperative programming:

- Statements that **mutate states**, such as updating variables.
- Use of **impure functions** with side effects, like modifying the global variable `counter`.
- Control flow that includes **loops** to manage execution order.

I slightly modify the function above:

```python
def accumulate_sum(numbers):
    counter = 0
    total = 0
    for n in numbers:
        total += n
        counter += 1
    print(counter)
    return total
```

Is the function pure or not? Why?

3.2 Functional Programming vs Imperative Programming

The following table highlights the fundamental differences between imperative and functional programming paradigms:

Characteristic	**Imperative Programming**	**Functional Programming**
Programmer Focus	How to perform tasks (algorithms) and how to track changes in state.	What information is desired and what transformations are required.
State Changes	Important	Non-existent
Order of Execution	Important	Low importance
Primary Flow Control	Loops, conditionals, and function calls	Function calls, including recursion
Primary Manipulation Unit	Instances of structures or classes	Functions as first-class objects and data collections
Languages	C++, Java	Scheme, Racket, Haskell

The factorial function can be written in both functional and imperative styles, as shown below:

Haskell (Functional Style)

```haskell
import Lib

main :: IO ()
factorial 0 = 1
factorial n = n * factorial (n - 1)

main = print (factorial 3)
```

Python (Imperative Style)

```python
def factorial(n):
    f = 1
    for i in range(1, n + 1):
        f = f * i
    return f

print(factorial(3))
```

Python (Functional Style)

```python
def factorial(n):
    if n < 1:
        return 1
    return n * factorial(n - 1)

print(factorial(3))
```

3.3 Syntax and Grammar

3.3.1 Syntax

Syntax refers to the set of rules that dictate how the symbols, keywords, and operators of a programming language can be arranged to form valid programs. It defines the structure or format of the code, analogous to grammar rules in natural languages that dictate how words are arranged to form valid sentences.
Structure and Order: In a programming language, syntax determines the correct order and structure of elements in the code. For example, in Racket, an `if` expression must follow a specific structure:

```
(if (> x 10)
    (display "x is greater than 10")
    (display "x is less than or equal to 10"))
```

The syntax rules here specify that the condition must be enclosed in parentheses, and the two branches (true and false cases) must also be enclosed in parentheses.

In Haskell, consider a function that adds two numbers:

```
add :: Int -> Int -> Int
add x y = x + y
```

The syntax rules dictate that the type declaration uses `::` to specify the function's type, and the function implementation uses `=` to define the function body. A syntax error would occur if, for example, the `=` sign was omitted or misplaced:

```
add x y  x + y  -- Syntax error: missing '='
```

Keywords and Symbols: Syntax rules also dictate how keywords (like `if`, `let`, `where`, etc.) and symbols (like `()`, `->`, `=`) must be used. Incorrect use of these elements—like missing a parenthesis or incorrectly using a symbol—results in syntax errors that prevent the program from being interpreted or compiled correctly.

Syntax Errors: If you violate the syntax rules of a language, the interpreter or compiler will generate an error. For example, in Racket, the following code would produce a syntax error because of the missing parenthesis:

```
(if (> x 10)
    (display "x is greater than 10")
    (display "x is less than or equal to 10" ; Missing closing parenthesis
```

In Haskell, a syntax error would occur if, for example, you incorrectly use the `let` keyword without the necessary `in` clause:

```
let x = 10
x * 2  -- Syntax error: 'in' keyword is missing
```

3.3.2 Grammar

Grammar in a programming language refers to the formal set of rules that define how different elements (like expressions, statements, and declarations) can be combined to produce meaningful code. It describes how to generate valid expressions or statements by substituting symbols according to certain rules, often expressed using a formalism like Backus-Naur Form (BNF).

Production Rules (BNF Style): Grammar consists of production rules that describe how to form valid sentences (or expressions) in the language. For instance, a simple grammar rule in Haskell might state that an `expression` can be formed by combining a `term`, an operator (like `+` or `-`), and another `term`:

```
expression ::= term '+' term
```

This rule states that an `expression` is made by combining two `terms` with a `+` operator in between. In Racket, a similar production rule might describe how to form a valid function application:

```
function_application ::= '(' function_name argument_list ')'
```

This rule indicates that a function application starts with an open parenthesis, followed by the function name, an argument list, and a closing parenthesis.

Generating Valid Code: Grammar helps in generating or recognizing valid structures in a program. For example, it might describe how a valid function definition is formed in Haskell:

```
function_definition ::= 'f' '::' type_signature '\n' 'f' parameter '=' expression
```

This rule indicates that a function definition starts with the function name, followed by a type signature, and then the function body with parameters and an expression. In Racket, a similar rule might describe how a `define` statement is structured:

```
definition ::= '(define' variable expression ')'
```

3.3.3 Example to Illustrate Syntax vs. Grammar

Consider a simple expression in natural language: *"The cat runs."*

- **Syntax:** Syntax rules in English dictate that the sentence should follow a Subject-Verb-Object (SVO) structure. The words must be arranged in a way that fits this structure.
- **Grammar:** Grammar ensures that the subject (*"cat"*) and the verb (*"runs"*) agree in number, and that the verb is conjugated correctly for the subject.

In a programming language, consider the following Racket expression:

```
(if (> x 10)
    (display "x is greater than 10")
    (display "x is less than or equal to 10"))
```

- **Syntax:** The syntax rules dictate that the `if` expression must have a condition followed by two branches (true and false cases), all enclosed in parentheses.
- **Grammar:** The grammar rules ensure that the condition `> x 10` is a valid expression, meaning that `x` is a variable that can be compared using the `>` operator, and that the `display` function is called correctly within each branch.

In Haskell:

```
fact :: Int -> Int
fact 0 = 1
fact n = n * fact (n - 1)
```

- **Syntax:** The syntax rules dictate the structure of the function declaration and definition, ensuring that the type declaration (`fact :: Int -> Int`) and the function body are correctly formatted.
- **Grammar:** The grammar ensures that the pattern matching (`fact 0` and `fact n`) aligns correctly with the function definition and that the recursive call `fact (n - 1)` is meaningful.

Again, **Syntax** is about the structure and order of code elements—how you write the code. **Grammar** is about the rules for generating meaningful expressions and statements—what the code means and how valid expressions are formed. Another example in Python is indentation. An indentation error will be reported as a syntax error, but will not be part of any grammar rules.

3.3.4 Define Syntax and Grammar by Ourselves

Let's explore the process of deriving grammar rules, particularly for arithmetic expressions. Consider the following grammar to generate arithmetic expressions:

```
<expr> ::= NUMBER
         | ‘(’ <expr> <op> <expr> ‘)’

<op> ::= '+' | '-' | '*' | '/'
```

Explanation:

- The non-terminal symbols `<expr>` and `<op>` represent expressions and operators, respectively.
- The vertical bar `|` denotes alternatives, meaning that an expression can be a number or a parenthesized expression consisting of two sub-expressions separated by an operator.
- This grammar is recursive, as `<expr>` can be expanded into more instances of `<expr>`, allowing for nested expressions.

Consider the following simple program:

```
52
```

The corresponding grammar rules are:

```
<prog> ::= <expr>

<expr> ::= NUMBER
```

In this rule, `<prog>` needs to match an expression, and `<expr>` is defined as a number. This rule allows the program to recognize any numerical value as valid.

Let's expand the grammar to handle more complex expressions:

```
(6 * 8)
```

We will derive the necessary grammar rules by analyzing each component of this expression and generalizing the rules to cover similar cases.
The expression '(6 * 8)' is composed of:

- An opening parenthesis '(',
- A number '6',
- An operator '*',
- Another number '8',
- A closing parenthesis ')'.

From this, we observe that a valid expression consists of:

- A pair of parentheses enclosing two numbers separated by an operator.

Let's start by defining the basic components that we identified:

```
<expr> ::= '(' <expr> <op> <expr> ')'
<op>   ::= '+' | '-' | '*' | '/'
<expr> ::= NUMBER
```

This defines that:

- An expression '<expr>' can be either a number ('NUMBER') or a more complex expression involving two sub-expressions enclosed in parentheses and separated by an operator ('<op>').
- The operator '<op>' can be one of the basic arithmetic operators: addition ('+'), subtraction ('-'), multiplication ('*'), or division ('/').

To generalize this further based on the expression '(6 * 8)', the grammar can be extended to handle nested and more complex expressions. For instance:

```
<expr> ::= NUMBER
         | '(' <expr> <op> <expr> ')'

<op> ::= '+' | '-' | '*' | '/'
```

This means:

- An expression can be a simple 'NUMBER'.
- An expression can also be a parenthesized combination of two expressions separated by an operator.

Now, let's verify that a more complicated expression can be derived using the above grammar:

```
( (1 + 2) / (3 * 4) )
```

The grammar needs to handle nested expressions, which is achieved by the recursive nature of the `<expr>` rule. The derivation process for this expression is as follows:

1. The outermost expression is a division, so we use the rule:

   ```
   <expr> ::= '(' <expr> <op> <expr> ')'
   ```

   The two sub-expressions are '(1 + 2)' and '(3 * 4)'.

2. The first sub-expression '(1 + 2)' is derived as:

   ```
   <expr> ::= '(' NUMBER <op> NUMBER ')'
   ```

   Both '1' and '2' are numbers, and '+' is a valid operator.

3. The second sub-expression '(3 * 4)' is derived similarly:

   ```
   <expr> ::= '(' NUMBER <op> NUMBER ')'
   ```

   Both '3' and '4' are numbers, and '*' is a valid operator.

It shows that the recursive structure of the grammar allows it to handle arbitrarily nested expressions, making the language expressive and capable of representing complex arithmetic.

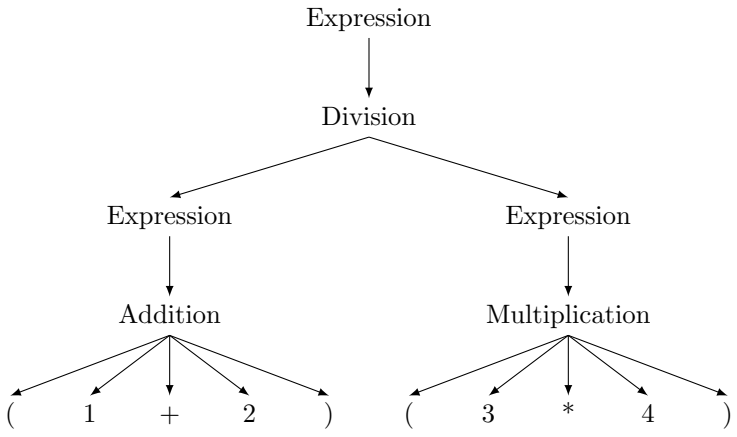
3.4 Parse Tree and Abstract Syntax Tree

Parsing is the process of analyzing a string of symbols according to the rules of a formal grammar. This process converts the linear sequence of tokens (the input) into a hierarchical structure, often represented as a parse tree or an abstract syntax tree (AST). Parsing ensures the code is syntactically correct according to the language's grammar before any further processing.

Consider the expression $(1 + 2)/(3 * 4)$. A parser would analyze this expression based on the grammar rules of the programming language, identifying components like numbers, operators, and parentheses, and structuring them according to precedence rules.

3.4.1 Parse Tree

A parse tree (or syntax tree) is a tree that represents the syntactic structure of a string according to some context-free grammar. Each node in the parse tree corresponds to a **grammar production**, and the structure of the tree mirrors the grammatical structure of the input.



3.4.2 Grammar Productions

In the context of formal languages, particularly in computer science and linguistics, a **grammar production** (or simply **production**) refers to a rule in a formal grammar that defines how a particular symbol or sequence of symbols (often called *non-terminals*) can be expanded or replaced by other symbols (which can be *terminals* or non-terminals).

A formal grammar consists of a set of such production rules, and these rules dictate how valid strings (sentences) in a language can be generated from a starting symbol, typically called the **start symbol**.

Non-terminal symbols are placeholders or variables that can be replaced by other non-terminal or terminal symbols based on the production rules. In our example, the non-terminal symbols are:

- **Expr**: Represents a complete arithmetic expression. It can be simple or composed of multiple terms added together.
- **Op**: Represents a valid operator.

Terminal symbols are the actual symbols of the language (e.g., specific characters or tokens like `+`, `*`, `num`, etc.) that appear in the final valid strings (expressions).

The **production rules** define how non-terminal symbols can be expanded or replaced by other symbols. For the arithmetic expression grammar, the production rules might look like this:

```
<expr> ::= NUMBER
        | ‘(’ <expr> <op> <expr> ’)’

<op> ::= '+' | '-' | '*' | '/'
```

When parsing or generating a string in the language defined by this grammar, the parser starts with the start symbol (in this case, `Expr`) and applies the production rules to replace non-terminal symbols with their corresponding definitions.

For example, if we want to generate the expression `((2 * 4) - (9 / 3))`, the derivation using the grammar productions could be (Leftmost derivation):

$$
\begin{aligned}
\langle expr \rangle &\Rightarrow (\ \langle expr \rangle\ \langle op \rangle\ \langle expr \rangle\ ) \\
&\Rightarrow (\ (\ \langle expr \rangle\ \langle op \rangle\ \langle expr \rangle\ )\ \langle op \rangle\ \langle expr \rangle\ ) \\
&\Rightarrow (\ (\ \texttt{NUMBER}\ \langle op \rangle\ \langle expr \rangle\ )\ \langle op \rangle\ \langle expr \rangle\ ) \\
&\Rightarrow (\ (\ \texttt{NUMBER}\ *\ \langle expr \rangle\ )\ \langle op \rangle\ \langle expr \rangle\ ) \\
&\Rightarrow (\ (\ \texttt{NUMBER}\ *\ \texttt{NUMBER}\ )\ \langle op \rangle\ \langle expr \rangle\ ) \\
&\Rightarrow (\ (\ \texttt{NUMBER}\ *\ \texttt{NUMBER}\ )\ -\ \langle expr \rangle\ ) \\
&\Rightarrow (\ (\ \texttt{NUMBER}\ *\ \texttt{NUMBER}\ )\ -\ (\ \langle expr \rangle\ \langle op \rangle\ \langle expr \rangle\ )\ ) \\
&\Rightarrow (\ (\ \texttt{NUMBER}\ *\ \texttt{NUMBER}\ )\ -\ (\ \texttt{NUMBER}\ \langle op \rangle\ \langle expr \rangle\ )\ ) \\
&\Rightarrow (\ (\ \texttt{NUMBER}\ *\ \texttt{NUMBER}\ )\ -\ (\ \texttt{NUMBER}\ /\ \langle expr \rangle\ )\ ) \\
&\Rightarrow (\ (\ \texttt{NUMBER}\ *\ \texttt{NUMBER}\ )\ -\ (\ \texttt{NUMBER}\ /\ \texttt{NUMBER}\ )\ )
\end{aligned}
$$

$$
\texttt{NUMBER} \mapsto 2,\ 4,\ 9,\ 3 \quad \Rightarrow \quad ((2 * 4) - (9/3)).
$$

At each step, we apply a production rule to replace a non-terminal with other symbols according to the grammar. The final result, `num * (num + num)`, can then be instantiated with actual numbers (e.g., `((2 * 4) - (9 / 3))`).

Another example to illustrate the relationship between terminal and non-terminal symbols. Consider the following grammar:

greeting → ’hi’ — ’hello’ — ’howdy’
person → name surname
name → ’john’ — ’jane’
surname → ’doe’ — ’smith’
sentence → greeting person

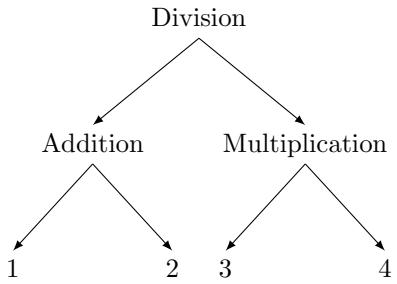
In this grammar:

- **Non-terminals:** `greeting, person, name, surname, sentence`
- **Terminals:** `’hi’, ’hello’, ’howdy’, ’john’, ’jane’, ’doe’, ’smith’`

Parsing an input like `hi john doe` would validate as true, as it matches the `sentence` production.

3.4.3 Abstract Syntax Tree (AST)

An abstract syntax tree is a simplified, abstracted version of the parse tree. Unlike the parse tree, the AST does not include every syntactic detail. It focuses on the structural and semantic meaning of the code. For example, it omits parentheses as their function is implied by the tree structure itself.



In this AST,

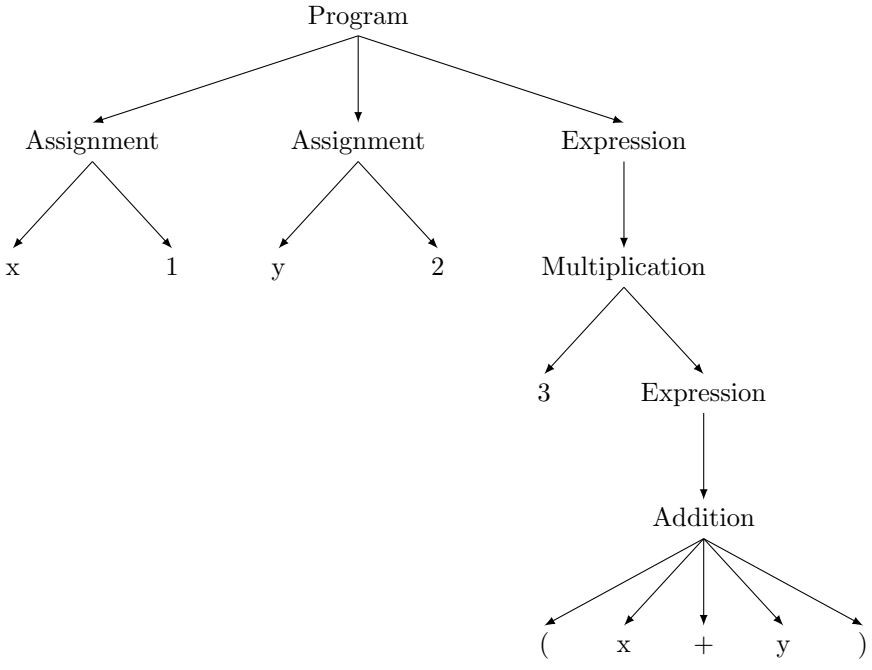
- AST nodes represent operators and operands.
- Non-terminals like $< expr >$ and $< op >$ don't appear; only operators and values remain.

3.4.4 PT vs AST: An Example

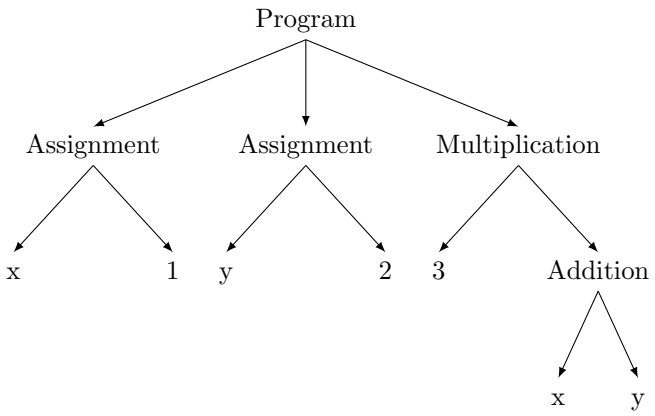
Both are very useful data structures used in compilers or interpreters. PT represents the full syntactic structure of the source code based on the grammar of the language. It is an important component of the parser of a compiler. it does syntax error detection: It shows every detail of the grammar, so it's ideal for checking whether the code conforms to the language's syntax rules.
Abstract Syntax Tree (AST) is a simplified, abstracted version of the parse tree that omits unnecessary syntactic details (like parentheses or semicolons), which is a foundation for Semantic analysis inside the compiler: Type checking, name binding (resolving variable/function names), scope resolution. And it is also very useful for later procedures such as optimization and code generation.
Consider a program:

```
x=1
y=2
3*(x+y)
```

Parse Tree



Abstract Syntax Tree (AST)



3.5 Semantics

Parsing a program into an Abstract Syntax Tree (AST) gives us its syntactic structure but not its meaning. The **semantics** of a programming language define the rules that govern the meaning of programs written in that language.

For imperative-style programs, the semantics must describe not only individual expressions, such as `3 + 5`, but also constructs like `return` (which interrupts control flow) and `for` (which iterates through a specified range).

In expression-based languages, running a program means evaluating an expression, and the program's semantics are determined by the value the expression produces upon evaluation. The semantics of such languages, therefore, relate to the value that the expression or program returns to the user.

3.5.1 Denotational Semantics

Denotational semantics specify the "value" or "output" of a program. Here are several Python expressions that all have the same denotational value, 10:

```
10
3 + 7
1 + 3 ** 2
(lambda x: x + 3)(7)
list(range(50000000))[11]
```

Although each of these expressions is written differently, they all evaluate to the same result, meaning they have the same mathematical value.

3.5.2 Operational Semantics

While expressions might have the same denotational value, the steps they take to reach that value can differ significantly. The **operational semantics** of a programming language specify these evaluation steps.

In imperative languages, operational semantics are complex because they must account for control flow, mutations, and function calls. Conversely, specifying the operational semantics of expression evaluation—especially in functional contexts—is generally more straightforward.

```
def find_even(numbers):
    for num in numbers:
        if num % 2 == 0:
            return num
    return None

result = find_even([1, 3, 7, 10, 15])
print(result)  # Output: 10
```

Denotational Semantics: The program's value is the first even number found in the list, or `None` if no even number exists. Here, the denotational value of the program when given the input `[1, 3, 7, 10, 15]` is `10`.

Operational Semantics:

- The program iterates over the list `[1, 3, 7, 10, 15]`.

- For each number, it checks if it is even (`num % 2 == 0`).
- Upon finding `10`, it immediately returns this value and exits the function.
- The operational steps involve looping, checking conditions, and potentially breaking out of the loop when the condition is met.

```
(define (find-even numbers)
  (cond
    [(null? numbers) #f]
    [(even? (car numbers)) (car numbers)]
    [else (find-even (cdr numbers))]))

(find-even '(1 3 7 10 15))  ; Output: 10
```

Denotational Semantics: The denotational value of the program is the first even number in the list, or `#f` if no even number is found. For the input `'(1 3 7 10 15)`, the denotational value is `10`.

Operational Semantics:

- The function evaluates the list by checking if it is empty (`null? numbers`).
- If the list is not empty, it checks if the first element (`car numbers`) is even.
- If the first element is even, it returns that element.
- Otherwise, it recursively processes the rest of the list (`cdr numbers`) until it finds an even number or reaches the end of the list.

3.6 Lambda Calculus

In contrast to this instruction-based approach, Alonzo Church developed a model known as the lambda calculus, where expressions themselves are the core, and indeed the sole, units of computation. Instead of viewing a program as a sequence of statements, in lambda calculus, a program is represented as a single expression, which may encompass numerous subexpressions. When we refer to a computer running a program in this context, we mean that it evaluates this single expression, not that it performs operations corresponding to various statements.

3.6.1 Syntax and Grammar of the Lambda Calculus

Here is a complete BNF grammar for the lambda calculus:

```
<lambda-exp>::= <var>
        | lambda<var> . <lambda-exp>
        | ( <lambda-exp> <lambda-exp> )
```

Based on the definition above, in lambda calculus, an expression can be one of the following three forms:

1. A variable (the first production above): typically, we will use a single letter, with an optional integer subscript, to denote a variable.
2. A function abstraction (the second production above): this type of lambda expressions, also called a lambda abstraction, corresponds to a function definition, which contains two components: the formal parameter of the function and the body of the function.
3. An application (the third production above): this type of lambda expressions corresponds to a function call (or application, or invocation), which contains two components: the function being called, followed by the argument that is passed into the function.

3.6.2 Semantics of the Lambda Calculus

A variable in the lambda calculus (the first rule in the lambda calculus grammar) is a placeholder for another lambda expression. As in all programming languages, a variable can refer to a value that may or may not be known at the time. For instance, variables x and p_1 in lambda calculus can be represented by variables `x` and `p1`, respectively, in Python.
In lambda calculus, unlike in Python, each variable can only be bound to a single value throughout the execution of the entire program. In contrast, in Python, the value of a variable can be changed multiple times using assignment

Lambda Expression	English Semantics	Python Implementation
x	The variable named x	`x`

Table 1: Semantics of Variables in Lambda Calculus

statements. Thus, variables in lambda calculus are more akin to named constants than variables in imperative languages. Additionally, since the only values in lambda calculus are functions, all variables are placeholders for function values.

Lambda Abstractions. A lambda abstraction in lambda calculus (the second rule in the grammar) defines a function. It is an expression that defines a function rather than a function call. Since all functions in lambda calculus are anonymous and take exactly one parameter, defining a function requires only the name of its parameter (i.e., the variable following the λ) and its body (a lambda expression).

Lambda Expression	English Semantics & Python Implementation
$\lambda x.x$	The function of x that returns x (identity function) `lambda x: x`
$\lambda x.y$	A constant function (of x) that returns y `lambda x: y`
$\lambda x.\lambda y.y$	The function of x that returns a function of y that returns y (a curried function) `lambda x: (lambda y: y)`

Table 2: Semantics of Lambda Abstractions in Lambda Calculus

Function Applications. A function application in lambda calculus (the third rule in the grammar) represents a function call, where an expression invokes a function with a single argument. The first part of a function application can either be a variable or a more complex lambda expression that evaluates to a function.

Lambda Expression	English Semantics & Python Implementation
$(\lambda x.x\, y)$	The identity function applied to y `(lambda x: x)(y)`
$(\lambda z.x\, y)$	The constant function x applied to y `(lambda z: x)(y)`
$(\lambda x.\lambda y.y\, z)$	The curried function applied to z, returning the identity function `(lambda x: (lambda y: y))(z)`
$((\lambda x.\lambda y.y\, u)\, v)$	The curried function applied to u and v, returning the value of the second argument v `(lambda x: (lambda y: y))(u)(v)`

Table 3: Semantics of Function Applications in Lambda Calculus

3.6.3 Free and Bound Variables

Lambda calculus, like Python and most other modern programming languages, uses *static binding* (also known as *static scoping* or *lexical binding*). This means that each variable use is bound to the closest variable declaration with the same name in the smallest lambda abstraction that contains that use. We can now discuss the concepts of variable declaration and use, binding occurrence, free and bound variables, and lexical scoping in the context of lambda calculus. Consider the following example:

$$\lambda y.(\lambda x.x\,(y\,x))$$

This lambda expression is an abstraction with parameter `y` and a body that applies the identity function to the expression $(y\,x)$. The `y` following the λ is the *binding occurrence* of the variable `y`. The scope of this declaration is the expression $\lambda x.x\,(y\,x)$, indicating that the rightmost occurrence of `y` is bound to the leftmost binding occurrence. In contrast, the scope of the binding occurrence of `x` in $\lambda x.x$ is only the second `x` (that is, as always, the body of the lambda abstraction). As a result, the third, rightmost occurrence of `x` in the expression is free: it is a use of `x` that does not lie within the scope of any declaration of `x`.

To summarize this example, from left to right, the first occurrence of `x` is its binding occurrence, the second occurrence is bound to the first one, and the third occurrence is free. This example illustrates that a variable can occur both free and bound within the same expression. Therefore, it may be misleading to ask whether a variable is free or bound in a lambda expression; it is more precise to ask this question about each specific occurrence of a variable. It is important to note that a binding occurrence is never free, as its purpose is to define a new variable.

Formal definition for free variable in lambda: a variable x is said to occur free in any lambda expression E if and only if:

1. E is a variable, and E is identical to x, or
2. E is of the form $(E_1\, E_2)$, and x occurs free in either E_1 or E_2 (or both), or
3. E is of the form $\lambda y.E'$, where y is different from x and x occurs free in E'.

The table 3.6.3 below illustrates all cases of this definition.

Expression E	**Case**	**Explanation**
x	1	yes, because x appears in (is equal to) E and E does not contain any binding occurrences (no λ).
$(x\, y)$	2	yes, because x occurs free in the first component of the function application (recursive application of case 1).
$(y\, x)$	2	yes, because x occurs free in the second component of the function application (recursive application of case 1).
$\lambda z.x$	3	yes, because x is different from z (the parameter of the lambda abstraction) and x occurs free in the body of the lambda abstraction (recursive application of case 1).
$\lambda z.\lambda x.x$	3	no, because x is different from z (the parameter of the lambda abstraction), but x does not occur free in the body of the lambda abstraction (recursive application of case 3). The body in this case is the lambda abstraction $\lambda x.x$.
$\lambda x.y$ or $\lambda x.x$	3	no, because x is identical to the parameter of the lambda abstraction E. x cannot be free in E since any free occurrences of x in the body of E would be bound by the leading binding occurrence of x.

Table 4: Cases of Free and Bound Variables in Lambda Calculus

Example 1:

$$(\lambda v.(v * (x - v)))(\lambda x.(x + z))$$

The first lambda function $(\lambda v.(v * (x - v)))$ is applied to the second lambda function $(\lambda x.(x + z))$. The application implies substituting v in the first lambda's body with $(\lambda x.(x + z))$, although it may not lead to meaningful outcomes as lambda functions are not typical numeric expressions.

Free and Bound Variables

- **Bound Variables:** v in its scope; x in the second lambda.
- **Free Variables:** x in the first lambda's scope, z in the entire expression.

Example 2:

$$\lambda y.(\lambda x.x\ (y\ x))$$

This lambda expression is a lambda abstraction whose parameter is y and whose body is the application of the identity function to the expression (y x).

Variables Analysis

- **Bound Variables:** Therefore, the y after the λ is the binding occurrence of the variable y. The scope of this declaration is (λx.x (y x)), which implies that the rightmost occurrence of y is bound to the leftmost binding occurrence. In contrast, the scope of the binding occurrence of x in λx.x is just the second x in it (that is, as always, the body of the lambda abstraction). .
- **Free Variables:** the third, rightmost occurrence of x in the expression above is free: it is a use of x that does not belong to the scope of any declarations of x

3.6.4 The Essence of Computation in Lambda Calculus

The key takeaway from this model is that function application (via substitution) is the sole mechanism for inducing computation. Functions can be created using λ and applied to values or even other functions, allowing us to construct complex computations by combining functions. It's crucial to note that functions in lambda calculus are much more restrictive than those typically encountered in programming. The only operation permitted when evaluating a function application is the substitution of arguments into the function body, followed by the evaluation of that body to produce a single value. These functions do not involve any concept of time or sequence of instructions, nor do they rely on external or global states to influence their behavior.

At this stage, lambda calculus might seem like an abstract mathematical concept. You may wonder what it means to treat everything as a function, especially when we also care about entities that aren't functions, like numbers, strings, classes, and various data structures you've learned about so far. However, since the Turing machine and lambda calculus are equivalent models of computation, anything achievable in one can also be accomplished in the other! Yes, we can represent numbers, strings, and data structures using functions. Although we will only touch on this briefly in this course, it is indeed possible. Despite the Turing machine's widespread use, the essence of lambda calculus remains relevant and vibrant, and understanding it will make you a better computer scientist.

Abstraction Definition:

$$\lambda\texttt{variable.expression}$$

- **Variable:** x
- **Expression (Function Body):** $x + 2$

Example:

$$\lambda x.(x + 2)$$

This abstraction defines a function that takes one parameter x and returns $x + 2$.

Application of Abstraction:

$$(\lambda\texttt{variable.expression})\ \texttt{argument}$$

- **Variable:** x
- **Expression (Function Body):** $x + 2$
- **Argument:** 3

Example:

$$((\lambda x.(x + 2))\ 3)$$

Evaluation Process:

1. Substitute 3 for x in the function body $(x + 2)$.

$$3 + 2$$

2. Simplify the expression to obtain the result.

$$5$$

Thus, the result of the application is 5.

3.6.5 Determine β-Redex

A λ expression that allows for evaluation rule to be applied is known as a β redex (short for β-reduction expression). More formally, a β-redex is defined as a λ expression that takes a specific form: an application where the first term is a function abstraction. It has the form:

$$(\lambda x.E)\, A$$

where:

- $\lambda x.E$ is a lambda abstraction, representing a function with parameter x and body E.

- A is an argument applied to this lambda abstraction.

For instance, the expression:

$$(\lambda x.(x\,v)\,(z\,(v\,u)))$$

is a β-redex because the first term, $\lambda x.(xv)$, is a function abstraction. Conversely, the expression:

$$((\lambda x.(x\,v)\,y)\,(z\,(v\,u)))$$

is not a β-redex since its first term is a function application, not a function abstraction. However, the function application within the first term is itself a β-redex. This example illustrates that expressions that are not β-redexes may still contain sub-expressions that qualify as β-redexes.

3.6.6 Evaluation by Substitution (β-Reduction)

Beta reduction is the process of applying a lambda function to an argument by substituting the argument for the parameter in the function body. Given a lambda expression $(\lambda x.\text{body})\ y$, where $\lambda x.\text{body}$ is the function and y is the argument, beta reduction involves replacing every occurrence of x in body with y.

Formally, if you have an expression of the form $(\lambda x.E)\ A$, where E is an expression (the function body) and A is the argument, beta reduction is defined as:

$$(\lambda x.E)\ A \to E[A/x]$$

Here, $E[A/x]$ means that in the expression E, every instance of x is replaced by A.

Normal order is an evaluation strategy that always reduces the left-most outer-most redex first (is the leftmost redex not contained in any other redex.). You can think about normal-order evaluation as traversing the lambda expression and evaluating every function you find before evaluating any function arguments. This strategy is also known as **lazy evaluation**.

Applicative order is an evaluation strategy that reduces the left-most inner-most redex first (is the leftmost redex not containing any other redex), or say internal reductions are applied first, and only after all internal reductions are complete, the left-most redex is reduced. More formally, we evaluate the left-most redex free of any internal redexes. This strategy is also known as **eager evaluation**.
Example 1:

$$(\lambda x.y)((\lambda z.z + z)3)$$

Evaluated using Normal Order:

1. Evaluate the outermost function $(\lambda x.y)$ first, substituting $((\lambda z.z + z)3)$ for x, Since the body of the function is just y, and x does not appear in it, the result is directly y, without evaluating the argument $((\lambda z.z + z)3)$:

$$(\lambda x.y)((\lambda z.z + z)3) \to y$$

2. The result is y.

Evaluated using Applicative Order:

1. Evaluate the innermost expression $(\lambda z.z + z)3$ first:

$$(\lambda z.z + z)3 \to 3 + 3 \to 6$$

2. Substitute the evaluated argument 6 into the outer function $(\lambda x.y)$:

$$(\lambda x.y)6 \to y$$

3. The result is y.

In this example, both normal order and applicative order yield the same result y. However, in normal order, the argument $((\lambda z.z + z)3)$ is never evaluated because it is not needed, whereas in applicative order, the argument is fully evaluated before substitution.

Example 2:

$$((\lambda m.\lambda n.\lambda o.mno)(\lambda x.xx)(\lambda x.x)m)$$

Evaluated using Normal Order:

1. Substitute the first argument ($\lambda x.xx$) for m in the outer function:

$$\lambda n.\lambda o.(\lambda x.xx)no$$

2. Substitute the second argument ($\lambda x.x$) for n:

$$(\lambda o.(\lambda x.xx)(\lambda x.x)o)m$$

3. Substitute the third argument m for o in the expression:

$$(\lambda o.(\lambda x.xx)(\lambda x.x)o)m = (\lambda x.xx)(\lambda x.x)m$$

   The expression simplifies to:

$$(\lambda x.x)(\lambda x.x)m$$

4.

$$(\lambda x.x)m = m$$

5. The result is m.

The Church-Rosser Theorem asserts that **if two different reduction strategies both lead to an answer, then they will result in the same answer**.

Let's look at a counterexample:

$$(\lambda x.m)((\lambda x.xx)(\lambda x.xx))$$

Evaluated using Normal Order:

1. Apply the outermost function:

$$(\lambda x.m)((\lambda x.xx)(\lambda x.xx)) \rightarrow m$$

2. The result is m, and no further reduction is needed as m does not depend on any further computation.

Evaluated using Applicative Order:

1. Evaluate the innermost expression:

$$(\lambda x.xx)(\lambda x.xx)$$

2. Substitute x with $\lambda x.xx$ in xx:

$$(\lambda x.xx)(\lambda x.xx) \rightarrow (\lambda x.xx)(\lambda x.xx)$$

3. This leads to an **infinite loop** as the expression does not reduce and keeps calling itself.

Another example:

$$(\lambda a.\lambda b.\lambda c.a(bc))(\lambda x.x + 2)(\lambda y.y - 1)3$$

Normal order evaluation evaluates the outermost function first and delays evaluation of function arguments until their values are needed.

1. We start by substituting $\lambda x.x + 2$ for a in the expression $\lambda a.\lambda b.\lambda c.a(bc)$:

$$(\lambda a.\lambda b.\lambda c.a(bc))(\lambda x.x + 2)(\lambda y.y - 1)3 \rightarrow (\lambda b.\lambda c.(\lambda x.x + 2)(bc))(\lambda y.y - 1)3$$

2. Next, substitute $\lambda y.y - 1$ for b in $(\lambda b.\lambda c.(\lambda x.x + 2)(bc))$:

$$(\lambda b.\lambda c.(\lambda x.x + 2)(bc))(\lambda y.y - 1)3 \rightarrow (\lambda c.(\lambda x.x + 2)((\lambda y.y - 1)c))3$$

3. Now, substitute 3 for c:

$$(\lambda c.(\lambda x.x + 2)((\lambda y.y - 1)c))3 \rightarrow (\lambda x.x + 2)((\lambda y.y - 1)3)$$

4. Evaluate the function $(\lambda x.x + 2)$:

$$(((\lambda y.y - 1)3) + 2)$$

5. Finally,evaluate $(\lambda y.y - 1)3$:

$$2 + 2 = 4$$

Applicative order evaluation evaluates the innermost function first, evaluating all arguments before applying them to their functions.

1. Evaluate $\lambda x.x + 2$ and $\lambda y.y - 1$ (which are already in their simplest form), then apply them:

$$(\lambda a.\lambda b.\lambda c.a(bc))(\lambda x.x + 2)(\lambda y.y - 1))3 \rightarrow (\lambda c.(\lambda x.x + 2)((\lambda y.y - 1)c)3$$

2.
$$(\lambda c.(\lambda x.x + 2)((\lambda y.y - 1)c))3 \rightarrow (\lambda x.x + 2)((\lambda y.y - 1)3)$$

3.
$$(\lambda y.y - 1)3 \rightarrow 3 - 1 = 2$$

4.
$$(\lambda x.x + 2)2 \rightarrow 2 + 2 = 4$$

Let's implement this lambda calculus in Racket and Haskell:
Basic Syntax:

- (lambda (parameters) expression)
- lambda → introduces an anonymous function
- (parameters) → list of arguments
- expression → the body (can be multiple expressions, the last one is returned)

```
#lang racket
((((lambda (a)
    (lambda (b)
      (lambda (c)
        (a (b c)))))
  (lambda (x) (+ x 2)))
 (lambda (y) (- y 1)))
3)
```

```
main :: IO ()
main = print $
  ((\a -> \b -> \c -> a (b c))
   (\x -> x + 2)
   (\y -> y - 1))
  3
```

3.6.7 Some Lambda Expressions in Python

```
f = lambda x : (lambda y : x + y)
g = f(5)
result = g(3) + f(3)(4)
print(result)  # 15

f = lambda x : (lambda y : (lambda z : z))
result = f(10)(20)(30)
print(result)  # 30
```

4 Functions Are Essential

Our goal in this chapter is to expose you to some of the central concepts in programming language theory, and functional programming in particular, without being constrained to one particular language. So in this chapter, we'll draw on examples from three languages: Racket, Haskell, and Python. Our hope here is that by studying the similarities and differences between these languages, you'll gain more insights into the deep concepts in this chapter than by studying any one of these languages alone.

4.1 Function Expressions

Let's explore one of the core concepts in functional programming: functions as first-class values. This means that functions can be handled and manipulated just like any other value in a program. Being "first-class" means that functions in Haskell can be:

- Assigned to variables: Functions can be bound to names and passed around in the same way as other values.
- Passed as arguments to other functions: This allows for higher-order functions, which are functions that take other functions as parameters.
- Returned as results from functions: Functions can also be the return values of other functions.
- Stored in data structures: Since functions are just values, they can be stored in lists, tuples, and other data structures.

This is a significant concept! Many programming languages do not support functions as first-class values, which limits what you can do with functions compared to other values. For instance, consider how we represent a standalone function value in a language.

If we want to represent the integer value three in a program, it's straightforward: simply write the numeric literal `3`. But what if we want to represent a function that adds 3 to a number? In Python, you might write:

```
def f(x):
    return x + 3
```

However, there's a key difference: the numeric value `3` stands alone, while the function is tied to the name `f`. In some programming languages, you can only define functions together with an identifier that refers to them. But if functions are truly first-class values, we should be able to write them independently of any name binding. A function value that is written without an identifier is called an **anonymous function**.

In the lambda calculus, all functions are anonymous: for example, $(\lambda x.x)$ represents a function value without an associated name. Racket, Haskell, and Python all support anonymous functions, each using syntax inspired in different ways by the lambda calculus.

```
; Racket
;(lambda (<param> ...) <body>)

(lambda (x y) (× 2 (+ x y)))
```

```
-- Haskell
-- \<param> ... -> <body>

\x y -> 3 * (x - y)
```

```
# Python
# lambda <param> ... : <body>

lambda x, y: x * (y ** 2)
```

In each of the examples above, `<param>` represents a parameter of the function and must be an identifier, while `<body>` is an expression that forms the body of the function.

Calling functions in all three languages—Python, Racket, and Haskell—is relatively straightforward, though it's important to note that both Racket and Haskell have some syntax that may be unfamiliar.

In Python, a function call is similar to what you've likely seen in other languages:

```
#<function>(<arg>, ...)
def multiply(x, y):
    return x * y

multiply(3, 4)
(lambda x, y: x * (y ** 2))(3, 4)
```

In Racket, the function expression is placed inside the parentheses, following what is known as Polish prefix notation:

```
;(<function> <arg> ...)
(define (subtract x y)
  (- x y))

(subtract 10 4)
((lambda (x y) (× 2 (+ x y))) 3 4)
```

One common pitfall for students learning Racket is the treatment of every parenthesized expression as a function call unless it begins with a keyword like `lambda`. This differs significantly from most programming languages, where parentheses are often used redundantly to clarify expression grouping. **In Racket, however, parentheses are never redundant**.

In Haskell, function calls are even more concise, as parentheses are not required at all. Any two expressions separated by a space are interpreted as a function call:

```
-- <function> <arg> ...
add :: Int -> Int -> Int
add x y = x + y

add 7 3

(\x y -> 3 * (x - y)) 7 4
```

This syntax can take some getting used to, especially if you come from a background where function calls are always enclosed in parentheses. However, once familiar, it can make the code more concise and readable.

4.2 Operators as Functions

Consider common binary arithmetic operations like addition and multiplication, which we typically write in infix form, meaning the operator is placed between its two arguments—such as `3 + 4` or `1.5 * 10`. In line with the importance of functions in programming, it's crucial to understand that these operators are actually functions, at least from a mathematical perspective. In fact, Racket, Haskell, and Python all treat these operators as functions.
In Racket, operators are simply identifiers that reference built-in functions and are invoked in the same way as any other function:

```
> (+ 10 20)
30
> (× 3 5)
15
```

In Python, operators work by calling underlying "dunder methods" in built-in classes:

```
>>> 10 + 20
30
>>> int.__add__(10, 20)  # Equivalent to 10 + 20
30
```

Haskell follows a similar approach. Every infix operator (such as `+`) is a function whose name corresponds to the operator, but when used in prefix notation, the operator must be enclosed in parentheses:

```
> 10 + 20
30
```

```
> (+) 10 20
30
```

At first glance, this might seem unusual or overly complex if you haven't encountered it before. Racket, with its consistent use of prefix notation for all functions (and no infix operators), presents the most uniform model, though it may seem less familiar. Python and Haskell, on the other hand, introduce a level of indirection to support the more familiar infix syntax we learn in mathematics.

4.3 Function Purity

One key aspect of our grammar rules for defining functions is that a function body consists of just a single expression. This stems from our model of an expression-based language, rather than the imperative approach commonly seen in other languages, which typically involves a sequence of statements.

The best way to conceptualize functions in this programming style is to draw an analogy to mathematical functions, such as $f(x) = x + 1$, which are purely defined by their input-output relationships. In this context, we describe the body of f as the expression $x + 1$, and we evaluate calls to f by substituting specific values for x, evaluating the expression, and returning the result.

According to the function definition rules we've discussed, all function values behave exactly like mathematical functions: their behavior is entirely determined by the evaluation of the body expression for a given set of arguments. For example, consider the Racket function $(\lambda(x)(+\ x\ 1))$. If we call this function with the number 10, we evaluate this function call by substituting 10 for x in the expression $(+\ x\ 1)$, resulting in 11 as the returned value. Notice that a return keyword is unnecessary; given the lambda expression, we know that the value of $(+\ x\ 1)$ (with the appropriate substitution) will be returned.

In programming, a function is considered (mathematically) **pure** if it satisfies the following properties:

1. **The function's behavior is entirely determined by its inputs.** For instance, the function cannot access data from outside its inputs, such as standard input, the file system, or the Internet. This also excludes randomized functions; pure functions must be deterministic.
2. **The function only returns a value and performs no other actions.** In parallel with the first property, this means that pure functions cannot produce side effects, such as printing to standard output, writing to the file system, or sending data over the Internet. We say that "pure functions have no side effects."

```python
external_value = 10

def f1(x, y): #pure
    return x + y

def f2(x): # impure
    return x * external_value

def f3(x): # impure
    print("Processing {x}".format(x))
    return x * 2
```

4.4 Name Bindings

Alonzo Church and Alan Turing demonstrated that anonymous functions and function application alone are sufficient to perform all computations that modern programming languages can achieve. However, even purely functional programming languages like Haskell provide additional conveniences for the programmer. We've previously discussed some of these conveniences, such as primitive values and built-in functions. The following example illustrates another:

```
(((lambda (x)
   ((lambda (f)
      (lambda (n)
         (if (equal? n 0) 1 (× n (f (- n 1))))))
    (lambda (v) ((x x) v))))
 (lambda (x)
   ((lambda (f)
```

```
      (lambda (n)
         (if (equal? n 0) 1 (× n (f (- n 1))))))
    (lambda (v) ((x x) v)))))
 10)
```

This complex program evaluates to 3,628,800, which is 10!, demonstrating the power of anonymous functions to implement recursive processes. However, we certainly don't want to be writing code like this regularly! Instead, every programming language allows us to bind identifiers to values, so that evaluating an identifier yields the value it is associated with.
You've already encountered one type of identifier: the **formal parameters** (such as x, f, n used in the above complex code) in function definitions. These are a special kind of identifier, bound to values when the function is invoked. In this section, we'll explore the more general use of identifiers. Identifiers can serve two primary purposes:

1. **To "save" the value of subexpressions for later reference.** This is primarily for programmer convenience
2. **To refer to a function's name within its own body, enabling recursive definitions.** This is essential for writing recursive functions.

It's important to remember that the use of identifiers beyond parameter names is mainly for convenience, making programs easier to understand, but not increasing the computational power of lambda calculus. Unlike in imperative programming languages, where identifier bindings can be mutable, **in pure functional programming, bindings are immutable**: once an identifier is bound to a value, it cannot be re-bound. This concept is closely related to **referential transparency**.

An identifier is considered referentially transparent if it can be substituted with its value in the source code without altering the program's meaning. This mirrors mathematical principles; when we define $x = 5$, any subsequent statement involving x should still make sense if we replace x with 5.

This approach to identifiers in functional programming is significantly different from what we are accustomed to in imperative programming, where re-binding names is not only allowed but often necessary, especially in constructs like loops or mutable data structures. Given that re-binding and mutation feel so natural, why would we choose to avoid them? Or, more specifically, why is referential transparency (which mutation violates) so valuable?

While mutation is a powerful tool, it **complicates our code by requiring us to constantly track the "current value" of each identifier throughout the program's execution**. Referential transparency, on the other hand, allows us to use names and values interchangeably, making reasoning about code much simpler. **Once a name is defined, it has the same meaning for the rest of its existence in the program.**

```python
import random
# Global variable
count = 0

def square(x):
    return x * x

# Example usage
result = square(5)  # Outputs: 25

# Referential transparency:
# The expression square(5) can be replaced with 25 without affecting the program
another_result = 25 + 10  # Outputs: 35

def increment_and_get_count():
    global count
    count += 1
    return count

# Example usage
first_call = increment_and_get_count()  # Outputs: 1
second_call = increment_and_get_count()  # Outputs: 2

# Non-referential transparency:
# The expression increment_and_get_count() cannot be replaced with 1 or 2 without changing the program's behavior

```

```python
def random_number():
    return random.randint(1, 100)

# Example usage
first_call = random_number()  # Could output any number between 1 and 100
second_call = random_number()  # Could output any number between 1 and 100

# Non-referential transparency:
# The expression random_number() cannot be replaced with a fixed number without changing the program's behavior
```

Advantages of Referential Transparency:

- **Memory Lookup**. In a referentially transparent function, given the same input, the output will always be the same. This allows the program to store the result of a computation (known as memoization) and reuse it later without needing to recompute it.
- **Parallelization**. Referentially transparent functions do not rely on or alter shared state, different parts of a program can be executed simultaneously without the risk of race conditions or inconsistent data. Each function operates independently, based solely on its input, and produces a predictable output.
- **Easy Reasoning and Testing**. Since functions are deterministic—always producing the same output for the same input—developers can predict the behavior of their programs with confidence. This predictability means that developers can substitute expressions with their evaluated results without changing the program's meaning, simplifying both reasoning and debugging.

4.4.1 Global (Top-Level) Name Bindings

In Racket, global name bindings are created using the keyword `define`:

```
(define <id> <expr>)
```

In Haskell and Python, global bindings are established using the familiar `=` symbol:

```
<id> = <expr>
```

These definitions associate the value of `<expr>` with the identifier `<id>`. Below are some examples of these bindings, including cases where a name is bound to a function:

Racket

```racket
(define a 3)
(define add-three
  (lambda (x) (+ x 3)))

(define (add-three x) (+ x 3))
(define (almost-equal x y) (<= (abs (- x y)) 0.001))
```

Haskell

```haskell
a = 3
addThree = \x -> x + 3

addThree x = x + 3

almostEqual x y = abs (x - y) <= 0.001
```

Python

```python
a = 3
add_three = lambda x: x + 3
```

4.4.2 Local Bindings

Most programming languages support both local and global scopes, with one of the most common examples being the local scope within functions. For instance, function parameters are local to the function's body:

```
(define (f x)
  (+ x 10)) ; can refer to x here in the body of the function

(+ x 10) ; can't refer to x out here (try it!)
```

In Racket, you can explicitly create a local scope using the `let*` keyword:

```
;(let× ([<id> <expr>] ...)
;  <body>)

(define (rectangle-properties length width)
  (let× ([area (× length width)]
         [perimeter (+ (× 2 length) (× 2 width))])
    (list area perimeter)))
```

A `let*` expression binds each expression to its corresponding identifier and evaluates the `<body>` expression using these bindings. The value of the `let*` expression is the value of `<body>`.
In Haskell, a similar local scoping is achieved using the `let` keyword:

```haskell
-- let <id> = <expr>
--     ...
-- in
--     <body>

rectangleProperties :: Double -> Double -> (Double, Double)
rectangleProperties length width =
    let area = length * width
        perimeter = 2 * (length + width)
    in (area, perimeter)
```

What about Python? Python also supports local scope, such as within function and class definitions, but it doesn't have specific syntax for `let` expressions—that is, an expression that involves local bindings and evaluates to produce a value. This difference highlights how Python is more statement-oriented, with assignment statements being the most common, whereas Racket and Haskell are more expression-oriented.

4.5 Lists

The list data type is one of the most fundamental in programming languages, serving as a flexible, variable-length compound data structure for storing any number of values. For many of us, lists represent the first opportunity to write code that can handle an arbitrary amount of data without knowing the exact size in advance.

In imperative languages, lists are typically processed using loops, which rely on the concepts of time and state to keep track of the "current" element being processed. This approach treats a list as a linear sequence of items, where each item is processed one at a time. However, in pure functional programming, where mutation is not allowed, and we cannot re-bind an identifier to each element of the list sequentially, a different approach is required.

The key insight is to recognize the recursive structure inherent in the list data type and use this structure to inform both the representation and operations on lists. A list can be recursively defined as follows:

- **The empty list is a list.** (Represented as `'()` in Racket, and `[]` in Haskell and Python.)
- **If `x` is a value and `lst` is a list, then we can construct a new list where `x` is the first element, and the remaining elements are those from `lst`.** This operation is known as `cons`, which stands for "construct". In Racket, this is represented by the `cons` function, and in Haskell by the infix operator `(:)`.

For example, the list `[1, 2, 3]` can be constructed using this recursive definition as `(cons 1 (cons 2 (cons 3 '())))` in Racket, and `1:2:3:[]` in Haskell.

In both Racket and Haskell, non-empty lists can be deconstructed into their first element and the rest of the list using functions like `first` and `rest` in Racket, or `head` and `tail` in Haskell. This deconstruction obeys the identity `lst = (cons (first lst) (rest lst))` for all non-empty lists `lst`.

4.6 From Iteration to Recursion

Iteration in imperative programming often relies on loops that terminate under certain conditions. In Python, for example, a loop might terminate when a variable `n` equals zero. The challenge is to transform such an iterative solution into a recursive one.

To convert an iterative operation into a recursive one, follow these steps:

1. **Identify the loop and its invariants:** A loop invariant is a condition that holds true before and after each iteration of the loop, and find a way to reduce the problem to a smaller instance of the same problem.
2. **Determine the base case:** The base case of the recursive function corresponds to the loop's termination condition, and also it is the simplest version of the problem that can be solved directly.
3. **Re-evaluate loop variables:** Instead of mutating variables, pass them as arguments into the recursive function with updated values.

Consider the problem of calculating the factorial of a number. Below is an iterative approach:

```
def factorial_iterative(n):
    result = 1
    while n > 0:
        result *= n
        n -= 1
    return result

# Example usage
print(factorial_iterative(5))  # Outputs: 120
```

Now, let's transform this iterative solution into a recursive one:

- The loop continues until n becomes 0.
- The base case is when n equals 0, the function returns 1.
- result is updated in each iteration by multiplying it with the current value of n. After each multiplication, n is decremented by 1.

```
def factorial_recursive(n):
    if n == 0:
        return 1
    return n * factorial_recursive(n-1)

def factorial_tail_recursive(n, accumulator=1):
    if n == 0 or n == 1:  # Base case: factorial of 0 or 1 is 1
        return accumulator
    else:
        return factorial_tail_recursive(n - 1, n * accumulator)  # Tail call with updated accumulator


# Example usage
print(factorial_recursive(5))  # Outputs: 120
```

For the first non-tail-recursive:

```
Summary of Recursive Calls and Accumulator Updates
Step 1: factorial_recursive(5) → 5 * factorial_tail_recursive(4)
Step 2: factorial_recursive(4) → 4 * factorial_tail_recursive(3)
Step 3: factorial_recursive(3) → 3 * factorial_tail_recursive(2)
Step 4: factorial_recursive(2) → 2 * factorial_tail_recursive(1)
Step 5: factorial_recursive(1) → return 1
Step 6: return 2 * 1 → 2
Step 7: return 3 * 2 → 6
Step 8: return 4 * 6 → 24
Step 9: return 5 * 24 → 120
```

For the second tail-recursive:

```
Summary of Recursive Calls and Accumulator Updates
Step 1: factorial_tail_recursive(5, 1) → factorial_tail_recursive(4, 5)
Step 2: factorial_tail_recursive(4, 5) → factorial_tail_recursive(3, 20)
Step 3: factorial_tail_recursive(3, 20) → factorial_tail_recursive(2, 60)
Step 4: factorial_tail_recursive(2, 60) → factorial_tail_recursive(1, 120)
Step 5: factorial_tail_recursive(1, 120) → returns 120
```

4.7 Structural Recursion on Lists

This recursive structure not only influences how lists are represented but also how operations on lists are implemented. Consider the problem of computing the sum of the elements in a list. While an iterative approach would process each element sequentially and accumulate the sum, a **recursive** approach mirrors the recursive structure of the list:

- The sum of an empty list is `0`.
- The sum of `x cons lst` is `x` plus the sum of `lst`.

This description directly translates into a pure function:

```racket
; Racket
(define (sum lst)
  (if (empty? lst)
      0
      (+ (first lst) (sum (rest lst)))))
```

```haskell
-- Haskell
sum lst =
  if null lst
  then 0
  else head lst + sum (tail lst)
```

Here's another example: a function that filters a list to return a new list containing only the multiples of three.

```racket
(define (multiples-of-3 lst)
  (cond [(empty? lst) '()]
        [(equal? 0 (remainder (first lst) 3))
         (cons (first lst) (multiples-of-3 (rest lst)))]
        [else (multiples-of-3 (rest lst))]))
```

This kind of recursion is known as structural recursion because the code mirrors the structure of the data type it operates on. What's notable about this technique is that it is entirely based on the form of the input data, allowing us to create a "**code template**" that can be used for a wide variety of list functions:

```racket
(define (f lst)
  (if (empty? lst)
      ...
      (... (first lst) (f (rest lst)))))
```

Moreover, this technique is not limited to lists but can be applied to any data type with a recursive definition.

Try to use structural recursion to solve the following questions:

```racket
;find an element in a given list or not

;The empty list will never hold `val`
;...or...
;val is equal to first list
;val is found in rest list

(define (find-in-list l v)
    (if (empty? l)
        #f
        (or (eq? (first l) v) (find-in-list (rest l) v))))
```

```
;find the length of a list

;The empty list has a length of zero
;...or...
;the list's length is one more than the rest of the list

(define (list-length l)
    (if (empty? l)
        0
        (+ 1 (list-length (rest l)))))
```

Thus we can summarize a function that operates on a list is therefore:

- **A base case on the empty list**
- **Some operation on the first element combined with a recursive call on rest of the list**

4.8 What Is Tail Call?

As explored in prior coursework, function invocations are managed on the call stack, a memory segment designated for tracking active functions during program execution. In non-recursive setups, the call stack's size rarely poses a problem; however, in recursive scenarios, it rapidly becomes populated with function calls. Consider this Python example where the number of function calls is $\mathcal{O}(n)$:

```python
def f(n):
    if n == 0:
        return 0
    else:
        return f(n - 1)

# This triggers a RuntimeError due to the call stack limit being surpassed.
f(10000)
```

This memory consumption issue is not unique to Python but is also prevalent in Racket and Haskell. Nevertheless, these languages, among others, implement **tail call optimization**, which drastically lowers the memory needs of recursive functions.

A tail call occurs when a function call is the final action before a function returns. The call to `f(n-1)` in the prior example exhibits this property. Tail calls do not require the system to retain the calling context since the subsequent action simply involves returning the value to the original caller. This mechanism is universally applicable across programming languages; however, its implementation varies. For instance, Racket optimizes tail calls by removing the previous function's stack frame from the call stack, maintaining a constant stack height for this:

```
1 (define (f n)
2   (if (equal? n 0)
3       0
4       (f (- n 1))))
```

Revisiting our sum function from an earlier discussion, observe that it is not **tail-recursive**. In the "else" clause, it is the addition operation, not the `sum`, that occurs in the tail position. The call to `sum` is nested within the subsequent function call:

```
(define (sum lst)
  (if (empty? lst)
      0
      (+ (first lst)
          (sum (rest lst)))))
```

In this sum function, the recursive call (sum (rest lst)) is not in a tail position because the result of the call is added by first element after it returns. Therefore, Racket cannot optimize this call. To convert this to a tail-recursive format, we introduce an accumulator to capture the ongoing sum, updating its value through each recursive call:

```
(define (sum-tail lst)
  (sum-helper lst 0))

```

```
(define (sum-helper lst acc)
  (if (empty? lst)
      acc
      (sum-helper (rest lst) (+ acc (first lst)))))
```

Using the accumulator, `acc`, we track the cumulative sum as the list is processed. The execution flow for a sample call is detailed below:

```
(sum-tail '(1 2 3 4))
(sum-helper '(1 2 3 4) 0)  % Initial call; acc = 0
(sum-helper '(2 3 4) 1)    % New acc value: (+ 0 (first '(1 2 3 4))) = 1
(sum-helper '(3 4) 3)      % acc = (+ 1 (first '(2 3 4))) = 3
(sum-helper '(4) 6)        % acc = (+ 3 (first '(3 4))) = 6
(sum-helper '() 10)        % acc = (+ 6 (first '(4))) = 10
10                         % Base case: acc is returned
```

This function computes the sum of all numbers from n to 0. The recursive call to sum is in the tail position because there are no further computations after the call. Racket will optimize this call with tail call optimization, allowing it to run efficiently even for large list.

Now we can give a more detailed explaination for two recursive functions:

- **Non-tail recursion: "solves by deferring work until the smaller problem is solved."**
- **Tail recursion: "solves by carrying the work along so that the solution is obtained immediately at the base case."**

We can also do it in Haskell:

```haskell
-- Define a simple recursive sum function
sumRecursive :: Num a => [a] -> a
sumRecursive [] = 0
sumRecursive (x:xs) = x + sumRecursive xs

-- Define the helper function as a separate top-level function
sumHelper :: Num a => [a] -> a -> a
sumHelper [] acc = acc
sumHelper (x:xs) acc = sumHelper xs (acc + x)

-- Define the main tail-recursive sum function
sumTail :: Num a => [a] -> a
sumTail list = sumHelper list 0

main :: IO ()
main = do
  print $ sumTail []                    -- Should output 0
  print $ sumTail [1, 2, 3, 4, 5]       -- Should output 15
  print $ sumRecursive [1, 2, 3, 4, 5]  -- Should output 15

```

Try more:

```
;find the length of a list

(define (list-length l)
    (if (empty? l)
        0
        (+ 1 (list-length (rest l)))))

;tail recursive
(define (list-length-tail l)
   (length-helper l 0))

(define (length-helper lst acc)
   (if (empty? lst)
       acc
       (length-helper (rest lst) (+ acc 1))))
```

```
(define (multiples-of-3 lst)
  (cond [(empty? lst) '()]
        [(equal? 0 (remainder (first lst) 3))
         (cons (first lst) (multiples-of-3 (rest lst)))]
        [else (multiples-of-3 (rest last))]))

;Tail-recursive
(define (multiples-of-3 lst)
  (multiples-of-3-helper lst '()))

(define (multiples-of-3-helper lst acc)
  (if (empty? lst)
      acc
      (multiples-of-3-helper (rest lst)
                             (if (equal? 0 (remainder (first lst) 3))
                                 (append acc (list (first lst)))
                                 acc))))
```

Consider again the function `sum-tail` defined in Racket:

```
(define (sum-tail lst)
  (sum-helper lst 0))

(define (sum-helper lst acc)
  (if (empty? lst)
      acc
      (sum-helper (rest lst) (+ acc (first lst)))))
```

Tracing the evaluation of `(sum-tail '(1 2 3 4))`, we observe the sequence of function calls:

- `(sum-helper '(1 2 3 4) 0)`
- `(sum-helper '(2 3 4) 1)`
- `(sum-helper '(3 4) 3)`
- `(sum-helper '(4) 6)`
- `(sum-helper '() 10)`

In a language lacking tail call elimination, each of these calls would consume a separate stack frame. However, in a language like Racket, which supports tail call elimination, each function call effectively replaces the previous one's stack frame. Thus, **we can interpret this not as distinct function calls, but rather as successive iterations of a loop, with the variables `lst` and `acc` updating in each iteration**. This is analogous to iterative loops.
To illustrate this, consider an analogous Python implementation, knowing that **Python does not support tail call elimination**, which however does not alter the algorithm:

```
def sum_tail(lst):
    return sum_helper(lst, 0)

def sum_helper(lst, acc):
    if lst == []:
        return acc
    else:
        return sum_helper(lst[1:], acc + lst[0])
```

We can transform `sum_helper` into a loop as follows:

- Initialize variables `lst` and `acc` to the initial parameters of `sum_helper`.
- Encapsulate the body of `sum_helper` in a `while True` loop.
- Replace the recursive call with updates to `lst` and `acc`.

Applying these transformations yields:

```python
def sum_tail2(main_lst):
    lst, acc = main_lst, 0
    while True:
        if lst == []:
            return acc
        else:
            lst, acc = lst[1:], acc + lst[0]
```

While this code captures the iterative equivalent of the tail-recursive function, it uses idiomatic Python techniques somewhat awkwardly, such as the perpetual `while True` loop and the slicing of lists. A more Pythonic approach, which simplifies iteration through the list, is as follows:

```python
def sum_tail3(main_lst):
    acc = 0
    for x in main_lst:
        acc += x
    return acc
```

This last version demonstrates a clear, idiomatic way to perform the sum, leveraging Python's inherent capabilities for direct iteration over lists.

4.9 Pattern Matching

Pattern matching allows programmers to define behavior based on the structure of data using a clear syntax. For illustration, consider this conventional value-based branching in a function:

```haskell
f x =
  if x == 0 then
    10
  else if x == 1 then
    20
  else
    x + 30

main :: IO ()
main = print (f 1)
```

Although this approach works, it can be greatly simplified in Haskell using pattern matching directly in the function definition:

```haskell
f 0 = 10
f 1 = 20
f x = x + 30

main :: IO ()
main = print (f 1)
```

Haskell's function definition syntax eliminates the need for explicit if-else statements by using patterns directly (0, 1, or a variable 'x' which matches anything). **The order of these patterns is crucial as they are evaluated from top to bottom, with the first match being selected.** This method automatically handles the condition testing when the function is called.
The concept is similarly implemented in Racket, though the syntax is slightly more verbose using the 'define/match' keyword:

```racket
#lang racket

(define/match (f x)
  [(0) 10]
  [(1) 20]
  [(_) (+ x 30)])  ; The underscore '_' matches any other case

(f 1)
```

define/match is a syntax extension used for **defining functions that incorporate pattern matching directly into their definitions**. It allows for concise function definitions that handle different patterns of inputs with corresponding outputs. Essentially, define/match integrates the match syntax directly into the function definition, enabling clear and elegant handling of various cases based on the function's arguments.

Can you explain the following code to me?

```
#lang racket
(define/match (slow-add m n)
    [(m 0) m]
    [(m n) (slow-add (+ m 1) (- n 1))])
```

While this example might seem trivial, pattern matching is incredibly powerful and extends beyond mere value equality checks. It is particularly useful in recursive functions, such as the one for summing a list:

```
#lang racket
(define (sum lst)
  (if (empty? lst)
      0
      (+ (first lst) (sum (rest lst)))))
```

Both Racket and Haskell enable structural pattern matching to decompose values into parts, assigning each to a new identifier. This simplifies complex data manipulations. For instance, in Racket, patterns like '(list)' match an empty list, and 'cons' helps in decomposing a list into its head and tail:

```
#lang racket
(define/match (sum lst)
  [((list)) 0]
  [((cons x xs)) (+ x (sum xs))])
```

More examples:

```

(define (list-length l)
    (if (empty? l)
        0
        (+ 1 (list-length (rest l)))))

(define/match (list-length l)
  [((list)) 0]                        ; empty list
  [((cons _ xs)) (+ 1 (list-length xs))])

(define/match (list-length l acc)
  [('() acc) acc]
  [((cons _ xs) acc) (list-length xs (add1 acc))])

(define (multiples-of-3 lst)
  (cond [(empty? lst) '()]
        [(equal? 0 (remainder (first lst) 3))
         (cons (first lst) (multiples-of-3 (rest lst)))]
        [else (multiples-of-3 (rest last))]))

(define/match (multiples-of-3 lst)
  [('()) '()] ; empty list case
  [((cons x xs))
   (if (zero? (remainder x 3))
       (cons x (multiples-of-3 xs))
       (multiples-of-3 xs))])

(define/match (multiples-of-3 lst acc)
  [('() acc) acc]
  [((cons x xs) acc)
   (if (zero? (remainder x 3))
       (multiples-of-3 xs (cons x acc))
       (multiples-of-3 xs acc))])
```

In Haskell, the syntax is even more concise, using '[]' for an empty list and ':' for decomposing a list:

```haskell
my_sum [] = 0
my_sum (x:xs) = x + sum xs

main :: IO ()
main = print (my_sum [1,2,3])
```

Structural pattern-matching, concise syntax for decomposing a value into subparts and binding each part to a new identifier that can be referred to independently (destructuring).

Also, you can match just part of the variables. **match** is the standard pattern-matching construct in Racket. It matches a single expression against multiple patterns, executing the corresponding code for the first pattern that matches.

```
(match expr
  [pattern1 result1]
  [pattern2 result2]
  ...)
```

```
(define (example-match x)
  (match x
    [0 "zero"]
    [1 "one"]
    [2 "two"]
    [_ "other"]))
```

For Racket, we also have a form for pattern match. **match*** matches a sequence of values against each clause in order, matching only when all patterns in a clause match. Each clause must have the same number of patterns as the number of val-exprs.

```
(match* (expr1 expr2 ...)
  [((pattern1 pattern2 ...)) result1]
  ...)
```

```
#lang racket

(define (describe-two a b)
  (match× (a b)
    [((1 1)) "both are one"]
    [((1 _)) "first is one, second is something else"]
    [((_ 2)) "second is two, first is something else"]
    [((_ _)) "could any number else"]))
```

Python introduced support for pattern matching in version 3.10, through the addition of Structural Pattern Matching, which is somewhat similar to the pattern matching found in languages like Haskell and Racket.

```python
def sum_list(lst):
    match lst:
        case []:
            return 0
        case [first, *rest]:
            return first + sum_list(rest)

# Example usage
print(sum_list([1, 2, 3, 4]))  # Output should be 10
```

4.10 Higher-Order Functions

As mentioned earlier, functions themselves are values, which naturally leads to the question: can functions operate on other functions? Yes. In fact, this is at the core of functional programming: the ability for functions to accept other functions as inputs, use them, combine them, and even return new functions. Let's look at some simple programming examples.

```
; Take an input ×function× and apply it to 1
(define (apply-to-1 f) (f 1))
(apply-to-1 even?) ; #f
(apply-to-1 list) ; '(1)
(apply-to-1 (lambda (x) (+ 15 x))) ; 16

; Take two functions and apply them to the same argument
(define (apply-two f1 f2 x)
  (list (f1 x) (f2 x)))
(apply-two even? odd? 16) ; '(#t #f)
```

A **higher-order function** is a function that either:

1. **Takes one or more functions as arguments**: It can receive other functions as inputs and use them in its operations.
2. **Returns a function as its result**: It can produce a new function as its output.

Higher-order functions are fundamental in functional programming because they allow functions to be composed, combined, and reused in powerful ways. The higher-order functions have some key characteristics:

- **Abstraction**: They allow abstraction over actions, not just values. You can write more generic and reusable code by abstracting the common patterns of computation.
- **Modularity**: By using higher-order functions, you can build complex behavior by combining simpler functions, leading to more modular and readable code.
- **Flexibility**: They enable creating new functionality dynamically by generating new functions at runtime or combining existing ones.

```
;Function that takes another function as an argument:
(define (apply-twice f x)
  (f (f x)))
```

Here, `apply-twice` is a higher-order function because it takes a function `f` as an argument and applies it twice to a value `x`.

```
;Function that returns another function
(define (make-adder n)
  (lambda (x) (+ x n)))
```

Here, `make-adder` is a higher-order function that returns a new function (a lambda expression) that adds `n` to its argument.
Let's think of a more complicated scenario, I want to write a function applies a given function f to every element in the list l and returns a new list containing the results.

```
(define/match (function-to-all f l)
  ((_ '()) '())
  ((_ (cons n xs)) (cons (f n) (function-to-all f xs))))

(define (double x)
  (+ x x))

(function-to-all double '(1 2 3))
```

What if I want to write a function to process the elements of the list l using a binary function f (e.g., addition + or multiplication *). It recursively reduces the list to a single value.

```
#lang racket
(define (list-process f l)
  (match× (l)
    [('()) 0]
    [((cons x xs)) (f x (list-process f xs))]))

(define (list-sum l) (list-process + l))
(define (list-product l) (list-product × l))
```

4.10.1 Map and Filter

With the discussion in the previous section, you might think that functional programmers spend all their time writing recursive functions. However, this is not actually the case! Rather than using explicit recursion, code often utilizes three essential higher-order functions for list processing. The first two of these functions, `map` and `filter`, are quite straightforward:

```
; (map f lst)
; Returns a new list by applying `f` to each element in `lst`
> (map (lambda (x) (× x 3)) (list 1 2 3 4))
'(3 6 9 12)

; (filter pred lst)
; Creates a new list whose elements are those in `lst`
; that make `pred` output #t.
> (filter (lambda (x) (> x 1)) (list 4 -1 0 15))
'(4 15)
```

To demonstrate the third common higher-order function, let's first see how we might implement `map` and `filter` using loops and mutation:

```
def map(f, lst):
    acc = []
    for x in lst:
        acc.append(f(x))
    return acc

def filter(pred, lst):
    acc = []
    for x in lst:
        if pred(x):
            acc.append(x)
    return acc
```

Both of these functions utilize the same accumulator pattern, where an accumulator variable stores a value that is updated at each iteration of the loop and is ultimately returned at the end of the function. This pattern can be generalized into a higher-order function that accepts two additional arguments: **an initial value and a function to update the accumulator during each loop iteration**.

```
def accumulate(init, acc, update):
    acc = init
    for x in lst:
        acc = update(acc, x)
    return acc
```

4.10.2 Foldl and Foldr

This more general loop pattern is codified recursively in both Racket and Haskell using a function called `foldl`, used to reduce a list (or other sequence) to a single value by applying a function to each element of the list, starting from the left (the beginning of the list).:

```
(define (foldl update init lst)
  (if (empty? lst)
      init
      (foldl update
             (update (first lst) init)
             (rest lst))))

;write it using pattern match
(define (foldl update init lst)
  (match lst
    ['() init] ; If the list is empty, return the initial value.
    [(cons x xs) ; If the list is not empty, separate it into head (x) and tail (xs).
     (foldl update (update x init) xs)]))
```

```haskell
foldl update init lst =
    if null lst
    then init
    else foldl update (update init (head lst)) (tail last)

--Using pattern match
foldl update init lst =
  match lst with
    [] -> init                          -- If the list is empty, return the initial value.
  | (x : xs) -> foldl update (update init x) xs  -- If the list is not empty, combine the head with the accumulated result.
Explanation:
```

Consider the expression `(foldl - 0 '(1 2 3 4))`. The recursive construction of the list `'(1 2 3 4)` is:

`(cons 1 (cons 2 (cons 3 (cons 4 empty))))`

You can think of `foldl` as a function that replaces `cons` with a function `f` and `empty` with a value `init` from the left side of the list.

becomes:

$$f(4, f(3, f(2, f(1, init))))$$

Applying this to our example:

$$(-4(-3(-2(-1\ 0))))$$

What if I want to do it from the right?

```racket
(define (foldr update init lst)
  (if (empty? lst)
      init
      (update (first lst)
              (foldr update init (rest last)))))

;write using pattern match
(define (foldr update init lst)
  (match lst
    ['() init] ; If the list is empty, return the initial value.
    [(cons x xs) ; If the list is not empty, separate it into head (x) and tail (xs).
     (update x (foldr update init xs))]))
```

Consider the expression `(foldr - 0 '(1 2 3 4))`. The recursive construction of the list `'(1 2 3 4)` is:

`(cons 1 (cons 2 (cons 3 (cons 4 empty))))`

You can think of `foldr` as a function that replaces `cons` with a function `f` and `empty` with a value `v` (starting from the right side of the list). Thus, starting from the right side:
becomes:

$$f(1, f(2, f(3, f(4, init))))$$

Applying this to our example:

$$(-1(-2(-3(-4\ 0))))$$

4.10.3 Compose in Racket and Haskell

In Racket, the `compose` function is used to create a new function by combining two or more existing functions. It takes multiple functions as arguments and returns a new function that represents the composition of those functions. The composed function applies its input to the last function first and then applies each preceding function to the result of the previous one. The `compose` function has the following form:

`(compose f1 f2 ...fn)`

- `f1, f2, ..., fn`: These are the functions that will be composed.

- The function returned by `compose` is equivalent to applying `f1` to the result of `f2`, `f2` to the result of `f3`, and so on, ending with `fn`.

If you have functions `f`, `g`, and `h`, and you compose them like this:

```
(define new-func (compose f g h))
```

Then, calling `(new-func x)` is equivalent to:

$$f(g(h(x)))$$

This means that the input `x` is first passed to `h`, the result of `h(x)` is then passed to `g`, and finally, the result of `g(h(x))` is passed to `f`.

Let's see an example to illustrate how `compose` works:

```
(define add1 (lambda (x) (+ x 1)))
(define square (lambda (x) (× x x)))

(define add1-then-square (compose square add1))

(add1-then-square 4) ; This will return 25
```

- Here, `add1` is a function that adds 1 to its input.
- `square` is a function that squares its input.
- `add1-then-square` is a composed function created by `(compose square add1)`.
- When `(add1-then-square 4)` is called:
  1. `add1` is applied first: `add1(4)` produces `5`.
  2. Then `square` is applied to the result: `square(5)` produces `25`.

Haskell has a similar `compose` function, which is represented by the `(.)` operator. This operator allows you to compose two or more functions in Haskell, just like the `compose` function in Racket. In Haskell, the `(.)` operator is defined as:

```
(.) :: (b -> c) -> (a -> b) -> a -> c
f . g = \x -> f (g x)
```

- **Type Signature**:
  - The `(.)` operator takes two functions:
    - `g`, a function from type `a` to type `b` (`a -> b`).
    - `f`, a function from type `b` to type `c` (`b -> c`).
  - It returns a new function that takes an argument of type `a` and returns a result of type `c`.
- **Definition**:
  - The expression `f . g` is equivalent to the function that applies `g` to its argument `x` and then applies `f` to the result of `g x`. In other words, `(f . g) x` is the same as `f (g x)`.

The `(.)` operator allows you to compose functions such that the result of one function is automatically passed as the input to another. If you have functions `f` and `g`, you can compose them as follows:

$$(\texttt{f . g})\ x$$

This is equivalent to:

$$\texttt{f (g x)}$$

Let's look at an example in Haskell to illustrate the use of the `(.)` operator:

```haskell
add1 :: Int -> Int
add1 x = x + 1

square :: Int -> Int
square x = x * x

add1ThenSquare :: Int -> Int
add1ThenSquare = square . add1

result = add1ThenSquare 4  -- This will return 25
```

- `add1`: A function that adds 1 to its input.
- `square`: A function that squares its input.
- `add1ThenSquare`: A composed function created by using `(.)`, which represents `square . add1`.
  - This means `add1` is applied first, followed by `square`.
- When you call `add1ThenSquare 4`:
  1. `add1` is applied first: `add1(4)` produces `5`.
  2. Then `square` is applied to the result: `square(5)` produces `25`.

4.10.4 Currying

Currying is a concept in functional programming that transforms a function that takes multiple arguments into a sequence of functions, each taking a single argument (**breaks down a function of multiple arguments into a series of unary (single-argument) functions**). This technique allows functions to be partially applied, meaning that **you can fix a certain number of arguments and generate a new function that takes the remaining arguments**.
In Haskell, all functions are automatically curried. This means a function that appears to take multiple arguments actually takes one argument and returns a new function that takes the next argument, and so on. Consider the following function definition in Haskell:

```haskell
add :: Int -> Int -> Int
add x y = x + y
```

- **Currying**: The function `add` has the type `Int -> Int -> Int`, which is equivalent to `Int -> (Int -> Int)`. This means that `add` is a function that takes an integer `x` and returns another function that takes an integer `y` and returns their sum.
- **Partial Application**: You can partially apply the function by providing only one argument:

```haskell
addFive :: Int -> Int
addFive = add 5

result = addFive 3  -- This returns 8
```

  Here, `addFive` is a partially applied version of `add` that always adds `5` to its input.

Racket does not have built-in currying like Haskell, but you can manually create curried functions using lambda expressions or the `curry` function from the `racket/function` library. To define a curried function in Racket, you can use nested lambdas:

```racket
(define (add x)
  (lambda (y) (+ x y)))

(define add-five (add 5))

(add-five 3) ; This returns 8
```

- **Currying**: The `add` function takes one argument `x` and returns another function `(lambda (y) (+ x y))` that takes the second argument `y` and returns their sum.
- **Partial Application**: `add-five` is a partially applied function that always adds `5` to its input.

Alternatively, you can use the `curry` function from the `racket/function` library:

```
(require racket/function)

(define add (curry +))

(define add-five (add 5))

(add-five 3) ; This returns 8
```

Here, `curry` creates a curried version of the `+` function. Now you can see the differences between Haskell and Racket:

- **Automatic Currying**: In Haskell, all functions are automatically curried. In Racket, functions are not curried by default; you must explicitly create curried functions.
- **Syntax and Usage**: Haskell's syntax for currying is built into the language and is straightforward due to its function definition style. In Racket, you use lambdas or the `curry` function to achieve the same effect.

4.10.5 Apply

The `apply` function is a higher-order function available in both Racket and Haskell that allows you to call a function with a list of arguments. It is useful when the number of arguments to a function is not known in advance or when the arguments are stored in a data structure like a list.
In Racket, the `apply` function takes a function and a list of arguments and calls the function with the arguments provided in the list.

`(apply function arg-list)`

```
(apply + '(1 2 3 4))   ; This returns 10
(apply max '(5 7 2 8)) ; This returns 8
; More generally,
(apply f (list x1 x2 x3 ... xn))
; equivalent to...
(f x1 x2 x3 ... xn)
```

Explanation:

- The first example calls the `+` function with the arguments `1, 2, 3,` and `4`. It sums these numbers and returns `10`.
- The second example calls the `max` function with the arguments `5, 7, 2,` and `8`. It returns the maximum value, which is `8`.

In Haskell, the `apply` function is not built-in with the same name as in Racket, but its behavior can be mimicked using function application techniques. Haskell allows you to use the `$` **operator** or function composition to achieve a similar result.

Example in Haskell Consider the following Haskell example that uses `foldl` to mimic `apply`:

```
apply :: (a -> a -> a) -> [a] -> a
apply f (x:xs) = foldl f x xs
apply _ [] = error "empty list"

result1 = apply (+) [1, 2, 3, 4]  -- This returns 10
result2 = apply max [5, 7, 2, 8]  -- This returns 8
```

Explanation:

- The `apply` function is defined using `foldl`, which folds the list from the left using the binary function `f`.
- The first example applies the `+` function across the list `[1, 2, 3, 4]` to compute the sum, resulting in `10`.

- The second example applies the `max` function across the list `[5, 7, 2, 8]` to find the maximum value, which is 8.

```haskell
($) :: (a -> b) -> a -> b
-- Composing multiple functions using £
example = negate $ abs $ -10 -- result is -10
--equal to example = negate (abs (-10))

```

For `apply`, the list elements must match the expected number of arguments, which means it represents a single function call! Different from the `foldl`, which repeatedly applies a binary function and an accumulator. For example:

```racket
(define (add2 x y) (+ x y))
(apply add2 '(9 10)) ;correct
(apply add2 '(9 10 11)) ; wrong, the list of arguments does not match the function parameters
```

4.10.6 Functions Returning Functions

We have now explored functions that take primitive values or other functions as inputs, but so far they have all returned primitive values. Now, let's focus on another type of higher-order function: a function that returns another function. This is an extremely powerful concept because it allows us to dynamically define new functions during the execution of a program. Here is a simple example:

```racket
(define (make-adder x)
  ; The body of make-adder is a function value.
  (lambda (y) (+ x y)))
(make-adder 10) ; #<procedure>

(define add-10 (make-adder 10))
add-10 ; #<procedure>
(add-10 3) ; 13
```

The function `add-10` indeed seems to add ten to its argument. Using the substitution model of evaluation for `(make-adder 10)`, we can see how this works:

```racket
(make-adder 10)
; ==> (substitute 10 for x in the body of make-adder)
(lambda (y) (+ 10 y))
```

4.11 Programming with Abstract Syntax Trees

In this section, we will explore how abstract syntax trees (ASTs) are represented in actual code and how structural recursion enables us to operate on these trees with ease.

4.11.1 Racket: Quoted Expressions

Let's first see how we can represent ASTs in Racket. This is one of the fundamental strengths of all Lisp languages: the parenthesized source code inherently creates a nested list structure, which is another way to represent a tree. To make this even more explicit, any Racket expression (regardless of its complexity or depth) can be converted into a static nested list by prefixing it with an **apostrophe**, which is known as *quoting* an expression.

```racket
> (+ 1 2) ; A regular Racket expression
3
> '(+ 1 2) ; Quoting the expression: a list of three elements.
'(+ 1 2)
> (first '(+ 1 2))
'+
> (second '(+ 1 2))
1
> (third '(+ 1 2))
2
> '(+ (× 2 3) (× 4 5)) ; This is a nested list
'(+ (× 2 3) (× 4 5))
```

Although a quoted expression in Racket is just a list, we refer to it as a *Racket datum* to differentiate it from a regular list. Here are the types of values that make up the "tree" in a Racket datum:

1. **Quoted literals** (e.g., `5`, `#t`, `"hello"`) represent the same values as in the original code. For instance, `(equal? '5 5)` returns true.
2. **Quoted identifiers and keywords** become symbols; a symbol is a primitive data type that functions as a unique identifier. For example, the datum `'(+ 1 2)` has its first element as the symbol `'+`, which can be compared using `equal?` and used in pattern matching. Keywords like `define` are also quoted into symbols. The only way to distinguish between identifiers and keywords is to start with a fixed set of keywords to check.
3. **Quoted compound expressions** become lists, where each element is the quoted version of the corresponding subexpression. For example, `'(+ 2 3)` is equivalent to `(list '+ '2 '3)`, which is further equivalent to `(list '+ 2 3)`.

Given this structure, we can use structural pattern matching to process a datum and perform recursive computations on it. For example, here is a function that takes a Racket datum consisting of numeric literals, identifiers, and arithmetic function calls, and returns the number of times the `+` symbol is used, the pattern (list expr ...) binds the contents of the list to expr, where expr is a list of sub-expressions of arbitrary length:

```
(define/match (num-plus datum)
  ; For a function call (which in this case is ×any× parenthesized
  ; expression), recurse on each subexpression and add up the total
  ; number of occurrences. Note the use of (list expr ...) to bind
  ; lists of arbitrary length to expr.
  [((list expr ...))
   (apply + (map num-plus expr))]
  ; Value pattern—match on '+
  [('+) 1]
  ; For any other atomic value (identifier or literal), return 0
  [(_) 0])

> (num-plus '+)
1
> (num-plus 3)
0
> (num-plus '(+ (× 4 5) (+ (- (+ 3 6) 7) 8)))
3
```

Let's walk through the pattern matching step-by-step to understand how the function processes this expression. We start with the entire input expression and follow the pattern matching rules of the `num-plus` function.

1. **Initial Call**: `(num-plus '(+ (* 4 5) (+ (- (+ 3 6) 7) 8)))`
   - The input is the list: `'(+ (* 4 5) (+ (- (+ 3 6) 7) 8))`.
   - This matches the pattern `[(list expr ...)]`, representing a list of arbitrary length.
   - The function calls `(map num-plus expr)` to recursively process each subexpression (`'+`, `'(* 4 5)`, `'(+(- (+ 3 6) 7) 8)`).
2. **Processing the First Subexpression**: `'(+)`
   - The first subexpression is `'(+)`, a list containing the `+` symbol.
   - This matches the pattern `[('+)]` (a quoted `+` symbol), so the function returns `1`.
   - **Result**: `1`
3. **Processing the Second Subexpression**: `'(* 4 5)`
   - The second subexpression is `'(* 4 5)`, another list.
   - This matches the pattern `[(list expr ...)]` again.
   - The function recursively calls `(map num-plus '(* 4 5))`:
     - The input `*` matches the pattern `[(_)]` (any other atomic value), returning `0`.
     - The input `4` matches the pattern `[(_)]` (any other atomic value), returning `0`.
     - The input `5` also matches the pattern `[(_)]` (any other atomic value), returning `0`.

- Result for `(* 4 5)`: `(apply + '(0 0 0))` = 0

4. **Processing the Third Subexpression**: `'(+ (- (+ 3 6) 7) 8)`
   - The third subexpression is `'(+ (- (+ 3 6) 7) 8)`, another list. This matches the pattern `[(list expr ...)]`.
   - First element `'+`. This matches the pattern `[('+)]` (a quoted + symbol), so the function returns 1.
   - The second element `'(- (+ 3 6) 7)` is a list. It matches the pattern `[(list expr ...)]` again. The function recursively calls `(map num-plus '(- (+ 3 6) 7)`:
     - First Element `'-`, matches `[(_)]`, returning 0.
     - second Element `'(+ 3 6)`:
       - The input is `'(+ 3 6)`, which matches the pattern `[(list expr ...)]`. It recursively calls `(map num-plus '(+ 3 6))`:
       - First Element (+): 3 matches `[('+)]`, returning 1.
       - Second element (3): 3 matches `[(_)]`, returning 0.
       - Third element (6): 6 matches `[(_)]`, returning 0.
       - Result for `(+ 3 6)`: `(apply + '(1 0 0))` = 1.
     - Third element (7): 7 matches `[(_)]`, returning 0.
     - Result for `(- (+ 3 6) 7)`: `(apply + '(1 0))` = 1.
   - Third element (8): 8 matches `[(_)]`, returning 0.
   - Result for `(+ (- (+ 3 6) 7) 8)`: `(apply + '(1 1 0))` = 2.

5. **Final Aggregation**:
   - `+`: 1
   - `(* 4 5)`: 0
   - `(+ (- (+ 3 6) 7) 8)`: 2
   - The final result is: `(apply + '(1 0 2))` = 3

The AST representation simplifies recursive algorithms by providing a clear, hierarchical structure for expressions, allowing for efficient pattern matching, recursive traversal, and easy manipulation of nested structures. This approach reduces complexity, enhances readability, and enables powerful operations on code represented as data.

Due to Haskell's type system, it is not possible to store values of different types in the same list. This makes it challenging to directly represent Racket datums in the same way, as Racket lists can freely mix nested lists, symbols, and literals of various types. Instead, Haskell's type system allows for the concise declaration of types with multiple constructors, and supports pattern matching on these types. We will explore this more formally later in the course, but for now, consider the following intuitive representation of an "expression" type:

```haskell
data Expr = NumLiteral Int        -- An integer literal
          | Identifier String     -- An identifier
          | Call Expr [Expr]      -- A function call, where the first
                                  -- Expr is the function being called,
                                  -- and the list [Expr] contains the
                                  -- argument expressions.
```

The specific syntax is not crucial here; the key idea is that `NumLiteral`, `Identifier`, and `Call` are three value constructors. These constructors respectively take an integer, a string, and an expression with a list of expressions, and they all return a value of type `Expr`. Note that `NumLiteral` and `Identifier` represent atomic expressions (they are the leaves in the AST), while `Call` is recursive. For example, the Racket expression `(+ 2 3)` would be represented by the following value:

```haskell
Call (Identifier "+") [NumLiteral 2, NumLiteral 3]
```

This might initially seem cumbersome, but because of structural pattern matching, it is equally straightforward to implement the `numPlus` function in Haskell.

```haskell
numPlus (Call f args) = numPlus f + sum (map numPlus args)
numPlus (Identifier "+") = 1
numPlus (Identifier _) = 0
numPlus (NumLiteral _) = 0
```

- **Base Cases**:
  - `numPlus (Identifier "+") = 1`: If the input expression is an `Identifier` with the value `"+"`, the function returns `1`, indicating that it found a `+` operator.
  - `numPlus (Identifier _ ) = 0`: If the input is any other `Identifier` (indicated by `_`), the function returns `0`, since it is not a `+` operator.
  - `numPlus (NumLiteral _ ) = 0`: If the input is a `NumLiteral` (any integer literal), the fun ction returns `0`, as integer literals do not contain the `+` operator.
- **Recursive Case**:
  `numPlus (Call f args) = numPlus f + sum (map numPlus args)`:
  - If the input expression is a `Call`, the function recursively processes the function being called (`f`) and all its arguments (`args`).
  - `numPlus f`: Recursively calls `numPlus` on the function `f` to count any occurrences of the `+` operator in it.
  - `map numPlus args`: Applies `numPlus` to each argument in the list `args`, recursively counting the number of `+` operators in each argument.
  - `sum (map numPlus args)`: Sums up the counts from all the arguments.
  - The total number of `+` operators in the current `Call` is the sum of `numPlus f` and the counts from all arguments.

Let's see how the `numPlus` function works with a specific example. Suppose we have the following expression, which represents the Racket expression `(+ 2 3)`:

```
1 expr = Call (Identifier "+") [NumLiteral 2, NumLiteral 3]
```

We call `numPlus` on `expr`:

1. **First Call**: `numPlus (Call (Identifier "+") [NumLiteral 2, NumLiteral 3])`
   - This matches the pattern `Call f args`, where:
     - `f` is `Identifier "+"`.
     - `args` is `[NumLiteral 2, NumLiteral 3]`.
2. **Recursive Processing**:
   - `numPlus f`:
     - `numPlus (Identifier "+")` matches `numPlus (Identifier "+") = 1`, so it returns `1`.
   - `sum (map numPlus args)`:
     - **First Argument** (`NumLiteral 2`):
       - `numPlus (NumLiteral 2)` matches `numPlus (NumLiteral _ ) = 0`, so it returns `0`.
     - **Second Argument** (`NumLiteral 3`):
       - `numPlus (NumLiteral 3)` matches `numPlus (NumLiteral _ ) = 0`, so it returns `0`.
     - The sum of these results is `0 + 0 = 0`.
3. **Combine Results**:
   The total result is `numPlus f + sum (map numPlus args) = 1 + 0 = 1`.

Thus, `numPlus expr` correctly counts `1` occurrence of the `+` operator in the expression.

4.12 Function Evaluation Order in Racket

Most modern programming languages use **strict semantics** for function calls. In this context, a function call consists of two types of subexpressions: the function being called and the arguments to the function. If any of these subexpressions contain an error, the function call itself generates this error. This is typically implemented (i.e., induces an operational semantics) through a **left-to-right eager evaluation** of function call expressions:

- When evaluating a function call, first evaluate the subexpression representing the function being called (often, but not always, an identifier).
- Then evaluate each argument subexpression in left-to-right order.
- Finally, "call" the function by substituting the value of each subexpression into the body of the function.

Because arguments are evaluated before being substituted into the body of a function, this guarantees that if one of the argument subexpressions (or even the function subexpression itself) is undefined, this is discovered before the function is actually called.

Similarly, most languages use strict semantics for name bindings. For example, in Racket, a top-level binding `(define <id> <expr>)` first evaluates `<expr>`, then binds the value of that expression to `<id>`.

```
((lambda (x) (+ x 6)) (+ 4 4))
; evaluate (+ 4 4)
((lambda (x) (+ x 6)) 8)
; substitute 8 for x
(+ 8 6)
14
```

And we know there is another evaluation order, which is the lazy evaluation order; we will see it later in Haskell. The Church-Rosser Theorem (informally) asserts that in lambda calculus, for any program of this type, regardless of the order in which functions are applied, the final result will always be the same (We've discussed this in the previous lambda calculus chapter). In simple terms, all paths lead to the same outcome. However, this informal interpretation only holds for valid function applications—those that don't result in an error or infinite loop. But what happens when errors or non-terminating computations occur?

This question brings us to a discussion about different evaluation strategies for function applications, which have significant consequences for programming languages. Although we'll focus on the pure functional context here, evaluation order becomes even more critical in imperative languages that include functions with side effects.

For instance, consider the function call (f 10 (/ 1 0)). Is it always guaranteed to trigger an error for any binary function f?

4.12.1 Syntactic Forms

In your programming experience, you have likely assumed eager evaluation order in function calls. However, you may have noticed that certain types of expressions do not eagerly evaluate all their subexpressions; for example, boolean operators and conditionals. In Racket, these include `and`, `or`, `if`, and `cond`. Even though these may appear to be ordinary functions, they are not! Consider writing your own "and" function by wrapping the built-in Racket `and`:

```
(define (my-and x y) (and x y))
```

Even though `my-and` looks similar to `and`, it is not the same due to differences in evaluation order. The built-in `and` can stop evaluating its arguments as soon as it encounters a "false" value, but this is not true for `my-and`—which, being a function, eagerly evaluates all of its arguments before passing their values to `and`:

```
(and #f (/ 1 0)) ; evaluates to #f
(my-and #f (/ 1 0)) ; raises an error
```

Because of this, we say that `and`, `or`, `if`, and `cond` are **syntactic forms**, rather than identifiers referring to built-in functions. This distinction is quite subtle and is not specific to Racket; in any programming language that uses eager evaluation for functions, short-circuiting boolean operations and conditionals are implemented as syntactic constructs, not simple function calls.

4.12.2 Function Bodies and Delaying Evaluation

Now that we have identified eager evaluation, consider other contexts where languages may or may not eagerly evaluate expressions. Take the following Racket function:

```
(define (f x)
  (length (range 3000000)))
```

Note that the body of this function does not use the parameter `x`; when the Racket interpreter evaluates this function, does it eagerly evaluate the body of `f`? How about in other languages you know?

It turns out that Racket does not: **in general, a function's body is not evaluated until the function is called, even if the body does not depend on the function's parameters**. This means that we can delay the evaluation of an expression simply by placing it inside a function body. **A *thunk* is a nullary function whose purpose is to delay the evaluation of its body expression**. For example, `(lambda () (/ 1 0))` is a thunk that wraps an error-raising division expression. When we evaluate the lambda, its body is not evaluated:

```
> (lambda () (/ 1 0)) ; A thunk as an anonymous function
#<procedure>
> (define (bad) (/ 1 0)) ; A thunk bound to a name
> bad ; No error when the name is evaluated
#<procedure:bad>
> (bad) ; ×Calling× the thunk raises an error
/: division by zero
```

Using this idea, we can simulate non-strict semantics for function calls by passing thunks that wrap the "actual" arguments:

```
; x and y are now thunks
(define (my-and x y)
  (and (x) (y)))

> (my-and (lambda () #f) (lambda () (/ 1 0)))
#f ; The (/ 1 0) is never evaluated!
```

4.13 Function Evaluation Order in Haskell

As mentioned at the start of this chapter, one of the key design features that distinguish Haskell from other languages is its use of **non-strict semantics** for both function calls and name bindings. In a function call, the argument subexpressions are not evaluated immediately. Instead, they are only evaluated if and when they are needed. If an argument is never used, it is never evaluated. This evaluation strategy is known as *lazy evaluation*. Consider the following example:

```
> onlyFirst x y = x
> onlyFirst 15 (head [])
15
> onlyFirst (head []) 15
*** Exception: Prelude.head: empty list
```

The function `onlyFirst` only requires its first argument to evaluate its body. In the first call, the argument `(head [])` is never evaluated. This allows us to define a short-circuiting "and" function just like any other Haskell function:

```
> myAnd x y = if x then y else False
> myAnd False (head [])
False
```

In fact, the boolean operators `(&&)` and `(||)` are just built-in functions in Haskell, unlike in most other programming languages! The non-strict semantics also apply to name bindings. In Haskell, a binding of the form `<id> = <expr>` does not evaluate `<expr>`; instead, it binds the expression to the identifier. The expression `<expr>` is only evaluated when the identifier needs to be evaluated:

```
> x = error "This is an error" -- No error when x is bound.
> x -- Error when x is evaluated.
*** Exception: This is an error
```

You might have wondered about the syntax for defining nullary functions in Haskell, given that the general function definition syntax `<id> <param> ... = <body>` seems to coincide with a "simple" name binding `<id> = <expr>` when there are no parameters. Now we can answer this: **every name binding in Haskell defines a function!** The binding `<id> = <expr>` defines a *thunk*, similar to `(define (<id>) <expr>)` in Racket, making it clear that the expression is not evaluated at the time of definition.

4.13.1 The Trouble with Haskell's `foldl`

Although we have not focused much on efficiency (in terms of time or space) in this course, there is one place where it has already come up: the concept of *tail call optimization*, which is crucial for reducing call stack growth when using recursion. Let's examine how the familiar `foldl` function is implemented in Haskell (mirroring the definition in Racket):

```haskell
foldl _ acc [] = acc
foldl f acc (x:xs) =
  let acc' = f acc x
  in
  foldl f acc' xs
```

Here, the `let` expression introduces an intermediate definition of `acc'` before the recursive call, making it clear that the recursive call to `foldl` is in tail call position. Indeed, Haskell performs tail call optimization, as expected of a pure functional language. However, when running a large computation like `foldl max 0 [1..100000000]`, not only does the computation take a long time, but its memory usage increases linearly over time. Why is this?

The key lies in understanding **Haskell's lazy evaluation order**. Although laziness might seem like a neat feature, it has drawbacks. The evaluation of functions becomes less predictable, making it harder to reason about both the time and space efficiency of a program. In the case of our `foldl` implementation, **laziness means that the `let` bindings are not evaluated until absolutely necessary**—and when is that? Not during the recursive call itself. The expression

```haskell
let acc' = f acc x
in
foldl f acc' xs
```

reduces to `foldl f (f acc x) xs`. This means that the second argument passed to the recursive call is a *thunk* whose body is `(f acc x)`, not the value produced by actually calling `f`. Let's illustrate this with a concrete example. Consider calling `foldl max 0 [1, 2, 3]`. In the initial call, the body of `foldl` becomes:

```haskell
let acc' = max 0 1
in
foldl max acc' [2, 3]
```

In a language with strict binding semantics, we would expect `max 0 1` to evaluate to `1` before being bound to `acc'`, resulting in the recursive call `foldl max 1 [2, 3]`. This is the case in a language like Racket, and combined with tail call optimization, this results in constant space usage with respect to the size of the input list, as the space used to store the accumulator is fixed (a single integer). However, in Haskell, `acc'` is bound to a thunk containing `(max 0 1)`. Therefore, even when the recursive call undergoes tail call optimization, the resulting argument data takes up more space than before: a thunk instead of the initial integer `0`. This space usage accumulates through the recursive calls, with each new binding of `acc'` resulting in a new thunk, leading to the linearly increasing space usage observed when we call this function. These thunks are only evaluated when we reach the end of the list and `acc` is returned and printed. Summary of acc' Bindings:

1. First Binding: acc' = (max 0 1) — a thunk representing the computation of max 0 1.
2. Second Binding: acc' = (max (max 0 1) 2) — a thunk representing max of the previous thunk and 2.
3. Third Binding: acc' = (max (max (max 0 1) 2) 3) — a thunk representing max of the previous thunk and 3.

Because laziness introduces unique challenges, Haskell provides ways to explicitly force the evaluation of an expression, even when the expression's value is not "necessary" for evaluating its containing outer expression. One approach is with the compiler extension `BangPatterns`, as shown here:

```haskell
{-# LANGUAGE BangPatterns #-}

foldl _ !acc [] = acc
foldl f !acc (x:xs) =
  let acc' = f acc x
  in
  foldl f acc' xs
```

The only difference is the first line (telling the compiler to use the `BangPatterns` extension) and the **exclamation mark** preceding the `acc` parameter in the function definition. This makes `acc` a strict parameter: whenever `foldl` is called, the argument expression corresponding to `acc` must be evaluated before being passed to `foldl`, including during recursive calls. **With this implementation of `foldl`, we achieve constant space evaluation of `foldl max 0 [1..100000000]`, benefiting from both tail call optimization and eager evaluation of the calls to `f`**

to ensure that only an integer accumulator is passed through the recursive calls. Why?

The expressions `(range 1 100000001)` in Racket and `[1..100000000]` in Haskell both generate a sequence of numbers from `1` to `100,000,000`, but they differ significantly in how they occupy space due to differences in the languages' evaluation strategies and data structures.

In Racket, `(range 1 100000001)` produces a sequence of numbers from `1` to `100,000,000`. By default, Racket evaluates expressions eagerly. When you call `(range 1 100000001)`, it generates a complete list of numbers from `1` to `100,000,000` in memory. This list is fully realized in memory at once, requiring space proportional to the size of the list, which is `100,000,000` integers. The space complexity is **O(n)**, where `n` is the number of elements (`100,000,000` in this case).

In Haskell, `[1..100000000]` represents a range of numbers from `1` to `100,000,000`. Haskell uses lazy evaluation by default. The expression `[1..100000000]` does not immediately generate all the numbers in memory. Instead, it represents a lazy list — a description of the computation needed to produce the numbers. The numbers are generated on demand as the program iterates over the list. This means that only a small portion of the list (typically a single element or a small number of elements, depending on what is needed) is held in memory at any given time. The space complexity is **O(1)** for the range itself because the list is represented as a series of thunks (unevaluated expressions) that produce the next value when required.

4.14 Lexical closures

Recall the `make-adder` function, which takes a number `x` and returns a new "add `x`" function:

```
(define (make-adder x)
  (lambda (y) (+ x y)))
```

Suppose we call `make-adder` multiple times: `(make-adder 10)`, `(make-adder -1)`, `(make-adder 9000)`, etc. A naive (but correct) implementation of Racket would create and store new functions in memory for each call:

```
; (make-adder 10)
(lambda (y) (+ x 10))
; (make-adder -1)
(lambda (y) (+ x -1))
; (make-adder 9000)
(lambda (y) (+ x 9000))
```

However, it seems inefficient for Racket to create and store new functions each time, especially since all these functions essentially have the same body, differing only in the value of `x`. A more efficient implementation factors out the common lambda expression `(lambda (y) (+ x y))`, which remains the same across all calls to `make-adder`, and stores only the binding for `x`, which varies with each function call. This is precisely how Racket implements it. An abstract but more accurate representation of the above function calls is: (Keep in mind that the `(lambda (y) (+ x y))` is stored only once in memory, and `make-adder` returns just a reference or pointer to that lambda.)

```
; (lambda (y) (+ x y)) is stored only once in memory
; (make-adder 10)
(x: 10) '(lambda (y) (+ x y))
; (make-adder -1)
(x: -1) '(lambda (y) (+ x y))
; (make-adder 9000)
(x: 9000) '(lambda (y) (+ x y))
```

The data structure that stores the reference to the function code and the binding for `x` is called a *closure*. When we refer to lambdas as "function values," each one is actually implemented as a closure! This concept only becomes relevant when the function body contains a *free identifier*, which is an identifier that is not local to the function. Because if every identifier in a function's body is either a parameter or bound in a local `let` expression, all the data needed to evaluate the function call is contained in the arguments and the function body itself—no additional lookup is necessary.

In the definition of `make-adder`, the inner lambda expression has a free identifier `x`, which must be looked up in the closure when the function is called. For example, when the expression `((make-adder 10) 3)` is evaluated, we take the function `(lambda (y) (+ x y))` and look up the value of `x` in the closure to retrieve `10`.

In summary, **a closure is a data structure that contains a pointer to a function and a collection of name-value bindings for all free identifiers in that function**. If the function body has no free identifiers, then the

closure can be identified with the function body itself, since its name-value binding is empty. For a closure to be evaluated, all identifiers inside the function body must be in scope when the function is defined, but they are not necessarily local to the function. This was true in our earlier example: while `x` was a free identifier in the inner lambda, it was still in scope because it was a parameter of the enclosing `make-adder`. In contrast, the following variation would raise a runtime error due to an unbound identifier:

```
(define (make-adder x)
  (lambda (y) (+ z y)))
```

4.15 Lexical and Dynamic Scope

We have just discussed that closures—and specifically, name bindings—are created when a lambda expression is evaluated. However, this is not the complete picture; there is another factor to consider when reasoning about how closures work. Consider the following variation of `make-adder`:

```
(define z 10)
(define (make-z-adder) (lambda (x) (+ x z)))
(define add-z (make-z-adder))

> (add-z 5)
15
```

Everything seems fine. The body of `make-z-adder` is a function with a free identifier `z`. Calling `make-z-adder` evaluates the lambda, returning a closure where the free `z` refers to the global variable, binding it to the value `10`. But what happens when we shadow `z` and then call `make-z-adder`?

```
(let× ([z 100]
       [add-z-2 (make-z-adder)])
    (add-z-2 5))
>;15 or 105?
```

Now the lambda expression is evaluated when the function is called in the local scope. But which value of `z` is bound in the closure? Does this expression output `15` (because `z` is bound to `10`) or `105` (because `z` is bound to `100`)? It turns out that this expression evaluates to `15`, just like the previous example. Our goal is to understand why.

In Racket, the value used is the one bound to the name that is in scope where the lambda expression appears in the source code. That is, although closures are created at runtime, how the closures are created (i.e., which bindings are used) is determined solely by where they are written in the source code. In the previous example, the `z` in the closure returned by `make-z-adder` is bound to the global `z`, which is the one in scope where the lambda expression is written.

Generally, the binding for any identifier in Racket can **be determined by its location in the source code, proceeding outwards through enclosing expressions until you find where the identifier is first introduced** (either through a binding or a parameter declaration). This is known as *lexical scope* or *static scope*, which means that this resolution can be determined by analyzing the source code itself, before running any code.

In contrast, *dynamic scope* resolves identifiers based on when they are used during program execution, rather than where they appear in the code. If Racket used dynamic scope, the above example would output `105`, because the closure created by `make-z-adder` would now bind to the enclosing local `z`.
In other words, lexical scoping resolves names based on the context from the source code, while dynamic scoping resolves names based on the context from the program state at runtime. Early programming languages used dynamic scope because it was easier to implement; names could be resolved by finding the nearest binding on the function call stack or by recursively accumulating name bindings in an interpreter. However, dynamic scope makes programs much harder to reason about, as the values of non-parameter names of a function depend on the various places the function is used, rather than the single place where it is defined. Lexical scope was revolutionary in how it simplified programming tasks, and is now used by every modern programming language.

The concepts of lexical scope and dynamic scope primarily apply to free variables — variables that are used within a function but are not defined locally inside that function. Local variables are defined within the function itself (either as parameters or declared within the function). For local variables, there is no ambiguity in their binding because they are always resolved within the function's local scope. Thus, Lexical and dynamic scope do not affect local variables because these variables are defined directly inside the function.

4.16 Closures in Python

A closure in Python is a function object that:

- Remembers the values in its enclosing scope even when the scope in which it was created is no longer active.
- Captures and retains references to the variables from its enclosing environment, allowing it to access those variables even after the outer function has finished executing.
- In simpler terms, a closure allows a nested function to "remember" the environment in which it was created, including any variables that were in scope at that time.

A closure is formed when:

- A nested function references one or more variables from its enclosing function.
- The enclosing function returns the nested function.
- The returned function maintains a reference to the variables in the outer scope, even after the outer function has finished executing.

Closures are powerful because they allow you to:

- Encapsulate State: Closures can store and remember state across multiple function calls without using global variables.
- Create Function Factories: Closures can be used to generate specialized functions that are tailored to specific tasks by capturing different variables from the enclosing environment.
- Implement Callbacks and Handlers: In asynchronous programming or event-driven programming, closures can be used to create callback functions that maintain state.

Here is a Python program:

```
def make_functions():
    flist = []
    for i in [1, 2, 3]:
        def print_i():
            print(i)
        print_i()
        flist.append(print_i)
    print('End of flist')
    return flist

def main():
    flist = make_functions()
    for f in flist:
        f()

>>> main()
#1
#2
#3
#End of flist
#3
#3
#3
```

- **Function** `make_functions`:
  - This function creates a list of functions (`flist`) that, when called, print the value of `i` from the enclosing scope of the `for` loop.
  - It iterates over the list `[1, 2, 3]` using a `for` loop, and for each value of `i`, it:
    - Defines a nested function `print_i` that prints the value of `i`.
    - Immediately calls `print_i` to print the current value of `i`.
    - Appends the function `print_i` to the list `flist`.
  - After the loop completes, it prints "End of flist" and returns the list `flist`.
- **Function** `main`:
  - Calls `make_functions()` and stores the returned list of functions in the variable `flist`.

– Iterates over each function `f` in `flist` and calls it.

Why Does It Print 3 Three Times at the End? Python uses **lexical scoping**, meaning that when a function is defined, it captures the environment in which it was created. The `print_i` function captures a reference to the variable `i` from the outer scope (the loop in `make_functions`). All three `print_i` functions in `flist` share the same reference to the variable `i`.

1. **Closures in Python Store References, Not Values**: The closure (the nested function `print_i`) stores a reference to the variable `i`, not its value at the time the function was created. All closures share the same reference.
2. **Variable `i` Is Evaluated When the Function Is Called, Not When It Is Defined**: When the loop in `main` calls each function, `i` is evaluated, and since `i` has the final value `3` after the loop in `make_functions` finishes, each function call prints `3`.

What if I still want it to print 1 2 3 instead of 3 3 3. To fix this and capture the value of `i` at each iteration, we can use a higher-order function to create a new scope that "locks in" the value of `i` at each iteration:

```
def create_printer(x):
    def print_x():
        print(x)
    return print_x

def make_functions():
    flist = []
    for i in [1, 2, 3]:
        print_i = create_printer(i)
        print_i()
        flist.append(print_i)
    print('End of loop')
    return flist

def main():
    flist = make_functions()
    for f in flist:
        f()

main()
```

Explanation:

- The function `create_printer(x)` creates a new function `print_x` that captures the current value of `x` at each iteration.
- `create_printer(i)` is called inside the loop to create a new function with the current value of `i`.
- Now, each function in `flist` has its own closure that references a different value of `x`.

Another example:

```
x = 10

def outer():
    x = 20
    return inner

def inner():
    print(x)

f = outer()
f()   # What does this print?
```

In real Python (lexical scope):

- inner() is defined at the top level of the file.
- At the point of definition, the visible x is the global one (x = 10).

- When inner() runs, Python looks up x lexically (i.e., in its definition environment), not at the call site.

So it prints 10.
If Python were dynamically scoped:

- The interpreter would resolve variable names based on the call stack, not the definition location.
- When f() runs, it was returned from outer(), which had its own local x = 20.
- Under dynamic scoping, the lookup for x would climb up the runtime call chain, find x = 20 in outer(), and use that.

So it would print 20.

5 Revisiting OOP Using Macros

If we want to build a house, then Python is the design rendering, theory is the tool, Racket is the prepared land, and our goal is to first learn how to use the tools, then refer to the design to construct our own house on the land.

In this chapter, you will first develop a deeper understanding of OOP by examining the design choices and trade-offs involved in its implementation. Second, you will learn how to utilize a powerful macro system to significantly expand the syntax and semantics of a programming language.

We've often talked about the big differences between imperative programming (where you tell the computer step by step what to do) and functional programming (where you rely on functions to get results). So, it might surprise you that by studying functions, you already have the tools to build a basic object-oriented system, similar to what you've seen in Python or other programming languages.

Let's revisit what a **closure** is: it's a structure that stores both a function and the values (of free variables) the function needs to work. Now, think about what an object is in object-oriented programming (OOP): it's a structure that holds data (called attributes) and is also linked to functions (called methods) that can manipulate this data. Do you see the similarity?

This approach in OOP is designed to keep the internal details (or private parts) of the object hidden while only allowing other code to interact with the data in specific ways (through a public interface). Unlike pure functions, methods always receive a special argument—an object that "calls" the method. This object's internal data can't be accessed from the outside but can be used inside its own methods. This argument in python is "**self**" and in C++ is "**this**".

Over time, people started describing objects in terms of "sending and receiving messages"—objects respond to these messages by either changing something inside or returning a value. And when you think about it, "receiving a message and returning a value" is pretty much what a function does, as shown in this simple example:

```
(define (point msg)
  (cond [(equal? msg 'x) 10]
        [(equal? msg 'y) -5]
        [else "Unrecognized message"]))
```

"*'x*" and "*'y*" do not have meaning unless the developer specifies; here they are **symbols**.

5.1 Symbols

Here's an example: The sleep function pauses the program's execution for a certain duration, but what is the unit of n? Seconds? Minutes? Hours? As users, we have no way of knowing unless the programmer provides documentation that clearly specifies the unit of n.

```
#lang racket

;sleep pauses the execution of the program
;for the amount of time given

(define (sleep n)
    ...)

;it's 12:00:00 right now, when do we resume?
(sleep 60)
```

The programmer could simply set the unit of n as seconds, and if we, as users, want to pause the program for an hour, we can just call snooze(3600). However, the function can be made more versatile by allowing multiple units of time. This way, users can choose the unit they want to work with, such as seconds, minutes, or hours.

In Python, this could be achieved by passing a string that represents the unit of time. For example, you could define the function as def snooze(n=0, unit='s'), where the unit parameter allows users to specify seconds, minutes, or other units.

Similarly, in Racket, you can achieve the same flexibility by using symbols to represent the time units. **Symbols allow you to create a local context for the time unit, making it clear what unit the function is operating with.** In this case, the **symbol** serves as a clear indicator of the unit being used.

By incorporating multiple unit options, the function becomes more powerful and user-friendly, enabling more flexible use.

```
;a time−unit(tu) is a unit of time, and is one of:
; 'h : a symbol representing an hour (60 × 60 seconds)
; 'm : a symbol representing a minute (60 seconds)
; 's : a symbol representing a second
; 'ms : a symbol representing a millisecond (1/1000 seconds)

;snooze pauses the execution of the program for the amount of time given.

; 'n' is a value in seconds

#lang racket
(define (snooze n tu)(let [(in-seconds (match tu
                                         ('h (× n 60 60))
                                         ('m (× n 60))
                                         ('s n)
                                         ('ms (/ n 1000))))]
                       (sleep in-seconds)))

(snooze 2 's)
```

What is the difference between symbols and identifiers?

A symbol **is a specific kind of value in a program, with meaning determined by the context of the language**. A symbol is atomic, meaning it is not a character string and doesn't contain anything. For example, if you evaluate 'x, the result will be 'x, indicating it is treated as a symbol.

An identifier, on the other hand, **is a syntactic category within the language itself**. Identifiers are typically used to denote memory locations, such as variables, special forms, or keywords. If you try to evaluate an identifier like n without defining it, you'll get an error like "unbound identifier in: n", because the identifier has not been associated with a value.

5.2 Classes as higher-order functions

```
(define (point msg)
  (cond [(equal? msg 'x) 10]
        [(equal? msg 'y) -5]
        [else "Unrecognized message"]))
```

This "point object" is quite basic: it has attributes but no methods, making it more like a C struct, with hard-coded attribute values. One way to improve this is by creating a point class, which acts as a template that defines both the attributes and methods for a specific type of object. In class-based object-oriented programming (OOP), every object is an instance of a class, inheriting its attributes and methods from the class definition.

Objects are created by calling a class constructor, a function that returns a new instance of that class, often initializing the object's attributes. To translate this into the context of functions, a constructor for Point is a function that takes two numbers and returns a function similar to the one we saw earlier, but this time the values 10 and 5 are replaced by the constructor's arguments.

```
(define (Point x y)
  (lambda (msg)
    (cond [(equal? msg 'x) x]
          [(equal? msg 'y) y]
          [else "Unrecognized message"])))

> (define p (Point 2 -100))
> (p 'x)
2
> (p 'y)
-100
```

This shows the relationship between closures and objects in this model. The function returned by Point has x and y as free variables, and their values are stored in the closure created when the constructor is called. Since x and y are local to the Point constructor, they can only be accessed through messages sent to the object, ensuring that they

remain private and encapsulated.

This encapsulation is made possible by lexical closures. If we were using a dynamically-scoped language instead, attribute encapsulation would break, especially when creating multiple instances of the same class, as demonstrated in this example:

```
(define p (Point 2 3))
(let ([x 10])
  (p 'x))  ;; This would cause issues in a dynamically—scoped language
```

In lexical scoping, when you define the Point function, the variables x and y inside the closure are bound to the values passed to the Point constructor (in this case, 2 and 3). These bindings are fixed in the environment where the closure was defined, and that environment doesn't change later. So when you create p using (Point 2 3), the x and y are internally tied to 2 and 3.

In the expression (p 'x), the closure accesses the value of x that was bound when the Point object was created, which is 2. The let ([x 10]) inside the block does not affect the x in the closure because of lexical scoping—the closure "remembers" the environment in which it was created.

In a dynamically-scoped language, variable bindings are resolved based on the current runtime environment, not where they were originally defined. This means that when you call (p 'x) inside the let block, the value of x would be looked up in the current scope.

Under dynamic scoping, the x inside the let block (which is 10) would be used instead of the x stored in the closure (which is 2). So when you evaluate (p 'x), the result would be 10 instead of 2, because x in the current environment shadows the original x in the closure.

5.3 Adding Methods to Our Class

Now, let's add two simple methods to our Point class. In Racket, functions are treated as first-class values, which means we can handle attributes and methods in the same way. A method is essentially just an attribute that happens to be a function!

One key feature that distinguishes methods from regular functions is that within a method, we expect to access all instance attributes of the calling object. Fortunately, this isn't a problem in our case. Take a look at the example below and see if you can figure out why accessing attributes within methods works as expected:

```
(define (Point x y)
  (lambda (msg)
    (cond [(equal? msg 'x) x]
          [(equal? msg 'y) y]
          [(equal? msg 'to-string)
           (lambda ()
             (format "(~a, ~a)" x y))]
          [(equal? msg 'distance)
           ;; Return the distance between this and another point.
           (lambda (other-point)
             (let ([dx (- x (other-point 'x))]
                   [dy (- y (other-point 'y))])
               (sqrt (+ (× dx dx) (× dy dy)))))]
          [else "Unrecognized message"])))

> (define p (Point 3 4))
> (p 'to-string)
#<procedure>
> ((p 'to-string))
"(3, 4)"
> ; Note that (p 'distance) returns a function, so the following is
> ; a nested function call.
> ((p 'distance) (Point 0 0))
5
```

In this example, methods like to-string and distance are able to access the instance attributes x and y directly. The closure created when the Point object is instantiated captures these values, allowing them to be used in any method within the object.

In programming, boilerplate code refers to sections of code that you have to write repeatedly, even though it doesn't change much. It's necessary for the program to work but can feel tedious because it involves repetitive tasks.

When implementing class-based objects using pure functions in Racket, you often need to manually define how messages (or method calls) are handled. For example, you need to specify what happens when an object receives a message (like a method call) and how it should respond. This is usually done with conditionals, like cond or if statements, that check which message was sent and handle it accordingly. If you have many classes or objects, this message-handling code gets repeated over and over, creating boilerplate code.

Moreover, you would also need to manage unrecognized messages, meaning if a message (method call) is sent to an object that it doesn't know how to handle, you'd have to include extra code to deal with that scenario (e.g., returning an error like "Unrecognized message"). Imagine for every class, you have to manually write code like this:

```
(cond
  [(equal? message 'greet) ...]   ; Handle 'greet' message
  [(equal? message 'age) ...]     ; Handle 'age' message
  [else "Unrecognized message"])   ; Handle unknown messages
```

This would have to be done for each new class, which quickly becomes repetitive. Instead, we want our code like this:

```
(class Person
  ; Attributes
  (name age likes-chocolate)

  ; Method: greet
  [(greet other-person)
    (string-append "Hello, "
                   (other-person 'name)
                   "! My name is " name ".")]

  ; Method: can-vote?
  [(can-vote?) (>= age 18)])
```

This demonstrates a much cleaner (the identifiers include all the information unlike the symbols), more concise way of defining class-like structures in Racket, making the syntax more manageable and easier to use. We will see the power of "**macro**" in Racket.

5.4 Pattern-based Macros

We typically think of functions as operations that take in values and return new ones. As essential components of programming, functions are versatile and powerful. However, it is easy to overlook their limitations: their syntax and behavior are determined by the programming language, not the programmer. For instance, no matter what function we write in Racket, we know that when it's called, the arguments will be evaluated from left to right in an eager manner, because that's how the Racket interpreter processes function calls.

In contrast, **a macro is a function that operates on the program's source code.** Instead of values, macros take code as input and generate new code as output. There is an extra step between parsing the source code and either generating machine code or evaluating it at runtime. This step, called macro expansion, **applies both built-in and user-defined macros to the abstract syntax tree (AST) to produce a new AST**. As a result, the AST used to generate machine code or execute the program may differ significantly from the one initially produced by the parser, depending on the macros applied.

This might sound a bit abstract, so why would we want to use macros? **One of their key uses is to introduce new syntax into a programming language.** We will first explore how to build a useful syntactic feature: the list comprehension. If you're familiar with Python, you've likely encountered list comprehensions, as Python borrowed this syntax from Haskell.

```
>>> [x + 2 | x <- [0, 10, -2]]
[2,12,0]
```

A list comprehension consists of three key components: an output expression, an identifier, and an input list. As a grammar rule, it looks like this:

```
<list-comp> = "[" <out-expr> "|" <id> "<-" <list-expr> "]"
```

Unfortunately, Racket doesn't have built-in list comprehension syntax, but fortunately, we can implement it ourselves using Racket's macro system! Here's an example of the kind of expression we aim to create in Racket (note that we use : instead of — for technical reasons):

```
>(list-comp (+ x 2) : x <- (list 0 10 -2))
'(2 12 0)
```

Let's first think about how to implement this functionality in Racket, without worrying about syntax. If you recall from the previous chapter, a list comprehension is essentially a map operation. Consider this:

```
>(map (lambda (x) (+ x 2)) (list 0 10 -2))
'(2 12 0)
```

Now, we can generalize the concept by doing some pattern matching:

```
; Putting our examples side by side...
(list-comp (+ x 2) : x <- (list 0 10 -2))
(map (lambda (x) (+ x 2)) '(0 10 -2))

; leads to the following generalization.
(list-comp <out-expr> : <id> <- <list-expr>)
(map (lambda (<id>) <out-expr>) <list-expr>)
```

The symbol like <list-expr> is typically referred to as **pattern variables** here in Racket. we'll follow a convention of naming these with enclosing angle brackets, but this is not required by Racket. This is a crucial step because it tells us, as programmers, what syntactic transformation needs to happen: each time the interpreter encounters a list comprehension, it should translate it into a map call. Now, we just need to instruct the Racket interpreter to perform this transformation, which we can achieve by writing a **pattern-based macro**:

```
(define-syntax list-comp
  (syntax-rules (: <-)
    [(list-comp <out-expr> : <id> <- <list-expr>)
     (map (lambda (<id>) <out-expr>) <list-expr>)]))
```

Let's break this down: The define-syntax form defines a new macro, similar to how define binds values. The first argument is the macro's name, list-comp in our case, and the second argument is the macro expression. We use syntax-rules to define pattern-based macros, which take **a list of literal keywords** (like : and < .- here) followed by **one or more pattern rules**. In each rule, **the first expression is the pattern to match, and the second is the template that specifies how the new syntax should be generated**. Right now, we only have one pattern, but we could expand this later.

```
>(list-comp (+ x 2) : x <- (list 0 10 -2))
'(2 12 0)
```

Although this looks like a regular function call, it's not! There are two phases involved to produce the result:

1. the list-comp expression is transformed into a map call by applying the syntax pattern rule. This is a code transformation—equivalent to typing (map (lambda (x) (+ x 2)) (list 0 10 -2)) directly.

2. the map expression is evaluated using normal function call semantics.

```
;call it like a function
>(list-comp 1 2 3)
list-comp: bad syntax in: (list-comp 1 2 3)

;call it using a number instead of a list
> (list-comp (+ x 2) : x <- 10)
map: contract violation
  expected: list?
  given: 10
```

The first error is a **syntax error**, meaning Racket cannot find a pattern rule that matches the expression provided. The second error, a **runtime error**, however, clearly shows that a syntax transformation is happening: although (list-comp (+ x 2) : x <- 10) may be syntactically correct, it expands to (map (lambda (x) (+ x 2)) 10), which results in the runtime error seen above because map expects a list, not a single value like 10.

During macro expansion, when Racket finds a match for the pattern, it binds the pattern variable to the corresponding expression and substitutes that expression wherever the variable appears in the template. This process is somewhat similar to how function calls work, where argument values are bound to parameter names and substituted into the function body. However, there's a key difference: **in macro expansion, the expressions are not evaluated before they are bound. This distinction is important because macros are designed to transform code, not evaluate it.** When an expression is bound to a pattern variable, the entire expression is bound, not its evaluated value.

literal keywords in a macro's syntax rule must appear exactly as they are in the syntax expression. These keywords function similarly to reserved words in other programming languages (e.g., else or : in Python) by providing structure to the syntax.

```
> (list-comp (+ x2) x (list 0 10 -2))
list-comp: bad syntax in: (list-comp (+ x2) x (list 0 10 -2))
```

5.4.1 Macros with multiple pattern rules

In Haskell, list comprehensions support optional filtering of the input list:

```
>>> [x + 2 | x <- [0, 10, -2], x >= 0]
[2, 12]
```

To implement this form of list comprehension in Racket, we can simply add an additional syntax rule to our macro definition, just like how we can introduce new pattern rules with define/match.

```
(define-syntax list-comp-if
  (syntax-rules (: <- if))
    ; The original pattern rule
    [(<rule-name> <out-expr> : <id> <- <list-expr>)
     (map (lambda (<id>) <out-expr>) <list-expr>)]

    ; The new pattern rule with a condition
    [(list-comp-if <out-expr> : <id> <- <list-expr> if <condition>)
     (map (lambda (<id>) <out-expr>)
          (filter (lambda (<id>) <condition>) <list-expr>))]))
```

For the rule-name in the pattern part, you can use a pattern variable or directly put the defined rule name there. With these two rules in place, we can now use both the basic and the extended version of list-comp:

```
> (list-comp-if (+ x 2) : x <- (list 0 10 -2))
'(2 12 0)

> (list-comp-if (+ x 2) : x <- (list 0 10 -2) if (>= x 0))
'(2 12)
```

Just like functions that use pattern matching in both Racket and Haskell, **macro pattern rules are checked from top to bottom, with the first matching rule being applied**. In this case, the rules are mutually exclusive, but in general, it's possible for two syntax pattern rules to overlap. Therefore, be mindful of this when writing your own macros to avoid unintended behavior!

5.5 Text-based vs. AST-based macros

The C macro system operates on the source text itself, the typesetting language LaTeX also rather than a parsed AST. In the macro example above we noted that we couldn't uses `"|"` or "," as literal keywords; **this is because they are special characters that affect the parsing of Racket code into an AST, and so have an effect even before the macro expansion phase**. This limittation aside, it is far easier and safer to define AST-based macros. In this section, we look at two examples of how text-based C macros can lead to unexpected pitfalls. First, here is a simple example of a C macro that takes one argument and multiplies its argument by 2:

```
//C code
#define double(x) 2 * x
```

When we use this macro on an integer literal or an identifier, things seem to work fine:

```c
#include <stdio.h>
#define double(x) 2 * x

int main(void) {
    int a = 10;
    printf("%d\n", double(100)); // Correctly prints 200.
    printf("%d\n", double(a)); // Correctly prints 20.
    return 0;
}
```

In each of the above invocations, the source text involving double is replaced: double(100) by 2 * 100 and double(a) by 2 * a. But consider this usage of the macro instead:

```c
printf("%d\n", double(5 + 5));
```

We might expect the result to again be 20, but that's not what happens! To understand why, we need to look at how substitution works in C macros. Similar to Racket, C macros don't evaluate their arguments before substitution; instead, the entire expression 5 + 5 is substituted into the macro body as-is. The issue arises because C macros perform text-based substitution. Since the macro body is textually 2 * x, the expression expands to 2 * 5 + 5, resulting in 15 being printed instead of 20.

In other words, because C macros rely on text-based substitution, the arguments aren't treated as whole expressions when inserted into the macro body—how they are processed depends on how the compiler parses the expanded text. In this case, the precedence rules for arithmetic expressions caused the 5 + 5 to be split up.

To avoid this issue, a common best practice when writing C macros is to always use parentheses around macro arguments in the macro body:

```c
#define double(x) 2 * (x)
```

Unlike C macros, Racket macros work on abstract syntax trees (ASTs), which prevents this problem altogether. By the time macro expansion happens, the structure of the Racket program is already determined. Macro expansion simply replaces parts of the AST that match a pattern with the corresponding template, ensuring that the expression structure remains intact. The only new subexpressions in the resulting AST are those explicitly defined in the macro itself or its arguments.

```
AST Root: *
  |__ Left Operand: 2
  |__ Right Operand: (+ 5 5)
```

Now, consider the following macro, which uses a temporary variable to swap two integers. "a" and "b" are both placeholders, similar to the pattern variables we've seen before. It appears to work correctly in this small program:

```c
#include <stdio.h>

#define swap(a, b) int temp = a; a = b; b = temp;

int main(void) {
    int x = 0, y = 10;
    swap(x, y);
    printf("x is now %d, y is now %d\n", x, y);
    return 0;
}
```

We encounter a problem when attempting to compile the following code:

```c
#include <stdio.h>

#define swap(a, b) int temp = a; a = b; b = temp;

int main(void) {
    int x = 0, y = 10;
```

```
    swap(x, y);
    swap(x, y);
    printf("x is now %d, y is now %d\n", x, y);
    return 0;
}
```

The compiler throws the error: redefinition of 'temp'. This happens because each call to the macro defines a new local variable temp, which leads to a redefinition error. To resolve this, C allows for local scopes to be introduced using curly braces. We can fix the issue by enclosing the macro body in curly braces:

```
#define swap(a, b) { int temp = a; a = b; b = temp; }
```

However, a new problem arises: what if one of the variables we want to swap is already named temp? Consider the following code:

```
#include <stdio.h>

#define swap(a, b) { int temp = a; a = b; b = temp; }

int main(void) {
    int x = 0, temp = 10;
    swap(x, temp);
    printf("x is now %d, temp is now %d\n", x, temp);
    return 0;
}
```

Although this code compiles, the output is:

```
x is now 0, temp is now 10
```

The variables were not swapped! To understand why, let's look at the literal substitution of swap(x, temp):

```
{ int temp = x; x = temp; temp = temp; }
```

Here, all references to temp within the macro refer to the block-local temp variable, not the temp initialized to 10 in main. The local temp is set to the value of x (0), x is assigned that value (0), and then temp is reassigned to itself. Once the block finishes, neither x nor temp have changed.

A third approach is to require the declaration of temp before using the macro. This works correctly but adds an extra step for the user:

```
#include <stdio.h>

#define swap(a, b) { temp = a; a = b; b = temp; }

int main(void) {
    int x = 0, y = 10, temp;
    swap(x, y);
    printf("x is now %d, y is now %d\n", x, y);
    return 0;
}
```

The behavior of these macros depends not only on how they are defined but also on where they are used. If this sounds familiar, it should—**C macros follow dynamic scoping**. This behavior arises because identifiers in C macros are simply substituted as text during macro expansion.

Although Racket macros are based on AST transformations, it's reasonable to expect that individual identifiers might still be substituted as-is during macro expansion. Let's explore this with a simple example:

```
(define-syntax add-temp
  (syntax-rules ()
    [(add-temp <x>)
     (+ <x> temp)]))
```

At first glance, it seems like temp is undefined within the add-temp macro, similar to how temp was undeclared in our earlier C macro example. The question arises: what happens when we try the following?

```
(let× ([temp 10])
  (add-temp 100))
```

If Racket used literal textual substitution of identifiers, this would expand to:

```
(let× ([temp 10])
  (+ 100 temp))
```

This is valid Racket code, and we'd expect it to evaluate to 110. However, that's not what happens! Instead, we get:

```
ERROR: temp: undefined;
cannot reference an identifier before its definition.
```

This shows that the temp in the add-temp macro template follows lexical scoping and cannot be dynamically bound where the macro is used. This is true for all Racket macros: free identifiers in macros follow lexical scope, meaning they are bound to whatever is in scope where the macro is defined, not where it is used.

Identifiers defined inside a macro body remain local to the macro, and cannot be accessed outside, even though a direct textual substitution might suggest otherwise. For example:

```
(define-syntax defs
  (syntax-rules ()
    [(defs)
     (begin ; We use begin to enclose multiple expressions
       (define x 1)
       (define y (+ x 10))
       ; x and y are both in scope here.
       (+ x y))]))

(defs) ; This evaluates to 12
x ; This is an error! x and y are not in scope.
```

Contrast this behaviour against simply using the begin instead of our defs macro:

```
; This still evaluates to 12
(begin
  (define x 1)
  (define y (+ x 10))
  (+ x y))

; But now x is defined as well! The following evaluates to 1.
x
```

5.6 Macro ellipses

It is common to require a macro that can be applied to an arbitrary number of expressions. This presents a challenge when writing macro pattern rules, as it is not feasible to explicitly write a pattern variable for each expected expression. Instead, the ellipsis `...` can be used in a pattern: `<pat> ...` matches zero or more instances of the pattern `<pat>`, which could be a pattern variable or a more complex pattern itself.

Here is an example of using ellipses in a recursive macro that implements `cond` in terms of `if`. Recall that a sequence of `else if` expressions can be rewritten as nested `if` expressions:

```
(cond [c1 x1]
      [c2 x2]
      [c3 x3]
      [else y])

; as one cond inside an if...
(if c1
    x1
    (cond [c2 x2]
          [c3 x3]
```

```
          [else y]))

; eventually expanding to...
(if c1
    x1
    (if c2
        x2
        (if c3
            x3
            y)))
```

Here is the macro that achieves this behavior:

```
(define-syntax my-cond
  (syntax-rules (else)
    [(my-cond) (void)]
    [(my-cond [else <val>]) <val>]
    [(my-cond [<test> <val>] <next-pair> ...)
     (if <test> <val> (my-cond <next-pair> ...))]))
```

This example illustrates two important concepts regarding Racket's pattern-based macros. First, the macro defines not just a simple syntax pattern, but a nested one. The first pattern rule will match `(my-cond [else 5])`, but not `(my-cond else 5)` — parentheses are significant.

The second concept is demonstrated by the `<next-pair> ...` part of the second pattern, which binds all arguments after the first. For instance, here's an example usage of the macro and one step of macro expansion, which provides insight into the recursive workings of the macro:

```
(my-cond [c1 x1]
         [c2 x2]
         [c3 x3]
         [else y])

; <test> is bound to c1, <val> is bound to x1,
; and <next-pair> ... is bound to all of
; [c2 x2], [c3 x3], and [else y]

(if c1
    x1
    (my-cond [c2 x2] [c3 x3] [else y]))
```

A crucial note is that the ellipsis should not be thought of as a separate entity, but rather as a modifier for `<next-pair>`. The ellipsis is "bound" to `<next-pair>` and must always appear together with it in the template. The binding is bidirectional: `<next-pair>` cannot appear without the ellipsis either. A common beginner mistake is treating the ellipsis as an independent identifier representing the remaining arguments:

```
(define-syntax my-cond-bad
  (syntax-rules (else)
    [(my-cond-bad [else <val>]) <val>]
    [(my-cond-bad [<test> <val>] ...)
     ; This would make sense if ... was a variable representing the
     ; remaining arguments, but it isn't.
     ; Instead, the following rule is a syntax error, as the ellipsis
     ; cannot be used independently of the pattern it's bound to.
     (if <test> <val> (my-cond-bad ...))]))
```

To summarize, **a pattern variable modified by an ellipsis is not bound to a single expression, but rather to an entire sequence of expressions**. In the next section, we will explore how to leverage this sequence effectively.

5.7 Revisiting Objects

Now that we have explored how to define macros in Racket, let's revisit our basic implementation of objects and classes. As a starting point, consider the following simplified version of the `Point` class, which contains only two attributes and no methods:

```
(define (Point x y)
  (lambda (msg)
    (cond [(equal? msg 'x) x]
          [(equal? msg 'y) y]
          [else "Unrecognized message!"])))
```

The objective is to abstract away the details involved in defining a constructor function and handling messages, so that new classes can be declared more easily. Since we are interested in **defining new identifiers for class constructors**, using functions for abstraction is not sufficient. To solve this, we use macros to introduce a new syntactic form, `my-class`:

```
; This expression should expand into the above definition.
(my-class Point
          (x y))
```

By performing a pattern-matching process similar to what we did for `list-comp`, we can write the skeleton of a macro that accepts an arbitrary number of attribute names, not just two:

```
(define-syntax my-class
  (syntax-rules ()
    [(my-class <class-name> (<attr> ...))
     (define (<class-name> <attr> ...)
       (lambda (msg)
         (cond ???
               [else "Unrecognized message!"])))]))
```

What is missing are the additional expressions in the `cond` that handle messages corresponding to each attribute name. Consider the first attribute, `x`. For this attribute, we want to generate the following code:

```
[(equal? msg 'x) x]
```

To achieve this, we need a way to convert an identifier into a symbol, which can be done using the `quote` form:

```
[(equal? msg (quote x)) x]
```

For the expression `(my-class Point (x y))`, we want similar code to appear for both `x` and `y`:

```
[(equal? msg (quote x)) x]
[(equal? msg (quote y)) y]
```

The question then becomes: how do we generate one of these expressions for each attribute in `<attr> ...`? Essentially, we need to repeat the same pattern for an arbitrary number of attributes, depending on how many are provided. Fortunately, **macro ellipses** support exactly this kind of repetition. Here is the completed macro:

```
(define-syntax my-class
  (syntax-rules ()
    [(my-class <class-name> (<attr> ...))
     (define (<class-name> <attr> ...)
       (lambda (msg)
         (cond [(equal? msg (quote <attr>)) <attr>]
               ; Repeat the expression once for each attribute in <attr> ...,
               ; replacing occurrences of <attr> accordingly.
               ...
               [else "Unrecognized message!"])))]))
```

5.8 Adding Methods

Next, we will extend our macro to allow the definition of methods. While we previously emphasized that methods are just another kind of attribute, there is one significant difference: for methods, we not only need to provide the name but also the body of the method. We will use the following syntax, similar to Racket's macro pattern-matching syntax:

```
(my-class <class-name> (<attr> ...)
          (method (<method-name> <param> ...) <body>) ...)
```

As before, the easiest way to construct the pattern-based macro is to create both the syntactic form we want to write and the expression we want it to expand into. Using our ongoing example of the `Point` class:

```
; The code we want to write:
(my-class Point (x y)
  (method (distance other-point)
    (let× ([dx (- x (other-point 'x))]
           [dy (- y (other-point 'y))])
      (sqrt (+ (× dx dx) (× dy dy))))))

; The code we want it to expand into:
(define (Point x y)
  (lambda (msg)
    (cond [(equal? msg 'x) x]
          [(equal? msg 'y) y]
          [(equal? msg 'distance)
           (lambda (other-point)
             (let× ([dx (- x (other-point 'x))]
                    [dy (- y (other-point 'y))])
               (sqrt (+ (× dx dx) (× dy dy)))))])))
```

After carefully comparing these two expressions, the pieces naturally fall into place. Here's the complete macro:

```
(define-syntax my-class
  (syntax-rules (method)
    [(my-class <class-name>
       ; This ellipsis is paired with <attr>
       (<attr> ...)
       ; This ellipsis is paired with <param>
       (method (<method-name> <param> ...) <body>)
       ; This ellipsis is paired with the entire previous pattern
       ...)
     (define (<class-name> <attr> ...)
       (lambda (msg)
         (cond [(equal? msg (quote <attr>)) <attr>]
               ...
               [(equal? msg (quote <method-name>))
                (lambda (<param> ...) <body>)]
               ...
               [else "Unrecognized message!"])))]))

(my-class Point (x y)
          (method (distance other-point)
                  (let× ([dx (- x (other-point 'x))]
                         [dy (- y (other-point 'y))])
                    (sqrt (+ (× dx dx) (× dy dy))))))

(define p1 (Point 3 4))
(define p2 (Point 4 5))
((p1 'distance) p2)
```

5.9 The Object __dict__

You may have noticed that the repeated `(equal? msg <symbol>)` clauses in the `cond` function effectively implement a (slow!) dictionary lookup, where an attribute's name is mapped to its corresponding value for that object. In Python, each object has a special attribute called `__dict__`, which refers to its dictionary of attributes (but not methods—this will be discussed later):

```
class A:
    def __init__(self, x):
        self.x = x
        self.y = 10
        self.z = 'hello!'

>>> a = A(True)
```

```
>>> a.__dict__
{'x': True, 'y': 10, 'z': 'hello!'}
```

To improve the performance of our `my-class` macro, we can utilize Racket's built-in hashing data structure to store both the attributes and methods:

```
#lang racket
(define-syntax my-class
  (syntax-rules (method)
    [(my-class <class-name> (<attr> ...)
               (method (<method-name> <param> ...) <body>) ...)
     (define (<class-name> <attr> ...)
        (let ([__dict__ ; Use the same name as Python
                  (make-immutable-hash
                   (list (cons (quote <attr>) <attr>)
                         ...
                         (cons (quote <method-name>)
                               (lambda (<param> ...) <body>))
                         ...
                         ))])
          ; Look up the given attribute in the object's dictionary.

       (lambda (msg)
         (cond [(equal? msg (quote <attr>)) <attr>]
               ...
               [(equal? msg (quote <method-name>))
                (lambda (<param> ...) <body>)]
               ...
               [(equal? msg (quote __dict__)) __dict__]
               [else "Unrecognized message!"])))]))

(my-class Point (x y)
          (method (distance other-point)
                  (let× ([dx (- x (other-point 'x))]
                         [dy (- y (other-point 'y))])
                    (sqrt (+ (× dx dx) (× dy dy))))))

(define p1 (Point 3 4))
(p1 '__dict__)
```

5.10 The Problem of `self`

Thus far, our implementation of classes seems functional, but it has a significant limitation: we cannot explicitly reference the calling object within a method. Consider the following Python example:

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def size(self):
        return math.sqrt(self.x ** 2 + self.y ** 2)

    def same_radius(self, other):
        return self.size() == other.size()
```

Our current implementation can handle the initialization method `__init__` (which is done automatically), and it can also manage simple attribute access for `x` and `y`, as these identifiers are bound within the closure returned by the class constructor. However, the method `self.same_radius()` presents a problem:

```
(my-class Point (x y)
  (method (size) (sqrt (+ (× x x) (× y y))))
  (method (same-radius other)
```

```
    (equal? ??? ((other 'size)))))
```

Unlike `x` and `y`, the identifier `size` is not directly available in the closure. Instead, `'size` is a key in the dictionary, but there is no automatic way to reference it! To address this, we will adopt a strategy that makes `self` an accessible identifier, allowing it to reference the calling object:

```
(method (same-radius other)
  (equal? ((self 'size)) ((other 'size))))
```

To guide our design and implementation, let's first revisit how `self` works in Python. You are likely familiar with the most visible aspect: `self` is an explicit parameter included in every instance method of a class. Technically, Python interprets the first parameter as being bound to the calling object, regardless of the actual parameter name. We can write the following Racket code without modifying our macro at all:

```
(my-class Point (x y)
  (method (size self) (sqrt (+ (× (self 'x) (self 'x))
                               (× (self 'y) (self 'y)))))
  (method (same-radius self other)
    (equal? ((self 'size) self) ((other 'size) other))))
```

Take a close look at the last line—what exactly is happening here? The goal is for `(self 'size)` to return a function that takes no arguments, allowing us to call it simply by enclosing it in parentheses. However, in this case, `(self 'size)` returns a function that expects a single argument, `self`, which means we have to explicitly pass that argument. The same applies when calling `(other 'size)`. This redundancy is unnecessary: since we are already calling `(self 'size)`, we know the calling object and shouldn't need to pass it again.

In Python, this is handled automatically. An instance method call like `self.size()` binds the value to the left of the dot (i.e., `self`) to the first parameter of the method, while the arguments inside the parentheses are bound to the remaining method parameters. So, how can we modify our macro to add this "automatic `self` binding"?

Let's again look to Python for inspiration. Recall that when we discussed Python's `__dict__`, we noted that an object's `__dict__` stores its data attributes, but not its methods. This is because methods are stored in the `__dict__` of the class itself:

```
>>> Point.__dict__
# something that's a wrapper around a dictionary
>>> Point.__dict__['size']
<function Point.size at 0x04986300>
>>> Point.__dict__['same_radius']
<function Point.same_radius at 0x04986228>
```

This gives us a direction for implementing methods in our Racket macro:

1. Store methods in a separate class-level dictionary shared among all instances.
2. When methods are looked up in this dictionary, instead of returning the method as-is, we bind the first parameter of the method to the calling object and return the newly bound method. In other words, we partially apply the method to the first argument.

Here's an implementation of the first step, showing the macro template:

```
(define-syntax my-class
  (syntax-rules (method)
    [(my-class <class-name> (<attr> ...)
               (method (<method-name> <param> ...) <body>) ...)

     ; This is a dictionary of the methods associated with a class.

     (define (<class-name> <attr> ...)
       (letrec
           ([self__dict__
             ; The object's __dict__ now only stores non-function
             ; attributes. Methods are looked up in class__dict__.
             (make-immutable-hash
              (list (cons (quote <attr>) <attr>) ...))]
            [class__dict__
```

```
            (make-immutable-hash (list
                                  (cons (quote <method-name>)
                                        (lambda (<param> ...) <body>))
                                  ...))])

          (lambda (msg)
           (cond
             ; Note the lookup order (first object, then class).
             ; This is the same as Python.
             [(hash-has-key? self__dict__ msg)
              (hash-ref self__dict__ msg)]
             [(hash-has-key? class__dict__ msg)
              (hash-ref class__dict__ msg)]
             [else "Unrecognized message!"])))))]))

(my-class Point (x y)
          (method (size self) (sqrt (+ (× (self 'x) (self 'x))
                                       (× (self 'y) (self 'y)))))
          (method (same-radius self other)
                  (equal? ((self 'size) self) ((other 'size) other))))
                  ;(equal? ((self 'size)) ((other 'size)) )))

(define p1 (Point 3 4))
((p1 'size) p1)
```

In `letrec` expression, all variable locations (memory slots for each identifier, or `id`) are created first, before any expressions are evaluated. This means that every `id` is bound and accessible in all val-exprs as well as in the body of the `letrec` form.

```
(letrec ([id val-expr] ...) body ...)
```

Once the locations for all the `ids` are created, the expressions corresponding to each `id` are evaluated from left to right. Importantly, each `id` is initialized immediately after its val-expr is evaluated. This allows the defined variables to refer to each other within the val-exprs, enabling mutual recursion or complex interdependencies between variables.

Consider the following example of `letrec`:

```
(letrec ([a (+ 1 b)]
         [b 10])
  a)
```

Here's what happens:

1. First, memory locations for `a` and `b` are created but not yet initialized.
2. The val-expr for `a` is evaluated. In this expression, `b` is already bound (even though it is not initialized yet), allowing `a` to reference `b`.
3. The val-expr for `b` is evaluated next, assigning `b` the value `10`.
4. Once both val-exprs are evaluated, the body (`a`) is evaluated to `11` (i.e., `1 + 10`).

The function `hash-has-key?` checks whether a given key exists in a hash table. In Racket, hash tables are used to store key-value pairs, and `hash-has-key?` is useful when you want to verify if a particular key is present in the hash before attempting to retrieve its value.

The function `hash-ref` retrieves the value associated with a given key from a hash table. If the key is found, `hash-ref` returns the corresponding value; otherwise, it can raise an error or return a default value if specified.

This piece of code is a bit longer, but conceptually, we have simply divided the original object dictionary into two: one for data attributes and one for methods. This doesn't yet solve the second issue. We need a reference to the object itself, which is the entire `lambda` expression, so we bound self to the entire **lambda**:

```
#lang racket
(define-syntax my-class
  (syntax-rules (method)
```

```
    [(my-class <class-name> (<attr> ...)
               (method (<method-name> <param> ...) <body>) ...)

     ; This is a dictionary of the methods associated with a class.

     (define (<class-name> <attr> ...)
       (letrec
           ([self__dict__
             ; The object's __dict__ now only stores non-function
             ; attributes. Methods are looked up in class__dict__.
             (make-immutable-hash
              (list (cons (quote <attr>) <attr>) ...))]
            [class__dict__
             (make-immutable-hash (list
                                   (cons (quote <method-name>)
                                         (lambda (<param> ...) <body>))
                                   ...))]
         [self
          (lambda (attr)
           (cond
             ; Note the lookup order (first object, then class).
             ; This is the same as Python.
             [(hash-has-key? self__dict__ attr)
              (hash-ref self__dict__ attr)]
             [(hash-has-key? class__dict__ attr)
              ((hash-ref class__dict__ attr) self)]
             [else "Unrecognized message!"]))])self))]))

(my-class Point (x y)
          (method (size self) (sqrt (+ (× (self 'x) (self 'x))
                                       (× (self 'y) (self 'y)))))
          (method (same-radius self other)
                  (equal? ((self 'size)) ((other 'size)) )))

(define p1 (Point 3 4))
(p1 'size)
```

This method can solve functions like size which only has a single input parameter that is the object itself. But how about the method such as `same-radius`? how to call the function in the manner of `((p1 'same-radius) p2)`? To fix that, we introduce a higher-order function that takes a function `f` with `n + 1` arguments and a value `x`, and returns a new function `g` that takes `n` arguments, such that `g(x1, x2, ..., xn)` is equivalent to `f(x, x1, x2, ..., xn)`.

```
#lang racket
(define-syntax my-class
  (syntax-rules (method)
    [(my-class <class-name> (<attr> ...)
               (method (<method-name> <param> ...) <body>) ...)

     ; This is a dictionary of the methods associated with a class.

     (define (<class-name> <attr> ...)
       (letrec
           ([self__dict__
             ; The object's __dict__ now only stores non-function
             ; attributes. Methods are looked up in class__dict__.
             (make-immutable-hash
              (list (cons (quote <attr>) <attr>) ...))]
            [class__dict__
             (make-immutable-hash (list
                                   (cons (quote <method-name>)
                                         (lambda (<param> ...) <body>))
                                   ...))]
         [self
```

```
          (lambda (attr)
           (cond
             ; Note the lookup order (first object, then class).
             ; This is the same as Python.
             [(hash-has-key? self__dict__ attr)
              (hash-ref self__dict__ attr)]
             [(hash-has-key? class__dict__ attr)
              ;(hash-ref class__dict__ attr)]
              (lambda args (apply (hash-ref class__dict__ attr) (cons self args)))]
             [else "Unrecognized message!"]))])self))]))

(my-class Point (x y)
          (method (size self) (sqrt (+ (× (self 'x) (self 'x))
                                       (× (self 'y) (self 'y)))))
          (method (same-radius self other)
                  (equal? ((self 'size)) ((other 'size)) )))

(define p1 (Point 3 4))
(define p2 (Point 5 6))
((p1 'same-radius) p2)
((p1 'size)) ;why two nested parentheses
```

In Racket, (lambda args body) is shorthand for a procedure that takes any number of arguments and binds them as a list to the name args. It's equivalent to:

```
(lambda (args ...) body)
```

but with variable arity — it automatically packs all arguments into one list called args.

For example:

```
((lambda args args) 1 2 3)  ; => '(1 2 3)
```

So here, args is a list of the arguments passed when the method is called.

6 Stream and Backtracking

6.1 Stream

The first half of this chapter provides a glimpse into the flexibility and power of macros: by modifying the syntax of our base language, we enable new paradigms, which in turn allow us to craft new idioms and techniques for building software. Now, we will shift focus to another common application of macros: **manipulating control flow to bypass Racket's default eager evaluation of function calls**.

Our initial step in this direction will be to implement streams, **a "lazy" counterpart to the familiar list data type**. To start, let's recall the recursive definition of a list:

- A list is either empty, or
- A value "cons'd" with another list.

In Racket, `cons` is a function, which means that all lists constructed using `cons` are evaluated eagerly—each element is evaluated as the list is created. However, this is often unnecessary: when traversing a list, we typically only need to access one element at a time, with no immediate need to evaluate the remaining elements. This motivates the creation of streams in Racket, which adhere to the following recursive definition:

- A stream is either empty, or
- A thunk wrapping a value "cons'd" with a thunk wrapping another stream.

As discussed in the previous chapter, thunks are used here to delay the evaluation of stream elements until they are explicitly needed. While we could manually pass thunks into the built-in `cons`, we can use macros to simplify the syntax and eliminate boilerplate code.

```
; Define the empty stream value and a check for empty streams.
(define s-null 's-null)
(define (s-null? stream) (equal? stream s-null))

#|
(s—cons <first> <rest>) → stream?
<first>: any/c?
<rest>: stream?
E.g., s—null or another s—cons expression.

Creates a stream where the first value is <first>, and the subsequent
items are those in <rest>. Unlike a standard list, both <first> and <rest>
are wrapped in thunks, delaying their evaluation.

Note: s—cons is a MACRO, not a function!
|#
(define-syntax s-cons
  (syntax-rules ()
    [(s-cons <first> <rest>)
     (cons (thunk <first>) (thunk <rest>))]))

(define (s-first stream) ((car stream)))
(define (s-rest stream) ((cdr stream)))

#|
(make—stream <expr> ...) → stream?
<expr> ... : any/c?

Returns a stream containing the specified values.
This is also a macro. (Why?)
|#
(define-syntax make-stream
  (syntax-rules ()
    [(make-stream) s-null]
    [(make-stream <first> <rest> ...)
     (s-cons <first> (make-stream <rest> ...))]))
```

`car` and `cdr` work on pairs (or cons cells) and don't evaluate the contents of those pairs. Instead, `first` and `rest` are part of Racket's list functions that work on eagerly evaluated lists. When you use first or rest, they assume the list is already fully evaluated.

In Racket (and Lisp-like languages), there is a fundamental difference between a **pair** (or cons cell) and a **list** containing two elements, even though they might look similar at first glance. Here's a detailed explanation of the difference:

A **pair** is a structure that contains two values: a **car** (the first value) and a **cdr** (the second value). Pairs are created using the `cons` function, which takes two arguments: the first becomes the **car**, and the second becomes the **cdr**. A pair is not necessarily a list, especially if the **cdr** is not a proper list (e.g., it could be a single value, not another list).

```
(cons 1 2)      ; This creates the pair (1 . 2), not a list
```

This pair has `1` as the **car** and `2` as the **cdr**. It is **not a list**, because the second element is not a proper list but an atom (`2`).

A **list** in Racket is a sequence of elements where each element is stored in the **car** of a cons cell, and the **cdr** points to either another cons cell (for the next element) or the empty list (`'()`), which marks the end of the list. Lists are essentially **linked structures** made up of pairs where the **cdr** of each cons cell is either another cons cell or `null` (the empty list `'()`).

```
(cons 1 (cons 2 '()))   ; This creates the list (1 2)
```

Here, the first `cons` creates a pair with `1` as the **car** and another list as the **cdr**. The second `cons` creates a pair with `2` as the **car** and `'()` (the empty list) as the **cdr**. This marks the end of the list, making this a proper list.

- **Structure**:
  - A **pair**: `(a . b)` has two elements: a **car** (`a`) and a **cdr** (`b`), where `b` can be any value (not necessarily a list).
  - A **list**: In a proper list, the **cdr** of each cons cell is either another cons cell or the empty list `'()`.
- **Terminators**:
  - **Pairs** don't require any particular terminator for the **cdr**. It can be any value.
  - **Lists** are terminated with `'()` (the empty list), which defines the end of a list.
- **Visual Difference**:
  - A **pair** is displayed as `(car . cdr)` when printed.

    ```
    Example: (1 . 2) is a pair.
    ```

  - A **list** is printed as a sequence of elements, like `(1 2)`, without a dot.
- **Behavior with `car` and `cdr`**:
  - With a **pair**, `cdr` can return any value (not necessarily a list), so it's not guaranteed to behave like a list.
  - With a **list**, `cdr` should always return the rest of the list or `'()`.

The elegance of this macro-based stream implementation is that its public interface is identical to that of Racket's built-in lists:

```
> (define s1 (s-cons 3 (s-cons 4 (s-cons 5 s-null))))
> (s-first s1)
3
> (s-first (s-rest s1))
4
> (s-first (s-rest (s-rest s1)))
5
> (s-null? (s-rest (s-rest (s-rest s1))))
#t
```

The primary difference, however, is in how these streams are created:

```
> (define items1 (list 1 2 (error "error")))
ERROR error

> (define items2 (make-stream 1 2 (error "error")))
> (s-first items2)
1
```

In the case of `items1`, the entire list is evaluated eagerly, leading to an error when it is constructed. By contrast, `items2`, which is built using `make-stream`, delays the evaluation of its elements. The error in the stream is not triggered until it is explicitly accessed.

6.2 Infinite Stream

6.2.1 Infinite Stream in Haskell

Haskell employs lazy evaluation for all of its function calls, meaning that the cons operation `(:)` is already lazy. Consequently, in Haskell, the list data type we've been using all along are inherently streams:

```haskell
x = [1, 2, error "error"]
head x
1
```

We will briefly examine one of the fascinating consequences of using streams to delay evaluation of sequence elements: constructing and manipulating infinite sequences. We will utilize Haskell's built-in list data types to highlight the fact that Haskell's lists are, in fact, streams.

```haskell
myRepeat x = x : myRepeat x
nats = 0 : map (+1) nats

main = do
         print (take 3 (myRepeat 6))
         print (take 5 nats)
```

What is happening with the `nats` definition? How does `take 5 nats` work? To understand this, let's consider a simpler example: `head nats`. We can trace its evaluation using the central concept of pure functional programming: substitution.

1. In Haskell, `head` is defined via pattern-matching as `head (x:_) = x`.
2. The list `nats` is defined as `0 : map (+1) nats`. When we match `nats` with the `head` definition, we find `x = 0`, and thus 0 is returned. Note that `head` ignores the remainder of the list, so we don't need to evaluate the recursive part.

Now let's consider something slightly more complex: `head (tail nats)`. Again, we use substitution, taking into account the definition of `tail`, which is `tail (_:xs) = xs`.

```haskell
head (tail nats)
-- We cannot call `head` until we can pattern-match on (x:_).
-- To achieve this, we'll need to expand `(tail nats)`:
head (tail (0 : map (+1) nats))
-- We can now apply pattern-matching using `tail`:
head (map (+1) nats)
-- Next, we use the definition of `map`:
-- map _ [] = []
-- map f (x:xs) = (f x) : (map f xs)
-- We expand `nats` again:
head (map (+1) (0 : map (+1) nats))
-- This allows us to pattern-match: x = 0, and xs = map (+1) nats.
head ((+1) 0 : map (+1) (map (+1) nats))
-- At this point, we can apply `head` to the result:
(+1) 0
-- The final result is:
1
```

Thus, the first two elements of `nats` are 0 and 1. The expansion we performed suggests what happens when further elements are accessed. The part we discarded, `map (+1) (map (+1) nats)`, continues adding one repeatedly. In subsequent recursive expansions of `nats`, the `map (+1)` function accumulates, producing increasingly larger numbers.

Since Haskell's higher-order list functions operate recursively, they are equally applicable to infinite lists. For example:

```haskell
nats = 0 : map (+1) nats
squares = map (\x -> x * x) nats
evens = filter (\x -> x `mod` 2 == 0) nats

main = do
        print (take 3 evens)
        print (take 4 squares)
```

To conclude, here's a delightful example that generates the Fibonacci sequence:

```haskell
fibs = 1 : 1 : zipWith (+) fibs (tail fibs)

main = do
        print (take 3 fibs)
```

6.2.2 Infinite Stream in Racket

In this section, we will explore how to create infinite streams in Racket using thunks and recursion (like Haskell). Streams allow you to handle potentially infinite sequences of data in a lazy manner, meaning values are computed only when needed.

In Racket, we can define an infinite stream using *thunks*, which are functions that take no arguments and delay evaluation. Below is the code to define an infinite stream of natural numbers:

```racket
(define (nature-nums x)
  (s-cons x (nature-nums (+ x 1))))
```

The `nature-nums` binds a stream starting from `x`, where each element is a thunk (a delayed value), and the tail of the stream is recursively computed by calling `nature-nums` with `x+1`.

We define a function `s-cons-take` to take the first `n` elements from a stream and return them as a list:

```racket
(define/match (s-cons-take s n)
  [('s-null _) '()]          ; Base case: empty stream
  [(_ 0) '()]                ; Base case: when n = 0, return an empty list
  [((cons x xs) n)            ; Recursive case: take the first n elements
   (cons (x) (s-cons-take (xs) (- n 1)))])

(define s1 (nature-nums 0))   ; Define the infinite stream
(s-cons-take s1 6)            ; Output: '(0 1 2 3 4 5)
```

let's further write a map function that maps a function over a stream `s-cons-map`, applying it to each element lazily:

```racket
(define (s-cons-map f s)
  (match s
    ['s-null '()]
    [(cons x xs) (s-cons (f (x)) (s-cons-map f (xs)))]
    ))

(s-cons-take (s-cons-map (lambda (x) (+ x 1)) s1) 10)   ; Output: '(1 2 3 4 5 6 7 8 9 10)
```

We can also define `s-cons-filter` to filter elements of a stream based on a condition:

```racket
(define (s-cons-filter f s)
  (match s
    ['s-null '()]
    [(cons x xs) (if (f (x))
                     (s-cons (x) (s-cons-filter f (xs)))
```

```
                    (s-cons-filter f (xs)))]))

(s-cons-take (s-cons-filter (lambda (x) (= (modulo x 3) 0)) (nature-nums 0)) 10)  ; Output: '(0 3 6 9
    12 15 18 21 24 27)
```

What will happen if I modify `(s-cons (x) (s-cons-filter f (xs)))` to `(cons (x) (s-cons-filter f (xs)))`? There will be an **out of memory** runtime error? Think why?

6.3 the ambiguous operator `-<` and `next`

In the previous section, we explored a simple use of macros to implement streams, a lazy data structure used to separate the creation of data from its consumption. In this chapter, we will further examine this concept in a different context, where the data we create is specified by expressions using non-deterministic choice, and the consumption of this data is made explicit through an impure function, `next!`. This implementation introduces an important programming language theory concept: **continuations**, which represent the control flow at a given point in a program's execution.

In Python, you probably have seen a program below:

```
list1 = [1,2,3,4]
itr1 = iter(list1)    # create iterator object

print (next(itr1))
print (next(itr1))
```

So, `-<` is used to create an iterator object like the one in Python, and function `next!` works similar to the `next` function in the above python program.

Since you may not be familiar with these language features, we will first explain the functionality of two key expressions before discussing their implementation. The primary one is a macro called `-<` (pronounced "amb", short for *ambiguous*) which behaves as follows:

- `-<` takes any number of argument subexpressions, representing possible choices for the entire `-<` expression.
- If there is at least one argument, `-<` evaluates and returns the value of the first argument.
- If there are multiple arguments, `-<` stores a "choice point" containing the remaining arguments, which can be accessed one at a time by calling the function `next!`.
- After all arguments have been evaluated, subsequent calls to `next!` return a special constant, `DONE`.

At this point, we are describing the denotational semantics of `-<` and `next!`, but not their operational semantics. Before we delve into the implementation, it is important to first understand the expected behavior. Below is an example of how we would like to use these macros:

```
> (-< 1 2 (+ 3 4))   ; First evaluates to 1
1
> (next!)            ; Calls to (next!) return other choices
2
> (next!)
7
> (next!)            ; Done after all choices
'done
```

We will now write a skeleton implementation for both `-<` and `next!`. The inspiration—likely shared from the previous example—is that of a self-modifying stream: a sequence of values accessed one at a time using `next!`, but using mutation to track the next item.

At a high level, the implementation will use a private stream-like variable `choices`, initialized by `-<` and accessed (and mutated) by `next!`:

```
(define choices (void))
;(void): The (void) function in Racket returns a "void" value, which is generally used to indicate that a function doesn't
    produce a meaningful result.
(define (set-choices! val) (set! choices val))

```

```
; A constant representing the state of having "no more choices".
(define DONE 'done)

(define-syntax -<
  (syntax-rules ()
    ; Single option: reset `choices` and return the option.
    [(-< <expr1>)
     (begin
       (set-choices! (void))
       <expr1>)]

    ; Multiple options: return the first and store the others in a thunk.
    [(-< <expr1> <expr2> ...)
     (begin
       ; 1. Store the remaining options in a thunk.
       (set-choices! (thunk (-< <expr2> ...)))

       ; 2. Return the first expression.
       <expr1>)]))

(define (next!)
  (if (void? choices)
      DONE
      (choices))); once you evaluate choices, it will become an expression, and will be evaluated automatically
    because you cannot return an expression without evaluating it in Racket.
```

In this implementation, we use mutation, marking a departure from previous functional code examples. The `set!` form allows us to reassign variables (encapsulated in the helper function `set-choices!`), while `begin` encloses a sequence of expressions, evaluating each in order and returning the value of the last expression. In pure functional programming, evaluating a sequence of expressions but returning only the last one does not make sense because previous expressions do nothing but return values. However, with `set!`, expressions can have side effects such as variable reassignment. This is also why the function name `next!` ends with an exclamation mark.

We can verify that our macro works as intended, as shown in the following example:

```
> (-< 1 2 (+ 3 4))
1
> (next!)
2
> (next!)
7
> (next!)
'done
```

Here, the initial call to `-<` returns the first argument (1) and stores the remaining expressions in `choices` as a stream. Calling `next!` is like calling `s-first` on the stream to retrieve the next value. The `-<` macro replaces `choices` with the rest of the stream, effectively consuming the first element and leaving the rest of the stream.

One particularly nice feature of this implementation is that, because the `set-choices!` expression is evaluated before `<expr1>`, `choices` is updated even if the next expression raises an error:

```
> (-< 1 2 (/ 1 0) 4)
1
> (next!)
2
> (next!)
ERROR: division by zero
> (next!)
4
```

While this implementation is capable of storing the other choices, it does not preserve the computational context. For example:

```
> (+ 3 (-< 1 2 (+ 3 4)))
4
```

```
> (next!)
2
> (next!)
7
```

6.4 Continuations

In our studies so far, we have mainly been passive participants in the operational semantics of programming languages, following the evaluation orders of function calls and special syntactic forms. Even though macros allow us to manipulate evaluation order, they do so by transforming expressions into predefined macros and function calls. However, Racket provides a data structure that directly represents and manipulates control flow within a program: the **continuation**. Consider the following expression:

```
(+ (× 3 4) (first (list 1 2 3)))
```

By now, you should have a clear understanding of how this expression is evaluated, particularly in which order the subexpressions are executed. **For each subexpression `s`, we define its continuation as a representation of what remains to be evaluated after `s` has been evaluated.** In other words, the continuation of `s` is the control flow from the point immediately after `s` is evaluated up to the final evaluation of the enclosing expression.

For example, the continuation of the subexpression `(* 3 4)` can be described as "evaluate `(first (list 1 2 3))` and then add the result to the current value." Symbolically, we represent its continuation as `(+ _ (first (list 1 2 3)))`.

It is important to note that continuations depend on the position of an expression in the larger program, not on the evaluation of the expression itself. For example, the continuation of `first` in the above expression is `(+ 12 _)` due to left-to-right evaluation.

6.5 Shift: Reifying Continuations

So far, continuations might seem like an abstract representation of control flow. However, in Racket, continuations are reified, meaning they are exposed as values that programs can access and manipulate. Racket allows us to capture and handle continuations using the syntactic form `shift`, which is imported from `racket/control`. Here is the relevant syntax:

```
(shift <id> <body> ...)
```

A `shift` expression works as follows:

1. The current continuation of the `shift` expression is bound to `id` (often denoted `k`).
2. The `body` is evaluated.
3. The current continuation is discarded, and the value of the last expression in `body` is returned.

The third point is particularly interesting, as it causes the value of the `shift` expression to "escape" any enclosing expressions, similarly to using `return` inside a function. Here is a simple example:

```
> (shift k (+ 4 5)) ; The shift's body is evaluated and returned.
9
> (× 100 (shift k (+ 4 5))) ; The continuation (× 100 _) is discarded!
9
```

In the second example, the continuation `(* 100 _)` is discarded, and the expression simply evaluates to `9`. We can bind the continuation to a variable and use it later. For example, we can store the continuation and reuse it after the `shift` expression has completed:

```
> (define saved-cont (void))
> (× 100
    (shift k
      (set! saved-cont k)
      (+ 4 5)))
9
> saved-cont
#<procedure>
> (saved-cont 9)
900
```

In this case, we stored the continuation `(* 100 _)` in a global variable `saved-cont` and then invoked it later by passing `9` to it, resulting in `900`.

To ensure that the continuation is both saved and executed in the regular control flow, we can invoke the continuation `k` within the body of the `shift` expression:

```
> (× 100
    (shift k
      (set! saved-cont k)
      (k (+ 4 5))))
900
```

Our motivation for learning about continuations was to improve our `-<` macro, which previously stored only the choices but not the computational context around each choice. With continuations, we can now capture that context. Here is an improved version of the `-<` macro using `shift`:

```
(require racket/control)
(define-syntax -<
  (syntax-rules ()
    [(-< <expr1>)
     (begin
       (set-choices! (void))
       <expr1>)]

    [(-< <expr1> <expr2> ...)
     (shift k
       (set-choices! (thunk (k (-< <expr2> ...))))
       (k <expr1>))]))
```

For instance, in the expression `(+ 1 (-< 10 20))`, the continuation `k` is bound to `(+ 1 _)`. The continuation is now saved in the choices, and when `next!` is called, it evaluates the next choice with the captured continuation. The power of continuations is demonstrated by how this small change greatly enhances the flexibility of our `-<` macro. And now we get our desired behaviour:

```
>(+ 1 (-< 10 20)) ; `k` is (+ 1 _)
11
>(next!) ; `choices` is (thunk ((+ 1 _) (-< 20))))
21
>(next!)
'done
```

6.6 Using choices as subexpressions

So far, our focus has been on the construction of the choices themselves, and we have verified our work by calling `(next!)` at the top-level. Since the continuations we store behave as unary functions, we can take the value returned and use it in a larger computation. For example:

```
> (× 100 (-< 1 2))
100
> (+ 3 (next!))
203
```

However, there are two subtle issues with our current implementation that arise when embedding `(next!)` inside larger expressions. By exploring these, we can refine our implementation and deepen our understanding of continuations. First, it turns out that `shift`'s ability to dynamically capture continuations is slightly too powerful. Suppose we extend our previous example with a third choice:

```
> (× 100 (-< 1 2 3))
100
> (+ 3 (next!)) ; Evaluates to (+ 3 (× 100 2))
203
```

If we call `next!` by itself one final time, something interesting happens:

```
> (next!)
303
```

This returns 303 instead of 300! It appears that the continuation `(+ 3 _)` was captured during the first call to `next!`—but why?

Recall that `(next!)` simply calls the thunk stored in `choices`, so let's take a closer look at the stored thunk. We start with `(* 100 (-< 1 2 3))` and perform one step of the macro expansion:

```
(× 100 (-< 1 2 3))
; ==>
(× 100
  (shift k
    (set-choices! (thunk (k (-< 2 3))))
    (k 1)))
```

This macro is recursive: the choices thunk contains another use of `-<`, which expands into another use of `shift`:

```
(× 100
  (shift k
    (set-choices!
      (thunk (k
        (shift k2 ; Renaming to k2 for clarity
          (set-choices! (thunk (k2 (-< 3))))
          (k2 2)))))
    (k 1)))
```

Thus, when we call `(next!)` inside `(+ 3 (next!))`, we evaluate the choices thunk, which triggers another `shift`. This new `shift` captures its current continuation, which includes not only `(* 100 _)`, but also the surrounding `(+ 3 _)`! The continuation bound to `k2` is not just `(* 100 _)`, but rather `(+ 3 (* 100 _))`, and when we apply this continuation to the last choice 3, we get 303.

This example is somewhat counterintuitive and highlights a potential pitfall of dynamic behavior: because our current design of `-<` involves delayed calls to `shift`, the choices produced by `-<` depend not only on the initial context where `-<` was used, but also on the contexts in which `(next!)` is invoked. **This is a pretty good example to demonstrate that laziness will make computation context pretty complex and difficult to analyze.** As with many dynamic language features, this is not inherently problematic, but it can make predicting the program's behavior more challenging. In this section, we will explore how to adjust our implementation so that calls to `next!` do not alter the continuation applied to subsequent choices.

Racket provides mechanisms to delimit continuations, allowing programmers to control which parts of a continuation are stored or discarded. The key mechanism we will use here is `reset`, which acts as a "barrier" limiting the continuation "scope" of a `shift`. When a `shift` occurs within a `reset`, the "current continuation" captured in the `shift` and discarded after the `shift` is evaluated only extends up to the `reset`, excluding any expressions outside the `reset`. Here are a few examples:

```
> (× 100 (reset (× 2 (shift k 9)))) ; The (× 2 \_) is discarded...
900                                 ; but the (× 100 \_) is not.

> (define saved-cont (void))
> (× 100
   (reset (× 2 (shift k
     (set! saved-cont k)
     9))))
900

> (saved-cont 9) ; saved—cont is (× 2 \_); doesn't include (× 100 \_)!
18
```

To prevent calls to `(next!)` from capturing new continuations in subsequent choices, we can wrap the thunk calls with `reset`:

```
(define (next!)
  (if (void? choices)
      DONE
      (reset (choices))))
```

Here's how it behaves:

```
;input the amb expression in the interactive window  to avoid repeated output
> (× 100 (-< 1 2 3))
100
> (+ 3 (next!))
203
> (next!)
300
```

Now let's do macro expansion:

```
;(× 100 (−< 1 2 3))
(× 100
   (shift k
          (set-choices! (thunk (k (-< 2 3))))
          (k 1)))

;(+ 3 (next!))
(+ 3 (reset(× 100 ;k (× 100 _)
               (shift k2
                      (set-choices! (thunk (k2 (-< 3))))
                      (k2 2)))))
;(next!)
(reset (× 100 3)); k2 (× 100 _)
```

Now, consider another problem:

```
> (× 100 (-< 1 2))
100
> (+ 3 (next!)) ; Evaluates to (+ 3 (× 100 2))
203
```

If we call `next!` again within a larger expression, we encounter an error:

```
> (+ 3 (next!))
+: contract violation
  expected: number?
  given: 'done
```

On the one hand, this behavior is expected: calling `(next!)` when no more choices are available returns our special constant `DONE`, which cannot be added to a number. On the other hand, this is quite inconvenient: under the current implementation, whenever we embed `(next!)` inside a larger expression, we either need to ensure that there is at least one remaining choice, or explicitly check whether the value returned by `(next!)` is `DONE` before continuing with the rest of the computation.

Given this trade-off, we would like a way for a call to `(next!)` to immediately return `DONE`, skipping any further computation surrounding the `(next!)`. As you may recall from earlier in this chapter, this exact behavior is provided by `shift`. So here is the modification to our `next!` implementation:

```
(define (next!)
  (if (void? choices)
      (shift k DONE)
      (reset (choices))))
```

If there are no more choices, instead of simply returning `DONE`, we evaluate `(shift k DONE)`, which causes any surrounding computation to be discarded. With this change in place, we can now freely call `(next!)` within larger expressions, allowing `'done` to "interrupt" the remaining computation:

```
> (× 100 (-< 1 2))
100
> (+ 3 (next!)) ; Evaluates to (+ 3 (× 100 2))
203
> (+ 3 (next!)) ; No more error!
'done
```

6.7 Branching Choices

Even with the improvements from the previous section, our work is not yet complete. The current implementation overwrites choices, meaning that at any given time, the program is only aware of a single choice point. For example, in the code below, evaluating the second choice overwrites the first:

```
> (+ (-< 10 20) (-< 2 3))
12
> (next!)
13
> (next!)
'done
```

The macro expansion is below:

```
;(+ (-< 10 20) (-< 2 3))
(+
 (shift k ;k, which represents the remaining computation after evaluating (-< 10 20).
   (set-choices! (thunk (k (-< 20))));
   (k 10))
 (shift k ;k, which represents the remaining computation after evaluating (-< 2 3).
   (set-choices! (thunk (k (-< 3))))
   (k 2)))

;simplify

(+ 10 (shift k
             (set-choices! (thunk (k (-< 3))))
             (k 2)))

;further evaluate the expression above, we get
(+ 10 2)

;(next!)
;what are stored in choices? (+ 10 _)
13
```

As language designers, we might stop here and declare, "Too bad! You can only use one `-<` expression per program." However, this would be unsatisfactory for users, so let's work harder to support multiple choice points. Clearly, adding more global variables (e.g., `choices1`, `choices2`, etc.) is not a feasible solution. Instead, we need a way to represent an arbitrary number of choice points, and a collection-type data structure is the most appropriate. For simplicity, we'll use a stack implemented as a list, where the front of the list represents the top of the stack:

```
; choices is now a list of thunks rather than a single thunk.
;null: This is a specific value representing an empty list or the end of a list in Racket.
;
(define choices null)
(define (add-choice! c) (push! choices c))
(define (get-choice!) (pop! choices))

; Push a value onto a stack (add to the front of the list).
(define-syntax push!
  (syntax-rules ()
    [(push! <id> obj)
     (set! <id> (cons obj <id>))]))

; Pop a value from a stack (remove from the front of the list).
(define-syntax pop!
  (syntax-rules ()
    [(pop! <id>)
     (let× ([obj (first <id>)])
       (set! <id> (rest <id>))
       obj)]))
```

The idea is straightforward: each time we encounter a choice expression, we push the corresponding thunk onto the `choices` stack. When `next!` is called, we pop a thunk off the stack and evaluate it (Depth-first Search).

```
(define-syntax -<
  (syntax-rules ()
    [(-< <expr1>) <expr1>]
    [(-< <expr1> <expr2> ...)
     (shift k
       (add-choice! (thunk (k (-< <expr2> ...))))
       (k <expr1>))]))
#|
(next!) → any/c

;Returns the next choice, or DONE if there are no more choices.
|#
(define (next!)
  (if (null? choices) ; null instead of void
      (shift k DONE)
      (reset ((get-choice!)))))


;Now, let's see what happens when we run the previous example:

> (+ (-< 1 2) (-< 3 10))
4
> choices ; choices stores two thunks, one for each -< expression
'(#<procedure> #<procedure>)
;What the two thunks actually looks like:
;(thunk (+ 1 (-< 10)))
;(thunk (+ (-< 2) (-< 3 10)))
```

After evaluating the expression containing two choice expressions, we see that `choices` now stores two thunks.

```
> (next!)
11
> (next!)
5
> (next!)
12
> (next!)
'done
```

6.8 Predicates and Backtracking

While we hope that you find our work on the choice macro intellectually stimulating, it is important to recognize that this is a powerful mechanism that can transform the way you write programs. Just as we saw in our discussion of streams, where thunks were used to decouple the production and consumption of linear data, our choice library allows for similar decoupling even when the production of data is non-linear.

Throughout this course, we have compared functional programming with imperative programming. One of the key differences is that functional programming encourages writing code that describes what computation you want the computer to perform, rather than detailing how to do it. In other words, functional programming tends to be more declarative.

For instance, here is a simple expression that generates all possible binary strings of length 5, following the English instruction "take five characters, each of which is either 0 or 1", Racket uses the # prefix to indicate a character literal:

```
; define five digits and give all the possible values to each digits

> (string (-< #\0 #\1)
          (-< #\0 #\1)
          (-< #\0 #\1)
          (-< #\0 #\1)
          (-< #\0 #\1))


"00000"
```

```
> (next!)
"00001"
> (next!)
"00010"
> (next!)
"00011"
> (next!)
"00100"
```

This easy composability is remarkable and is arguably as close as possible to expressing the task in natural language. Though we could achieve similar behavior with higher-order list functions or nested loops, the simplicity of this approach is quite elegant.

Generating all possible combinations of choices is impressive, but without a mechanism to filter unwanted combinations, it isn't very practical. While we could collect choices into a stream and apply a traditional filter, we will instead combine the automatic generation of choices with automatic backtracking—an approach central to logic programming languages like Prolog.

Logic programming is centered around the question: "Is this true?" Instead of writing programs as sequences of steps or combinations of functions, logic programs define a search space (the universe of values known to the program) and query the computer to answer questions based on these values (A more familiar example is SQL).

We already have a mechanism to define a search space using combinations of choices. What about queries? To start, we define a query operator `?-`, which takes a unary predicate and an expression. It returns the expression's value if it satisfies the predicate, or `DONE` if it doesn't.

```
(define (?- pred expr)
  (if (pred expr)
      expr
      DONE))

> (?- even? 10)
10
> (?- even? -3)
'done
```

Now, we can call this function with a choice expression!

```
> (?- even? (-< 1 2 3 4))
'done
> (next!)
2
> (next!)
'done
> (next!)
4
```

When `?-` is called, it receives the first value from the choice expression (e.g., 1). However, the continuation of the choice expression is `(?- even? _)`, so each time `next!` is called, the next choice is passed to the query. In AI terminology, calling `next!` causes backtracking, where the program goes back to a previous point in execution and makes a different choice.

When querying choices, we often only care about the successful ones and can ignore intermediate `'done` outputs. We can modify our predicate to automatically try the next choice if the predicate fails:

```
(define (?- pred expr)
  (if (pred expr)
      expr
      (next!)))

> (?- even? (-< 1 2 3 4))
2
> (next!)
4
> (next!)
```

```
'done
```

Now, every time the predicate test fails, `next!` is called, automatically backtracking on failures. Not so fast! There's a bug with this implementation of `?-`. Suppose we want to nest `?-` inside a larger expression. The following example represents the computation "multiply by 100 a choice of an even number between 1 and 5":

```
> (× 100 (?- even? (-< 1 2 3 4 5)))
20000
> (next!)
40000
> (next!)
'done
```

Something seems to work: `?-` correctly selects the even numbers, **but there is an extra multiplication by 100**. This suggests that `(* 100 _)` is being applied twice. Why?

```
> (× 100 (?- even? (-< 1 2 3 4 5)))
;Step 1: Expansion of (-< 1 2 3 4 5)
;The macro -< with multiple arguments is matched by the second clause in the syntax-rules. It expands as follows:

(shift k
  (add-choice! (thunk (k (-< 2 3 4 5))))
  (k 1))
;This captures the continuation k, which represents the remaining part of the expression: (× 100 (?- even? _)).

;The value 1 is evaluated.
;The remaining choices (-< 2 3 4 5) are stored in the choices stack via the add-choice! macro.
Thus, after this expansion, the full expression becomes:

(× 100 (?- even? (shift k
                        (add-choice! (thunk (k (-< 2 3 4 5))))
                        (k 1))))

;Step 2: ?- even? Check for 1
;Next, we expand the predicate check ?- even?. The first value passed to ?- even? is 1 (function call, eager evaluation).
    The expansion of ?- is:

(if (even? 1)
    1
    (next!))
;Since 1 is not even, this expands further into:

(× 100 (next!))
;At this point, the computation waits for the next call to next!, which will resume the stored continuation.

;Step 3: First Call to next!
;When next! is invoked, it retrieves the next stored choice from the choices stack. The stored choice corresponds to (-<
    2 3 4 5), which was saved earlier. The thunk is evaluated, expanding to:

(× 100(reset (× 100(?- even? (shift k
               (add-choice! (thunk (k (-< 3 4 5))))
               (k 2))))))

;Step 4: ?- even? Check for 2; k store continuation between shift and reset (× 100(?- even? _))
(× 100(reset (× 100(?- even? 2))))

(× 100 200)

20000
```

The subtle issue is that `next!` returns `((get-choice!))` without discarding the current continuation, allowing `next!` to be composed with larger expressions. But when backtracking, we do not want to preserve the current continuation. Instead, we want to restart at a previous point in the computation. To fix this, we wrap `next!` inside a `shift` to discard the current continuation.

```
(define (?- pred expr)
  (if (pred expr)
      expr
      (shift k (next!))))

> (× 100 (?- even? (-< 1 2 3 4 5)))
200
> (next!)
400
> (next!)
'done
```

We can do a similar analysis through macro expansion. The key change is that shift is now used in the ?- expression to discard the current continuation when next! is called. This means that when backtracking occurs, the multiplication by 100 will not be reapplied, as the previous continuation will be discarded.:

```

(× 100 (?- even? (-< 1 2 3 4 5)))
;Step 1: Macro Expansion of (-< 1 2 3 4 5)
;The macro -< with multiple arguments (1, 2, 3, 4, 5) expands as:

(shift k
  (add-choice! (thunk (k (-< 2 3 4 5))))
  (k 1))

;shift captures the continuation, which is everything after (-< 1 2 3 4 5) in this case, (× 100 (?- even? _)).
;The first choice, 1, is evaluated.
;The remaining choices (-< 2 3 4 5) are stored in the stack via add-choice!.
;Thus, after this step, the full expression becomes:

(× 100 (?- even? (shift k
                        (add-choice! (thunk (k (-< 2 3 4 5))))
                        (k 1))))
;Step 2: ?- even? Check for 1
The ?- predicate checks if the value is even. The first value passed to ?- even? is 1, so the
    expansion of ?- is:
(× 100 (?- even? 1))

; Let's further expand it into ?- function. Since 1 is not even, this expands further into:

(× 100 (shift k (next!)))

;At this point, we have used shift to discard the current continuation ((× 100 _)). The result of next! will not involve the
    current continuation anymore, because it has been captured and discarded by shift. The above expression is equal to:

(next!)

;Step 3: First Call to next!
;When next! is called, it retrieves the next stored choice from the stack. The stored choice corresponds to (-< 2 3 4 5),
    which was saved earlier. The thunk is evaluated, expanding to:

(reset (× 100 (?- even?(shift k
                (add-choice! (thunk (k (-< 3 4 5))))
                (k 2)))))


; k captures continuation between shift and reset, (× 100 (?- even? _)), and discard it. Thus, it is equal to:

(k 2)

;which is
(× 100 (?- even? 2))

```

```
;based on the definition of function ?-, further evaluation is

(× 100 2)

;This evaluates to 200.
```

Probably the easiest way to appreciate the power of combining choices with querying is to see examples in action. We have already seen how this allows lazy filtering, producing values one at a time rather than all at once. We can easily extend this to multiple choice points, though our `?-` function only accepts unary predicates, so we pass all choices as a list.

For example, given a number n, we can find all triples of numbers whose product is n. Here is the input data and predicate we use:

```
(define (num-between start end)
  (if (equal? start (- end 1))
      (-< start)
      (-< start (num-between (+ 1 start) end))))

(define (triples n)
  (list (num-between 1 (+ n 1))
        (num-between 1 (+ n 1))
        (num-between 1 (+ n 1))))

(define (product? n)
  (lambda (triple)
    (equal? n (apply × triple))))
```

Here's how we use the query for the range 1–12:

```
> (?- (product? 12) (triples 12))
'(1 1 12)
> (next!)
'(1 2 6)
> (next!)
'(12 1 1)
> (next!)
'done
```

In just a few lines of code, we expressed a computation for finding factorizations in a purely declarative way: specifying what we wanted to find rather than how to find it. This is the essence of logic programming: break a problem down into logical constraints, give the program these constraints, and let it find solutions.

Let's check some important details. We noticed that the function `num-between` is recursive function, and if we expand the function call (num-between 1 12), we will get:

```
(-< 1 (-< 2 (-< 3 ... (-< 12))))
;the thunk is (-< (-< 2 (-< 3 ... (-< 12)))), and (-< 2 (-< 3 ... (-< 12))) will be treated as a single element
;when we evaluate the thunk, it will treager the pattern [(-< <expr1>) <expr1>], which make it equal to (-< 2 (-< 3
    ... (-< 12)))
; thus, (-< 1 (-< 2 (-< 3 ... (-< 12)))) is equal to (-< 1 2 (-< 3 ... (-< 12)))
;and finally, you will get (-< 1 2 3 ... 12)
```

Now, the question is quite clear, we want a triple, each position can be a number from 1 to 12, and their multiplication should be equal to n. (Is backtracking brute-force?)

One famous problem in computer science is the satisfiability problem, which asks whether a propositional boolean formula can be made true. Here is an example written in Racket syntax:

```
(and (or x1 (not x2) x4)
     (or x2 x3 x4)
     (or (not x2) (not x3) (not x4))
     (or (not x1) x3 (not x4))
     (or x1 x2 x3)
     (or x1 (not x2) (not x4)))
```

With our current system, finding solutions is easy:

```
(define (sat lst)
  (let ([x1 (first lst)]
        [x2 (second lst)]
        [x3 (third lst)]
        [x4 (fourth lst)])
    (and (or x1 (not x2) x4)
         (or x2 x3 x4)
         (or (not x2) (not x3) (not x4))
         (or (not x1) x3 (not x4))
         (or x1 x2 x3)
         (or x1 (not x2) (not x4)))))

> (?- sat (list (-< #t #f)
                (-< #t #f)
                (-< #t #f)
                (-< #t #f)))
'(#t #t #t #f)
> (next!)
'(#t #t #f #f)
> (next!)
'(#t #f #t #t)
> (next!)
'(#t #f #t #f)
> (next!)
'(#f #f #t #t)
> (next!)
'done
```

7 Haskell Type System

A type is defined as a collection of values associated with specific behaviors on those values. This is a broad and abstract concept that varies between programming languages. For example, an integer in Racket may encompass a different range of values than an integer in C. In this chapter, we will discuss how types are represented and utilized within programming languages, emphasizing how differences in type design can significantly affect the structure and behavior of programs.

A type generally conveys essential information across languages: when we declare an expression E as having type "integer," it restricts both the values we expect E to hold (its denotational meaning) and how it can be used in a program (e.g., (+ 1 E) is valid, but E[10] is not). A type system is a set of rules within a programming language that governs the semantics of types. These rules specify how types are defined, the syntax for where and how types can be written, and how types influence both the operational and denotational semantics of the language.

7.1 Strong and Weak Typing

The distinction between *strong* and *weak typing* often gets mixed up with *static* and *dynamic typing*, but they address different aspects of a language's type system. To begin with, we'll clarify strong and weak typing, though it's worth noting that interpretations of these terms vary.

In a *strongly-typed* language, every value retains a consistent type throughout the program's execution. This does not mean that a variable's type cannot change; it simply means that each value itself is bound to a specific type. Despite being stored in binary form (0s and 1s), strongly-typed languages ensure that these values cannot be used in a way that disregards their types. Most modern languages are strongly-typed, which is why trying to use incompatible types in function calls results in type errors.

In contrast, a *weakly-typed* language lacks strict type enforcement, allowing values to be implicitly treated as a different type at runtime through a process known as *type coercion*. For instance, in some languages, the expression `"5" + 6` is valid and interprets `6` as a string, producing `"56"` instead of causing an error.

The extent of type coercion can differ widely. For instance, while many languages would interpret `3.5 + 4` without error, the results may vary depending on the language's approach to type flexibility. **Strong and weak typing thus reflect a range of nuanced rules governing type coercion and behavior, and understanding these nuances is essential for writing effective code.**

7.2 Static and Dynamic Typing

Another key property of a type system is the timing of type-checking—specifically, **whether types are checked before the code runs or during execution.** This distinction shapes a language's semantics and has major implications for interpreters and compilers.

In *statically-typed* languages, the type of each expression **is determined at compile time**, meaning all types are checked based on the source code before the program executes. Even in languages that allow variable reassignment, static typing enforces that reassigned values remain within the same type category. Typically, variables in statically-typed languages must declare their types explicitly, though there are exceptions.

On the other hand, *dynamically-typed* languages **defer type-checking until runtime**, making type errors akin to runtime exceptions like division by zero or accessing an array out-of-bounds. Type-checking at runtime adds flexibility but may result in errors that only appear during execution.

Static typing can be viewed as a form of program verification. By ensuring that variables maintain consistent types throughout the code, **a statically-typed language helps prevent runtime type errors. This checking also enables optimizations; if a compiler knows a variable's type won't change, it can omit type checks at runtime, enhancing performance.** Statically-typed languages conduct this verification by checking types during compilation, rejecting programs with type mismatches. While this often improves program reliability, it can also mean that some correct programs might be disallowed due to type constraints. Many statically-typed languages allow "escaping" strict typing with a universal type like `auto` or `Any`, enabling more flexibility without full type constraints.

7.3 The Basics of Haskell's Type System

In Haskell, types are denoted by capitalized names such as `Bool`, `Char`, and `Integer`. Within the Haskell compiler, you can check the type of any expression using `:type` (or the shorthand `:t`). Here are a few examples:

```
ghci> :t True
True :: Bool
ghci> :t 'a'
'a' :: Char
ghci> :t "Hello"
"Hello" :: String
```

Unlike Racket, lists in Haskell must contain values of the same type, so an expression like `[True, 'a']` is rejected. A similar restriction applies to `if` expressions: both the `then` and `else` branches must have the same type.

```
ghci> :t (if 3 > 1 then "hi" else "bye")
(if 3 > 1 then "hi" else "bye") :: String
```

This example demonstrates Haskell's static type-checking, where the type of the entire `if` expression is determined without evaluating it. Because both branches are strings, the compiler infers that the whole expression has the type `String`.

In Haskell, all functions have a type signature that can be inspected in the same way.

```
ghci> :t not
not :: Bool -> Bool
```

The type signature `Bool -> Bool` indicates that `not` is a function that accepts a `Bool` and returns a `Bool`. The presence of a type signature imposes constraints on function usage, requiring functions to have **a fixed number of arguments, with specific types for each, and a single return type.** For example, let's examine the `and` function (`&&`):

```
ghci> :t (&&)
(&&) :: Bool -> Bool -> Bool
```

While we might expect something like `(Bool, Bool) -> Bool` for a two-argument function, Haskell's type signature `Bool -> Bool -> Bool` reflects **right-associativity**, meaning the signature actually groups as `Bool -> (Bool -> Bool)`. This implies that `(&&)` is interpreted as a unary higher-order function, a concept known as *currying*: **any multi-parameter function can be decomposed into nested single-parameter functions.** For instance, consider the following partial application of `(&&)`:

```
ghci> newF y = (&&) True y
ghci> :t newF
newF :: Bool -> Bool
```

The type of `newF` is `Bool -> Bool`. Setting the first argument of `(&&)` to `True` effectively creates **a new function**. However, `y` is unnecessary, so we can simplify the definition to:

```
newF = (&&) True
```

The following definitions of `(&&)` are equivalent in Haskell:

```
-- Usual definition
(&&) x y = if x then y else False

-- As a higher-order function
(&&) x = \y -> if x then y else False
```

The second version defines (&&) using a higher-order function approach, taking advantage of **Haskell's currying and partial application**. Here, (&&) is defined as a function that takes a single argument x and returns another function \y -> if x then y else False. The returned function itself takes one argument, y, and performs the same conditional check.

For example, a ternary function can be written in four equivalent ways:

```haskell
f x y z = x + y * z
f x y = \z -> x + y * z
f x = \y -> (\z -> x + y * z) -- parentheses for clarity
f = \x -> (\y -> (\z -> x + y * z))
```

Haskell also allows for currying infix operators using *sectioning*(partially applying binary operators by putting them in parentheses), enabling partial application on either argument. For example:

```haskell
f = (True &&) -- equivalent to f x = True && x
g = (&& True) -- equivalent to g x = x && True
```

Sectioning offers flexibility in function composition. Consider the `addToAll` function, which adds a number to each element in a list:

```haskell
addToAll n lst = map (\x -> x + n) lst
```

Using sectioning, we can simplify this to:

```haskell
addToAll2 n lst = map (+ n) lst
```

Further, we can remove `lst` and write:

```haskell
addToAll3 n = map (+ n)
```

With familiarity in functional programming, this concise form reads as "`addToAll3` maps the 'add `n`' function."

7.4 Defining types in Haskell

Types created by combining constructors and unions are known as *algebraic data types*. The term "algebraic" refers to building types from a simple set of base types by applying operations on them. There are two main types of algebraic data types, `sum` and `product`.

Below we will first introduce product type (struct type): **These represent data structures where each element contains multiple values (like fields in a struct or record).** All fields are required to be present, which is like the "product" of the fields. A common example is a tuple or a struct. Throughout this section, we'll use geometric shapes as a running example. `Point` is a good example for product type of algebraic data type. To represent a point (x, y) on the Cartesian plane, we define a type `Point`:

```haskell
data Point = Point Float Float
```

This line creates a new type, `Point`, using the `data` keyword. The left side defines the type, while the right side shows how to construct instances of this type. `Point` is a value constructor that takes two `Float` arguments and produces a value of type `Point`. Overall, `Point` on the left side is the name of a type, while on the right, it represents a constructor function. Here's an example of using this type:

```haskell
> p = Point 3 4
> :t p
p :: Point
> :t Point
Point :: Float -> Float -> Point
```

Unlike in typical object-oriented languages, Haskell does not treat constructors as special methods (In C++, the constructor must have the same name as the class.). We could define `Point` with a different constructor name, as shown below:

```haskell
-- Here, Point is the type and MyPoint is the constructor.
data Point = MyPoint Float Float
```

This type definition is closer in spirit to `struct` types in C than to object-oriented classes. It allows us to define the structure of a `Point` (in this case, two floats) but doesn't bundle operations. To operate on points, we define top-level functions. For example, we can calculate the distance between two points:

```
distance :: Point -> Point -> Float
distance (Point x1 y1) (Point x2 y2) =
  let dx = abs (x1 - x2)
      dy = abs (y1 - y2)
  in sqrt (dx * dx + dy * dy)
```

Using pattern matching, we can access the `Float` components of a `Point` value. This feature allows us to define functions for custom types **by matching on constructors**, similar to matching on lists.

```
> distance (Point 3 4) (Point 1 2)
```

We can also create lists of `Point` values. Here's a function that takes lists of x and y coordinates and returns a list of `Point` values:

```
makePoints :: [Float] -> [Float] -> [Point]
makePoints xs ys = zipWith (\x y -> Point x y) xs ys
```

Partial application in Haskell allows you to provide fewer arguments than a function requires, producing a new function that takes the remaining arguments.

```
makePoints2 = zipWith (\x y -> Point x y)
```

This function could be simplified further, as the `Point` constructor itself is a function. Thus, we can write:

```
makePoints3 = zipWith Point
```

Concept	What It Is	Applies To	Purpose
Currying	How Haskell represents all functions internally	All functions	Turns multi-argument functions into chains of single-argument functions
Partial Application	Calling a function with fewer arguments than it expects	All functions	Produces a new function waiting for the remaining arguments
Sectioning	Syntactic sugar for partially applying binary (infix) operators	Infix operators only	Creates functions using `(op x)` or `(x op)` notation

Sum types (also called unions or variant types): **These represent values that could be one of several types, which is like the "sum" of possibilities.** Only one of the possible values is chosen at a time. Sum types are useful for representing data with different "cases" or "alternatives." For example, a `Day` type to represent days of the week:

```
data Day = Monday | Tuesday | Wednesday | Thursday | Friday | Saturday | Sunday
```

The vertical bar `|` separates the different **constructors**. Each day is a value constructor, just like `Point`, but these constructors don't take any arguments.

```
> :t Monday
Monday :: Day
```

Haskell allows us to define types with constructors that take different types of arguments. For example, we can define a `Shape` type that includes both circles and rectangles:

```
data Shape = Circle Point Float      -- ^ center and radius
           | Rectangle Point Point   -- ^ top-left and bottom-right corners
```

This code defines `Shape` with two constructors, `Circle` and `Rectangle`. Here's how we can use `Shape`:

```
> shape = Circle (Point 3 4) 1
> :t shape
shape :: Shape
```

We can define functions for `Shape` by using pattern matching for each constructor:

```
area :: Shape -> Float
area (Circle _ r) = pi * r * r
area (Rectangle (Point x1 y1) (Point x2 y2)) = abs ((x2 - x1) * (y2 - y1))
```

Using pattern matching, each constructor gets its own rule, and we can even nest patterns.

7.5 Comparison with Union in C

In C, you may be familiar with `struct` and `union` types, which together provide functionality somewhat similar to algebraic data types in Haskell. However, C syntax can be more verbose, and unions in C require additional management to keep track of which part of the union is currently in use. Below, we illustrate how to represent a `Shape` in C, which includes both circles and rectangles. Note that the `circle` and `rectangle` fields overlap in memory, meaning only one can be active at any time.

```
#include <stdio.h>

struct Point {
    float x, y;
};

union Shape {
    struct Circle {
        struct Point centre;
        float radius;
    } circle;

    struct Rectangle {
        struct Point corner1, corner2;
    } rectangle;
};

int main(void) {
    union Shape s;
    s.circle = (struct Circle){{1,2},1};
    printf("Circle has radius %f\n", s.circle.radius);

    s.rectangle = (struct Rectangle) {{4, 5}, {6, 2}};
    printf("Rectangle with corner (%f, %f)\n", s.rectangle.corner1.x, s.rectangle.corner1.y);

    return 0;
}
```

If we want to pass a `union Shape` to a function, we face a challenge: How do we determine if the shape is a circle or a rectangle? Unlike Haskell, which allows pattern matching, C requires a different approach. In C, the programmer must explicitly maintain a *tag* field to identify the active type within the union. This often means embedding the union within a `struct` that includes the tag field.

```
#include <stdio.h>
#include <math.h>
#define pi 3.1415926535897793

struct Point {
    float x, y;
};

```

```c
struct Shape {
    enum { CIRCLE, RECTANGLE } shape_tag;
    union {
        struct Circle {
            struct Point centre;
            float radius;
        } circle;

        struct Rectangle {
            struct Point corner1, corner2;
        } rectangle;
    } shape;
};

float area(struct Shape s) {
    if (s.shape_tag == CIRCLE) {
        float r = s.shape.circle.radius;
        return pi * r * r;
    } else { // rectangle
        float x1 = s.shape.rectangle.corner1.x;
        float y1 = s.shape.rectangle.corner1.y;
        float x2 = s.shape.rectangle.corner2.x;
        float y2 = s.shape.rectangle.corner2.y;
        return abs((x2 - x1) * (y2 - y1));
    }
}

int main(void) {
    struct Shape s;
    s.shape_tag = CIRCLE;
    s.shape.circle = (struct Circle){{1,2},1};
    printf("%f\n", area(s));

    s.shape_tag = RECTANGLE;
    s.shape.rectangle = (struct Rectangle) {{4, 5}, {6, 2}};
    printf("%f\n", area(s));

    return 0;
}
```

7.6 Algebraic data types

We now turn our attention to an advanced feature of Haskell's type system: polymorphism. The term "polymorphism" is derived from the Greek words *poly* (meaning "many") and *morphe* (meaning "form" or "shape"). In programming language theory, **polymorphism refers to an entity (e.g., function or class) that can operate correctly in different type contexts.** Let's explore this concept in Haskell, beginning with a familiar structure: the list.

7.6.1 The List Type in Detail

As previously discussed, Haskell's type system requires that all elements of a list share the same type:

```
ghci> :t [True, False, True]
[True, False, True] :: [Bool]
ghci> :t [(&&) True, not]
[(&&) True, not] :: [Bool -> Bool]
```

Lists in Haskell are quite flexible: as long as this type restriction is respected, lists can hold values of any type. But this raises a question: what is the type of functions like `head`, which operate on lists? The function `head` takes a list and returns its first element, but the type of that element depends on the list's type, which could be any type!

```
ghci> :t (head [True, False, True])
(head [True, False, True]) :: Bool
ghci> :t (head "abc")
```

```
(head "abc") :: Char
ghci> :t head
head :: [a] -> a
```

The type of `head` is `[a] -> a`, which differs from most type signatures we've encountered so far because it includes a *type variable* `a`. In Haskell, type variables must start with a lowercase letter, distinguishing them from concrete types. **A type variable is a placeholder in a type signature that can be instantiated with any type**, such as `Bool` or `Char`. This signature means that `head` works for any type of the form `[a]` and returns a value of type `a`. When `head` is called, Haskell matches the parameter type `[a]` to the argument's type and instantiates `a` as needed. For example, for the expression `head [True, False, True]`:

1. Haskell determines the type of the argument list as `[Bool]`.
2. Haskell matches `[Bool]` with the parameter type `[a]`, instantiating `a = Bool`.
3. Haskell then uses `a = Bool` to derive that the return type is `Bool`, making the overall type of the function call `Bool`.

Consider a more complex example, the `filter` function:

```
ghci> filter (>= 1) [10, 0, -5, 3] -- \x -> x >= 1.
[10,3]
```

What is the type of `filter`? Conceptually, `filter` takes two arguments: a list of any type, `[a]`, and a function from `a` to `Bool`, `a -> Bool`. It returns a list containing elements of the original type `[a]` that satisfy the predicate. Therefore, the type of `filter` is:

```
filter :: (a -> Bool) -> [a] -> [a]
```

7.6.2 The List Type Definition

How does Haskell know to allow the programmer to create lists of different types? The answer lies in the list type definition. To illustrate, we'll start by defining an integer list type. Such a list is either empty or an integer paired with another integer list:

```
data IntList = Empty | Cons Int IntList
```

What if we wanted a list of strings instead?

```
data StrList = Empty | Cons String StrList
```

The only difference is the type of the first argument to `Cons`, which determines the type of items stored in the list. To generalize, we can introduce a type variable, **all type variables in constructor fields must appear after the type name on the left-hand side**+. :

```
data List a = Empty | Cons a (List a)
```

Here, `a` is a type variable, allowing us to instantiate `List a` with different types (like `Int` or `String`). This construction makes `List` a *type constructor*, not a concrete type, since it requires a type argument to form a specific type like `List Int`. This general definition is how Haskell defines its built-in list type, though in a more concise format (Haskell use different symbols to represent Empty and Cons we used above):

```
data [] a = [] | (:) a ([] a)
```

These constructors are simply functions! The type of the list constructor `(:)` is no surprise, though the empty list constructor `[]` might be unexpected:

```
ghci> :t (:)
(:) :: a -> [a] -> [a]
ghci> :t []
[] :: [a]
```

The type of `[]` is `[a]`, which allows the empty list `[]` to be compatible with lists of any type, eliminating extra work for the programmer.

7.6.3 Generic Polymorphism

In Haskell, the list is a good example for *generic polymorphism*, a form of polymorphism where an entity (e.g., function or data type) functions consistently regardless of the specific type it is working with. Haskell lists are generically polymorphic, meaning they can contain elements of any type, while their structure (defined by `[]` and `(:)`) remains the same regardless of the element type. Similarly, nearly all built-in list functions are generic, allowing them to operate on lists without needing to know the type of elements contained within.

In terms of type signatures, a quick rule of thumb for identifying generic functions is to look for type variables in the signature: if a function's type signature contains a type variable, it is likely generic. Conversely, if no type variables appear in the signature, the function is specific to a particular type, known as a *concrete* type.

Generic polymorphism significantly reduces code duplication by enabling the reuse of a single function or data structure for multiple types. In C++, *templates* serve a similar role, allowing for type variables in both functions and classes. The following example demonstrates a function template, which also illustrates a more limited form of type inference available in C++.

```cpp
#include <iostream>

template<typename Type>
void f(Type s) {
    std::cout << s << '\n';
}

int main() {
    f<double>(1);           // instantiates and calls f<double>(double)
    f<>('a');                // instantiates and calls f<char>(char)
    f(7);                    // instantiates and calls f<int>(int)
    void (*ptr)(std::string) = f; // instantiates f<string>(string)
}
```

7.7 Type Classes

Recall our earlier definitions of the `Point` and `Shape` types:

```haskell
data Point = Point Float Float

data Shape
    = Circle Point Float      -- ^ center and radius
    | Rectangle Point Point   -- ^ top-left and bottom-right corners
```

Currently, we cannot directly view (print) instances of these custom types Haskell compiler does not know how to display them as strings. This is where *type classes* become useful.

A *type class* in Haskell is a collection of types accompanied by a set of functions that must be implemented for these types, resembling the concept of an abstract interface in object-oriented programming. **A type becomes a member of a type class when the class's functions are implemented for it.** Haskell includes a built-in type class called `Show`, which contains the function `show` and is defined (in simplified form) as follows:

```haskell
class Show a where
    show :: a -> String
```

Here, the type variable `a` means that any type `a` can belong to the `Show` type class if it has an implementation of the `show` function that takes a value of type `a` and returns a `String`.

Most built-in Haskell types, such as `Int` and `Bool`, are members of the `Show` type class. When evaluating an expression, it uses `show` to convert the resulting value to a string, enabling it to display the result. This behavior only works if

the type of the expression is a member of `Show`.

To make our `Point` type a member of the `Show` type class, we need to implement the `show` function specifically for `Point`. We do this using the `instance` keyword, as shown below:

```haskell
data Point = Point Float Float

instance Show Point where
    show (Point x y) = "(" ++ show x ++ ", " ++ show y ++ ")"

main :: IO ()
main = do
    let p = Point 3 4
    print p
```

So, what exactly is `show`? While we often refer to it as a function, it has unique characteristics. The function `show` has an initial declaration and type signature that are independent of its implementations, and it has multiple implementations spread across the Haskell standard library as well as our custom `Point` code. Unlike typical functions, `show` has a specific implementation for each type that is a member of the `Show` type class. Its type signature is slightly different:

```haskell
> :t show
show :: Show a => a -> String
```

This type signature contains a type variable modified by a new piece of syntax, `Show a =>`. This part before the `=>` is known as a *type class constraint*, indicating that the type variable `a` can only be instantiated to types that are members of the `Show` type class. We call `a` a *constrained type variable* because of this restriction.

- If a function's type signature contains no type variables, it is not polymorphic.
- If a function's type signature contains an unconstrained type variable, it is *generically polymorphic* in that variable, meaning the function must behave consistently for any instantiation of that type variable.
- If a function's type signature contains a constrained type variable, it is still polymorphic but limited to types that satisfy the specified type class constraints (Ad Hoc Polymorphism).

More examples:

```haskell
data Point = Point Float Float

instance Show Point where
    show (Point x y) = "(" ++ show x ++ ", " ++ show y ++ ")"

data Day = Monday
         | Tuesday
         | Wednesday
         | Thursday
         | Friday
         | Saturday
         | Sunday

data Shape = Circle Point Float
           | Rectangle Point Point

instance Show Shape where
  show (Circle x y) = "( Circle " ++ (show x) ++ ", " ++ (show y) ++ ")"

instance Show Day where
  show Monday = "Monday"
  show Tuesday = "Tuesday"
  show Wednesday = "Wednesday"
  show Thursday = "Thursday"
  show Friday = "Friday"
  show Saturday = "Saturday"
```

```
  show Sunday = "Sunday"

main :: IO()
main = do
  let p1 = Point 2 4
  let cir = Circle p1 3
  let weekday = [Monday, Tuesday, Wednesday, Thursday, Friday]
  putStrLn $ "Point Example: " ++ show p1
  putStrLn $ "Day Example: " ++ show weekday
  putStrLn $ "Shape Example: " ++ show cir
```

7.7.1 Some built-in type classes

In this section, we briefly describe some type classes you'll commonly encounter in Haskell. We'll start with two type classes used for comparing values:

- **Eq**: Members of this type class support the `(==)` and `(/=)` functions for testing equality. Only `(==)` needs to be explicitly implemented, as `(/=)` is given a default implementation in terms of `(==)`.
- **Ord**: Members of `Ord` can be compared using `(<)`, `(<=)`, `(>)`, and `(>=)`. For a type to be a member of `Ord`, it must also be a member of `Eq`; we say that `Ord` is a subclass of `Eq`. Only `(<)` and `(==)` (from `Eq`) need to be explicitly implemented.

```
data Point = Point Float Float

instance Show Point where
    show (Point x y) = "(" ++ show x ++ ", " ++ show y ++ ")"

instance Eq Point where
  (Point x1 y1) == (Point x2 y2) = x1 == x2 && y1 == y2

instance Ord Point where
  (Point x1 y1) <= (Point x2 y2) = (x1 == x2 && y1 == y2) || (x1 < x2) || (x1 == x2 && y1 < y2)

main :: IO()
main = do
  let p1 = Point 2 4
  let p2 = Point 1 2
  putStrLn $ "p1 " ++ show p1 ++ " is equal to " ++ "p2" ++ show p2 ++" which is " ++ show (p1 == p2)
  putStrLn $ "p1 " ++ show p1 ++ " > p3 " ++ show p2 ++ " which is "++ show (p1 > p2)
```

Now that we have a foundation in type classes, we can discuss Haskell's numeric types, starting with the addition operator.

```
> :t (+)
(+) :: Num a => a -> a -> a
```

The `Num a` constraint means that `a` must be a numeric type, and this is nearly correct. More specifically, `Num` is a type class that supports basic numeric operations, and which types like `Integer` and `Float` belong to. Haskell has three primary numeric type classes:

- **Num**: Provides basic arithmetic operations such as `(+)`, `(-)`, and `(*)`. Notably, `Num` does not include a division function, as division is managed by its subclasses. It also provides the function `fromInteger`, which we'll discuss shortly.
- **Integral**: Represents integer types and includes functions for integer division, such as `div` and `mod`.

```
    ghci> div 15 2
    7
    ghci> :t div
    div :: Integral a => a -> a -> a
```

- **Fractional**: Represents non-integer types, providing the standard division operator `(/)`, among other functions.

In Haskell, **numeric literals** are influenced by type classes, which is a unique feature of the language. Consider the type of the integer literal `1`:

```
ghci> :t 1
1 :: Num a => a
```

Now we can explain why the expression 1 + 2.3 correctly returns 3.3 in Haskell. The expression `1 + 2.3` works due to Haskell's use of *type classes* to handle numeric literals flexibly, ensuring they can interact in ways that satisfy both types involved. Here's a breakdown of how Haskell evaluates `1 + 2.3`.

- `1`: By default, `1` is an integer literal with the type `Num a => a`. This means `1` can represent any type that belongs to the `Num` type class, such as `Int`, `Integer`, `Float`, or `Double`.
- `2.3`: The literal `2.3` has the type `Fractional a => a` because it contains a decimal point, indicating it represents a non-integer (fractional) number. In Haskell, only types in the `Fractional` type class (such as `Float` and `Double`) can represent such values.

The expression `1 + 2.3` uses the `+` operator, which has the type:

```
(+) :: Num a => a -> a -> a
```

This type signature means that both arguments to `+` must have the same type, and this type must be a member of the `Num` type class.

1. **Unifying Types**: For `1 + 2.3` to work, both `1` and `2.3` must be of the same type. Since `2.3` is constrained to `Fractional`, Haskell must resolve `1` to a type compatible with both `Num` (required by `+`) and `Fractional`.

   Note: `::`(Type Annotation) is used to specify what type an expression should be interpreted as or constrained to, based on the context. It doesn't change the value's underlying type.

2. **Choosing `Fractional`**: Because `Fractional` is a subclass of `Num`, Haskell selects `Fractional` as the common type class for both literals. This allows Haskell to interpret `1` as a `Fractional` type.

   ```
   ghci>:t (1 + 2.3)
   (1 + 2.3) :: Fractional a => a
   ```

3. You can further specify its data type:

   ```
   ghci> a = (1 + 2.3)::Float
   ghci> :t a
   a :: Float
   ```

Right now, all cases are concrete data types, what if it is a generic polymorphic data type that contains a type variable?

```haskell
data Vector2 a = Vector2 a a deriving Show

-- Define the Eq instance for Vector2
instance Eq a => Eq (Vector2 a) where
    (Vector2 x1 y1) == (Vector2 x2 y2) = (x1 == x2) && (y1 == y2)

main :: IO ()
main = do
    let v1 = Vector2 3 4
    let v2 = Vector2 3 4
    let v3 = Vector2 5 6
    print (v1 == v2) -- Output: True, because v1 and v2 are the same
    print (v1 == v3) -- Output: False, because v1 and v3 are different
```

- Instance Declaration: instance Eq a => Eq (Vector2 a) means that Vector2 a will be an instance of Eq as long as a itself is an instance of Eq.
- Equality Function: (Vector2 x1 y1) == (Vector2 x2 y2) checks if the x and y components of both Vector2 values are equal.

7.7.2 Functors

In addition to lists, Haskell has various other "container" types, such as sets and vectors. One of the most useful higher-order transformations we've seen is `map`, which applies a function to each element in a list, producing a new list:

```
map :: (a -> b) -> [a] -> [b]
```

While `map` works for lists, we would like to apply a similar transformation to other container types like sets and vectors:

```
setMap :: (a -> b) -> Set a -> Set b
vectorMap :: (a -> b) -> Vector a -> Vector b
```

In Racket, this problem is addressed by defining separate "mapping" functions for each data type, using prefixes to avoid name collisions, as shown above. However, this approach limits our ability to write elegant, polymorphic code that works across container types. For instance, consider a function that maps an "add 1" operation across a collection of elements, regardless of the specific container type. In Racket, this might look like:

```
(define (add1 x) (+ x 1))

(define (add-1-all items)
  (cond
    [(list? items) (map add1 items)]
    [(set? items) (set-map add1 items)]
    [(vector? items) (vector-map add1 items)]))
```

In Haskell, the solution seems straightforward: we can make `map` part of a type class, allowing it to be implemented separately for each container type. This is indeed the approach Haskell takes, with one subtlety: since this type class represents "container" types, **its members are type constructors, not regular types**. Here's the definition of the built-in "mappable" type class in Haskell, called `Functor`:

```
class Functor f where
    -- the "f" in fmap stands for "Functor"
    fmap :: (a -> b) -> f a -> f b
```

In this definition, the type variable `f` represents a member of `Functor`. From the type signature of `fmap`, we can see that `f` is not a primitive type like `Int` or `Bool`, but rather a type constructor like the list type `[]` (empty list), `Set`, or `Vector`.

In Haskell, **a Functor is a type class that represents types that can be "mapped over."** Functors allow you to apply a function to each element inside a container (or structure) without changing the container's shape.
Below is an example:

We'll create a simple 2D vector type, `Vector2`, from scratch, make it an instance of the `Functor` type class, and use `fmap` to apply functions to its elements. We start by defining our own `Vector2` type that holds two values of the same type:

```
-- Define the Vector2 type with two components
data Vector2 a = Vector2 a a deriving Show
```

To make `Vector2` an instance of the `Functor` type class, we need to define how `fmap` applies a function to both elements inside `Vector2`:

```
-- Make Vector2 an instance of Functor
instance Functor Vector2 where
    fmap f (Vector2 x y) = Vector2 (f x) (f y)
```

Now that `Vector2` is an instance of `Functor`, we can use `fmap` to apply functions to both elements within a `Vector2`.

```haskell
data Vector2 a = Vector2 a a deriving Show

instance Functor Vector2 where
    fmap f (Vector2 x y) = Vector2 (f x) (f y)

main = do
         let myVector = Vector2 3 4
         let newVector = fmap (+1) myVector
         print newVector
```

Running this code should yield:

```
Vector2 4 5
```

We can use `fmap` with any function matching the type `(a -> b)`, allowing transformations on `Vector2` of any type:

```haskell
data Vector2 a = Vector2 a a deriving Show

instance Functor Vector2 where
    fmap f (Vector2 x y) = Vector2 (f x) (f y)

main = do
         let stringVector = Vector2 "hello" "world"
         let exclaimedVector = fmap (++ "!") stringVector

         print exclaimedVector
```

Expected output:

```
Vector2 "hello!" "world!"
```

7.8 Failing Representation

In Haskell, some functions are considered unsafe, such as `head`, `tail`, and any function with incomplete pattern matches. In this section, we'll explore how Haskell's type system can incorporate the possibility of errors in a way that is **statically checkable**. Specifically, we can indicate in our programs where computations might fail and make this known to the compiler by defining a data type that represents the outcome of a computation that may or may not be successful. `Maybe` forces you to handle the absence of a value, it prevents errors caused by assuming that a value will always be present.

The key type constructor for handling potentially failing computations in Haskell is `Maybe`, which is used to represent such values.

```haskell
data Maybe a = Nothing | Just a
```

Intuitively, the type `Maybe a` represents values with two possible states: they could be `Nothing`, indicating that a computation failed, or they could be `Just a`, indicating that the computation succeeded and produced a result of type `a`. Many programming languages encode this concept implicitly with special values like `None` or `null`; Haskell goes a step further by not only introducing a corresponding value, `Nothing`, but also **encoding this explicitly in the type system**. For example, `Maybe Int` is distinct from `Int`, allowing the compiler to distinguish between these types.

Below are some simple functions demonstrating the use of this data type. Note that the `a` type within `Maybe` can be any type, as `Maybe` is a generic polymorphic type.

```haskell
safeDiv :: Int -> Int -> Maybe Int
safeDiv _ 0 = Nothing -- divisor is 0
safeDiv x y = Just (div x y)

```

```haskell
safeHead :: [a] -> Maybe a
safeHead [] = Nothing
safeHead (x:_) = Just x

safeTail :: [a] -> Maybe [a]
safeTail [] = Nothing
safeTail (_:xs) = Just xs

assertNonNegative :: Int -> Maybe Int
assertNonNegative n =
    if n >= 0
    then Just n
    else Nothing
```

The final example, `assertNonNegative`, is a bit unconventional since it doesn't modify the input but instead checks a condition on it. A more generalized version might make this use clearer:

```haskell
assert :: (a -> Bool) -> a -> Maybe a
assert pred x =
    if pred x
    then Just x
    else Nothing
```

To fully appreciate the utility of the `assert` function, it helps to think beyond individual functions returning `Maybe` values, considering instead how to integrate such functions into larger expressions.

Run the code below to see the difference between `safeHead` and `head`:

```haskell
safeHead :: [a] -> Maybe a
safeHead [] = Nothing
safeHead (x:_) = Just x

main = do
    print (safeHead ([] :: [Int]))       -- switch the function to head, what will happen?
    print (safeHead ([1,2,3] :: [Int]))
```

7.9 Lift

Recall that Haskell already provides the operator `(.)` for composing regular functions (currying):

```haskell
(.) :: (b -> c) -> (a -> b) -> a -> c
f . g = \x -> f (g x)
```

Let's start with a simple example: defining a function that takes the first element of a list of numbers and adds 1 to it. If we are not concerned about runtime errors, we could define this function as follows:

```haskell
add1ToHead :: [Int] -> Int
add1ToHead = (+1) . head

main = do
    print (add1ToHead [1,2,3])
```

It might be tempting to make this function "safe" by replacing `head` with our earlier `safeHead` function, returning a `Maybe` result:

```haskell
safeHead :: [a] -> Maybe a
safeHead [] = Nothing
safeHead (x:_) = Just x

add1ToHead :: [Int] -> Int
```

```haskell
add1ToHead = (+1) . safeHead

main = do
    print (add1ToHead [1,2,3])
```

Unfortunately, this code will not be successfully compiled. The problem is that `(+1)` expects a numeric type (like `Int`), but `safeHead` produces a `Maybe Int` instead.

```
[1 of 1] Compiling Main             ( main.hs, main.o )
main.hs:6:14: error:
    * Couldn't match type `Maybe Int' with `Int'
      Expected type: [Int] -> Int
Actual type: [Int] -> Maybe Int
    * In the expression: (+ 1) . safeHead
      In an equation for `add1ToHead': add1ToHead =(+ 1) . safeHead
  |
6 | add1ToHead = (+1) . safeHead
  |              ^^^^^^^^^^^^^^
```

One solution is to define a new function that explicitly handles `Maybe` values using pattern matching:

```haskell
add1Maybe :: Maybe Int -> Maybe Int
add1Maybe Nothing = Nothing
add1Maybe (Just n) = Just (n + 1)
```

However, this approach does not scale well; we do not want to define a "Maybe" version of every function we intend to use. Instead, **we can define a higher-order function that takes any pure `a -> b` function and transforms it into one that operates on `Maybe a -> Maybe b` values.** We call this operation `lift`, borrowing a term from mathematics, where lifting often refers to the transformation of an object to place it in a more abstract or general context.

```haskell
lift :: (a -> b) -> (Maybe a -> Maybe b)
lift _ Nothing = Nothing
lift f (Just x) = Just (f x)
```

Using this function, we can now create a correct implementation of `safeAdd1ToHead`:

```haskell
safeHead :: [a] -> Maybe a
safeHead [] = Nothing
safeHead (x:_) = Just x

lift :: (a -> b) -> (Maybe a -> Maybe b)
lift _ Nothing = Nothing
lift f (Just x) = Just (f x)

safeAdd1ToHead :: [Int] -> Maybe Int
safeAdd1ToHead = (lift (+1)) . safeHead

main = do
    print (safeAdd1ToHead [1,2,3])
```

Doesn't the type signature for `lift` look familiar? Remember that the `->` operator in type expressions is right-associative, so parentheses in the second argument are not necessary:

```haskell
lift :: (a -> b) -> Maybe a -> Maybe b
```

Indeed, this type signature exactly matches that of `fmap`. This is no coincidence—`Maybe` is a member of the `Functor` type class! Thus, we can simplify our `safeAdd1ToHead` implementation by using `fmap`:

```haskell
safeAdd1ToHead :: [Int] -> Maybe Int
safeAdd1ToHead = (fmap (+1)) . safeHead
```

Let's consider defining a function that returns the second element of a list:

```
second :: [a] -> a
second = head . tail
```

Our first attempt to make this function safe is to replace the list functions with the safe variants we defined earlier:

```
safeHead :: [a] -> Maybe a
safeHead [] = Nothing
safeHead (x:_) = Just x

safeTail :: [a] -> Maybe [a]
safeTail [] = Nothing
safeTail (_:xs) = Just xs

safeSecond :: [a] -> Maybe a
safeSecond = safeHead . safeTail

main = print (safeSecond [1,2,3])
```

However, we encounter a familiar issue: `safeTail` returns a `Maybe [a]`, while `safeHead` expects a `[a]`. What if we try using `fmap` (or `lift`)?

```
safeSecond :: [a] -> Maybe a
safeSecond = (fmap safeHead) . safeTail
```

Unfortunately, this implementation returns a type of `Maybe (Maybe a)` instead of the desired `Maybe a`.

```
[1 of 1] Compiling Main             ( main.hs, main.o )
main.hs:1:1: error:
    The IO action 'main' is not defined in module 'Main'
  |
1 | safeHead :: [a] -> Maybe a
  | ^
main.hs:12:14: error:
    • Occurs check: cannot construct the infinite type: a ~ Maybe a
      Expected type: [a] -> Maybe a
        Actual type: [a] -> Maybe (Maybe a)
    • In the expression: (fmap safeHead) . safeTail
      In an equation for 'safeSecond':
          safeSecond = (fmap safeHead) . safeTail
    • Relevant bindings include
        safeSecond :: [a] -> Maybe a (bound at main.hs:12:1)
   |
12 | safeSecond = (fmap safeHead) . safeTail
```

To solve this, we'll take a different approach, starting with a correct (if verbose) implementation of `safeSecond`.

```
safeSecond :: [a] -> Maybe a
safeSecond xs =
    let xs' = safeTail xs
    in case xs' of -- similar to the switch case in C
        Nothing -> Nothing
        Just xs'' -> safeHead xs''
```

To understand this, let's examine the types involved:

- `xs` has type `[a]`, which is the original input to the function.
- `xs'` has type `Maybe [a]`, as returned by `safeTail`.
- `xs''` has type `[a]`—the "unwrapped" version of `xs'`, which is safe to access within the `case` expression by pattern matching on both possible outcomes of `xs'`.

Using `case` expressions, however, does not scale elegantly for composing multiple computations. Suppose we want to retrieve the fourth element of a list:

```haskell
safeFourth :: [a] -> Maybe a
safeFourth xs =
    let xs' = safeTail xs
    in case xs' of
        Nothing -> Nothing
        Just xs1 -> let xs1' = safeTail xs1
                    in case xs1' of
                        Nothing -> Nothing
                        Just xs2 -> let xs2' = safeTail xs2
                                    in case xs2' of
                                        Nothing -> Nothing
                                        Just xs3 -> safeHead xs3
```

To reduce this repetitive pattern matching, we can use a helper function:

```haskell
andThen :: Maybe a -> (a -> Maybe b) -> Maybe b
andThen Nothing _ = Nothing
andThen (Just x) f = f x -- Since f :: a -> Maybe b, applying f x will directly produce a Maybe b.
```

Like `lift`, `andThen` has a straightforward implementation. However, unlike `lift`, we don't wrap the function body in a `Just`—we can't assume that applying `f` will be successful! Using `andThen`, we can rewrite `safeSecond` in a much cleaner way:

```haskell
safeHead :: [a] -> Maybe a
safeHead [] = Nothing
safeHead (x:_) = Just x

safeTail :: [a] -> Maybe [a]
safeTail [] = Nothing
safeTail (_:xs) = Just xs

andThen :: Maybe a -> (a -> Maybe b) -> Maybe b
andThen Nothing _ = Nothing
andThen (Just x) f = f x

safeSecond :: [a] -> Maybe a
safeSecond xs =
    let xs' = safeTail xs
    in andThen xs' safeHead

main = print (safeSecond [1,2,3])
```

This approach becomes more readable when using `andThen` in infix notation:

```haskell
safeSecond :: [a] -> Maybe a
safeSecond xs = safeTail xs `andThen` safeHead

-- Or even more concisely:
safeSecond = safeTail `andThen` safeHead
```

With `andThen`, we can now remove all repetitive boilerplate from `safeFourth`, resulting in a clear and concise solution:

```haskell
safeFourth :: [a] -> Maybe a
safeFourth xs =
    safeTail xs `andThen`
    safeTail `andThen`
    safeTail `andThen`
    safeHead
```

One limitation of using `Maybe` for error handling is that chained computations do not retain information about where or why a computation failed. A `Nothing` value indicates failure, but it provides no further details. In practice, the `Either` type constructor is often preferred for error reporting:

```
data Either a b = Left a | Right b
```

`Either` is a versatile type constructor that represents a choice between two types (e.g., "this function returns either an `Int` or a `String`"). One of its common uses is to represent failure with the `Left` constructor, containing error information, and success with the `Right` constructor, analogous to `Just`. For example, a basic implementation might store an error message as a string:

```
safeDiv2 :: Int -> Int -> Either String Int
safeDiv2 _ 0 = Left "Division by 0 error"
safeDiv2 x y = Right (x `div` y)

main = print (safeDiv2 10 0)
```

More sophisticated systems may use a custom data type for the `Left` value to represent more detailed error information.

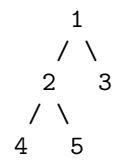
7.10 Modeling Mutation in Pure Functional Programming

Mutable state often provides a natural way to model certain problem domains. In this section, we address the question: "How can we simulate changing state in Haskell, a language that does not permit re-binding of identifiers?"

Consider the following problem: given a binary tree, label each node according to its position in a postorder traversal of that tree. A postorder traversal of a binary tree is a type of depth-first traversal where nodes are visited in the following order:

1. Visit the left subtree.
2. Visit the right subtree.
3. Visit the root node.

This traversal order is useful when you need to process child nodes before their parent, which is often needed in situations like deleting nodes or evaluating expressions in an expression tree. Consider the binary tree shown below:



The postorder traversal for this tree would visit the nodes in the order: $4, 5, 2, 3, 1$. To implement a postorder traversal in Python, we first define a `TreeNode` class that represents a node in a binary tree:

```
class TreeNode:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None
```

Now, given a binary tree, label each node with its position in a postorder traversal of that tree. We define a `postorder_traversal` function to perform the traversal. This function will label each node's position in postorder sequence.

```
i = 0

def postorder_traversal(root):
    global i
    if root is None:
        return

    # Traverse the left subtree
```

```
    postorder_traversal(root.left)

    # Traverse the right subtree
    postorder_traversal(root.right)

    # Visit the root node
    root.value = i
    i += 1
```

This code is elegant because it leverages the global variable `i`, which is mutated through recursive calls to `postorder_traversal`. Our goal here is to implement this algorithm in Haskell. For any mutable state, the two fundamental operations are `read` and `write`, here we use names `get` and `put`. We can define them as follows:

```
get :: s -> s
get state = state

put :: s -> s -> s
put item state = item
```

- `get` takes in the current state and simply returns it, simulating "reading" from the state.
- `put` takes an item and the current state, returning the item as the new state (the old state is discarded).

To better represent the difference between functions that return a value and those that modify the state, we unify their return types by using a tuple. The first element represents any return value, and the second element represents the new state.

```
get :: s -> (s, s)
get state = (state, state)

put :: s -> s -> ((), s)
put item state = ((), item)
```

- `get` takes the old state and returns it as both elements of the tuple. The first element represents the return value.
- `put` updates the state by setting the second element in the tuple to `item`, with the `()` in the first position indicating no return value.

To further streamline this approach, we use Haskell's built-in `State` data type, which encapsulates the "updating state" function type:

```
data State s a = State (s -> (a, s))

get :: State s s
get = State (\state -> (state, state))

put :: s -> State s ()
put item = State (\state -> ((), item))
```

Note: the term `State` here does not refer to the state itself but rather to a stateful computation that takes in and returns state. This historical naming can be misleading, so please be attentive when using it. Using this formulation, `get` is a stateful computation that retrieves the stored state without modifying it, while `put` is a stateful computation that takes an item, replaces the existing state with that item, and does not produce a value.

One side effect of using the `State` type to represent stateful computations is that the underlying function is wrapped, so it cannot be called directly. For instance, calling `get 5` will produce an error instead of the expected result $(5, 5)$. To address this, Haskell provides the built-in function `runState`, whose sole purpose is to return the wrapped function:

```haskell
-- s is updated state
-- a is the computed result.
runState :: State s a -> (s -> (a, s))
runState (State op) = op
```

In this code, `op` is a function; in order to actually "execute" the stateful computation, we need to provide a state value, which we think of as the initial state.

```
> runState get 5
(5,5)
```

With this approach, we can also chain stateful computations. As an example, let's translate the following C code:

```c
#include <stdio.h>

int main() {
    int x;
    x = 10;
    x = x * 2;

    printf("%d", x);

    return 0;
}
```

The idea is straightforward: apart from variable declaration and initialization, each line involves either a `get`, `put`, or a pure computation (like `(*2)`). Our approach is to perform these computations in sequence, using `let` to bind intermediate states so that each `State` computation receives the state from the previous one.

```haskell
data State s a = State (s -> (a, s))

get :: State s s
get = State (\state -> (state, state))

put :: s -> State s ()
put item = State (\state -> ((), item))

runState :: State s a -> (s -> (a, s))
runState (State op) = op

f :: State Int String -- return string and input is an integer
f = State (\state0 ->
    let (_, state1) = runState (put 10) state0
        (x, state2) = runState get state1
        (_, state3) = runState (put (x * 2)) state2
        (x', state4) = runState get state3
    in
    (show x', state4))

main = print (runState f 5)
```

Let's see how to understand `runState (put 10) state0`:

- runState (put 10) extracts and applies the function from put 10:

```haskell
    runState (State (\state -> ((), 10))) = \state -> ((), 10)
```

- Applying the function to state0:

```haskell
    (\state -> ((), 10)) state0
```

- Pattern match: (_, state1) = ((), 10)

While this code is easy to understand, it can feel cumbersome to manually track the state. Thus, we'll define a higher-order function to simplify composing two `State` operations. This main function, called `andThen`, helps chain computations. Here's an initial version:

```
-- Perform two state computations and return the second one's value.
andThen :: State s a -> State s b -> State s b
andThen op1 op2 = State (\state0 ->
    let (x1, state1) = runState op1 state0
        (x2, state2) = runState op2 state1
        in
        (x2, state2)
        )
```

While this is a good start, it's not yet general enough. The problem is that the "produced" value $x1$ from the first computation is unused in the second computation. Generally, when chaining two stateful computations, we want the second to use the result of the first. To achieve this, we define a function that performs two `State` operations in sequence, with the second operation depending on the first one's result.

Instead of taking two fixed `State` operations, `andThen` will take a first `State` operation and a function that takes the result of the first and returns a new `State` operation. Think of this second parameter as an "incomplete" `State` function that gets completed by the value produced by the first function. Here's the refined type signature and implementation:

```
-- Perform two state computations and return the second one's value.
andThen :: State s a -> (a -> State s b) -> State s b
andThen op1 opMaker = State (\state0 ->
    let (x1, state1) = runState op1 state0  -- Run the first state computation
        op2 = opMaker x1                    -- Use the result to create the next computation
        (x2, state2) = runState op2 state1  -- Run the second computation with the updated state
    in
    (x2, state2))
```

Using `andThen`, we can simplify our original example function:

```
f :: State Int String
f = put 10 `andThen` \_ -> get
    `andThen` \x -> put (x + 2)
    `andThen` \_ -> get
    `andThen` \x' -> State (\state -> (show x', state))

```

Why Use _ ->? If you don't need to use the argument in your computation, but you still need to write a function for syntactic reasons, you use _. This is common in monadic computations where you are chaining functions, but some steps don't care about the result of the previous computation. Let's take a specific step from the function f:

```
put 10 `andThen` \_ -> get
```

Here's what's happening:

- put 10: This updates the state to 10 and produces a State s () (where () is the unit type because put does not return any meaningful value).
- _ -> get: The lambda function _ -> get ignores the result of put 10 (which is ()) and moves on to call get.
- The _ indicates that you don't care about the value returned by put 10 (in this case, it's always ()).

This can be read as: "Set the state to 10, then ignore the result, and get the current state." Why Not Just Write get?

You must use _ -> get because andThen requires a function of type (a -> State s b) for the second argument. Simply writing get doesn't satisfy this because get is a value of type State s s. The lambda function ensures the type matches.

Finally, note that the last `State` value simply produces a value (`show x'`) without modifying the underlying state. We call this a "pure" computation and use a helper function to simplify the code:

```haskell
data State s a = State (s -> (a, s))

get :: State s s
get = State (\state -> (state, state))

put :: s -> State s ()
put item = State (\state -> ((), item))

runState :: State s a -> (s -> (a, s))
runState (State op) = op

andThen :: State s a -> (a -> State s b) -> State s b
andThen op1 opMaker = State (\state0 ->
    let (x1, state1) = runState op1 state0  -- Run the first state computation
        op2 = opMaker x1                    -- Use the result to create the next computation
        (x2, state2) = runState op2 state1  -- Run the second computation with the updated state
    in
    (x2, state2))

f :: State Int String
f = put 10 `andThen` \_ -> get
    `andThen` \x -> put (x + 2)
    `andThen` \_ -> get
    `andThen` \x' -> State (\state -> (show x', state))

main = print (runState f 5)
```

With the tools we've developed, we can implement a post-order binary tree labeling in Haskell. Here is the corresponding data type we'll use to represent binary trees in Haskell. Note the similarity to our recursive list definition!

```haskell
data BTree a = Empty
             | Node a (BTree a) (BTree a)
```

To maintain immutability, we won't mutate a `BTree` directly; instead, we'll return a new `BTree` with all nodes labeled properly. To accomplish this using the same algorithm as shown in the Python code, we'll need an `Int` of mutable state. Thus, our type signature will be:

```haskell
postOrderLabel :: BTree a -> State Int (BTree (a, Int))
```

The base case is straightforward to implement, requiring no statefulness:

```haskell
postOrderLabel Empty = pureValue Empty
```

For the recursive step, let's start by translating the two recursive calls from Python into equivalent chained Haskell code using our `andThen` function.

```haskell
postOrderLabel (Node item left right) =
    postOrderLabel left `andThen` \newLeft ->
    postOrderLabel right
```

Not too bad so far, although these recursive calls don't accomplish much yet (they're merely spawning other calls). Let's increment our "`i`" next:

```haskell
postOrderLabel (Node item left right) =
    postOrderLabel left `andThen` \newLeft ->
    postOrderLabel right `andThen` \newRight ->
    get `andThen` \i ->
    put (i + 1)
```

This version nearly compiles but ends with a `State` value of the incorrect type: `State Int ()` instead of `State Int (BTree (a, Int))`. To fix this, let's clarify what we want to return: a new `BTree` with all nodes labeled correctly.

Through recursion, we can assume that `newLeft` and `newRight` (produced by the recursive calls) are correctly labeled. To complete the solution, we need to create and return a new root node containing the original item, an updated label, and the new subtrees:

```
postOrderLabel (Node item left right) =              -- Recursive case: Node with children
    postOrderLabel left `andThen` \newLeft ->        -- Label the left subtree
    postOrderLabel right `andThen` \newRight ->      -- Label the right subtree
    get `andThen` \i ->                              -- Get the current label (i)
    put (i + 1) `andThen` \_ ->                      -- Increment the label and store it back
    pureValue (Node (item, i) newLeft newRight)      -- Return a new tree node with the labeled value
```

And that's it! This solution is remarkably close to what we'd write in Python if we needed to return a new tree rather than mutate the original. Now combine all we have and run a test case:

```
-- Define the State data type
data State s a = State (s -> (a, s))

-- Helper function to run a State computation
runState :: State s a -> s -> (a, s)
runState (State op) initialState = op initialState

-- Define get, put, and pureValue functions
get :: State s s
get = State (\s -> (s, s))

put :: s -> State s ()
put s = State (\_ -> ((), s))

-- pureValue allows embedding pure values into the State to be composed with other stateful computations.
pureValue :: a -> State s a
pureValue x = State (\s -> (x, s))

-- Implement andThen for chaining State computations
andThen :: State s a -> (a -> State s b) -> State s b
andThen op1 opMaker = State (\state0 ->
    let (x1, state1) = runState op1 state0
        op2 = opMaker x1
        (x2, state2) = runState op2 state1
    in (x2, state2))

-- Define the binary tree data type
data BTree a = Empty | Node a (BTree a) (BTree a)
    deriving (Show)

-- Define the postOrderLabel function
postOrderLabel :: BTree a -> State Int (BTree (a, Int))
postOrderLabel Empty = pureValue Empty
postOrderLabel (Node item left right) =
    postOrderLabel left `andThen` \newLeft ->
    postOrderLabel right `andThen` \newRight ->
    get `andThen` \i ->
    put (i + 1) `andThen` \_ ->
    pureValue (Node (item, i) newLeft newRight)

-- Example test tree
testTree :: BTree String
testTree = Node "root"
            (Node "left"
                (Node "left.left" Empty Empty)
                (Node "left.right" Empty Empty)
            )
            (Node "right" Empty Empty)

-- Function to run the postOrderLabel test
-- fst :: (a, b) -> a
```

```haskell
-- It is a built-in function and extracts the first element of a pair.
runPostOrderLabelTest :: BTree String -> BTree (String, Int)
runPostOrderLabelTest tree = fst (runState (postOrderLabel tree) 0)



-- Main function to display the result
main :: IO ()
main = do
    let labeledTree = runPostOrderLabelTest testTree
    print labeledTree
```

The output is

```
Node ("root",4) (Node ("left",2) (Node ("left.left",0) Empty Empty) (Node ("left.right",1) Empty
    Empty)) (Node ("right",3) Empty Empty)
```

You can verify it by yourself, you will find it is correct.

8 Reference

[1] David, Liu. "CSC324, Principles of Programming Languages" *Teaching Notes*, University of Toronto chrome-extension://efaidnbmnnnibpcajpcglclefindmkaj/https://www.cs.toronto.edu/~david/course-notes/csc324.pdf