Advanced Swift

Extending the language

Session 404

John McCall
Horrible Nerd

Dave Abrahams
Horrible Nerd

© 2014 Apple Inc. All rights reserved. Redistribution or public display not permitted without written permission from Apple.

Advanced Swift

Taking control of syntax

Generic programming

The Swift model

Taking Control of Syntax

John McCall
Syntax Artist

West of House

> _

West of House

> LOOK AROUND

You're standing in front of a quaint cottage, near the edge of a broad forest. A path leads around to the south.

> _

West of House

```
> LOOK AROUND
You're standing in front of a quaint cottage, near the edge of a broad
forest. A path leads around to the south.
> DESCRIBE COTTAGE
A fine example of crumbling-plaster-on-rotten-wood construction. The door has
been freshly boarded up.
> _
```

West of House

```
> LOOK AROUND
You're standing in front of a quaint cottage, near the edge of a broad
forest. A path leads around to the south.
> DESCRIBE COTTAGE
A fine example of crumbling-plaster-on-rotten-wood construction. The door has
been freshly boarded up.
> LOOK AT BOARDS
They've been nailed directly into the wall.
> _
```

```
West of House
> LOOK AROUND
You're standing in front of a quaint cottage, near the edge of a broad
forest. A path leads around to the south.
> DESCRIBE COTTAGE
A fine example of crumbling-plaster-on-rotten-wood construction. The door has
been freshly boarded up.
> LOOK AT BOARDS
They've been nailed directly into the wall.
> PULL BOARDS
They come off with little effort.
> _
```

West of House

> LOOK AROUND

You're standing in front of a quaint cottage, near the edge of a broad forest. A path leads around to the south.

> DESCRIBE COTTAGE

A fine example of crumbling-plaster-on-rotten-wood construction. The door has been freshly boarded up.

> LOOK AT BOARDS

They've been nailed directly into the wall.

> PULL BOARDS

They come off with little effort.

> _

The Thing Class

```
class Thing {
  init(location: Thing?, name: String,
       longDescription: String) { ... }

  weak var location: Thing?
  var name: String
  var longDescription: String
}
```

Ordinary Things

```
let wallWestOfHouse = Thing(location: westOfHouse,
        name: "wall",
        longDescription: "The plaster has crumbled " +
                "away, leaving the wood beneath to rot.")

let pathWestOfHouse = Thing(location: westOfHouse,
        name: "path",
        longDescription: "An overgrown path leads " +
                 "south around the corner of the house.")
```

Argument Names

```
let wallWestOfHouse = Thing(location: westOfHouse,
        name: "wall",
        longDescription: "The plaster has crumbled " +
                "away, leaving the wood beneath to rot.")

let pathWestOfHouse = Thing(location: westOfHouse,
        name: "path",
        longDescription: "An overgrown path leads " +
                 "south around the corner of the house.")
```

Argument Names

```
let wallWestOfHouse = Thing(location: westOfHouse,
        name: "wall",
        longDescription: "The plaster has crumbled " +
                "away, leaving the wood beneath to rot.")

let pathWestOfHouse = Thing(location: westOfHouse,
        name: "path",
        longDescription: "An overgrown path leads " +
                 "south around the corner of the house.")
```

Changing Argument Names

```
class Thing {
  init(location: Thing?, name: String,
       longDescription: String) { ... }
}
```

Changing Argument Names

```
class Thing {
  init(location location: Thing?, name name: String,
       longDescription longDescription: String) { ... }
}
```

Making Something Anonymous

```
for (key, _) in dictionary {
  println(key)
}
```

Making Something Anonymous

```
var red, blue: Int
(red, _, blue, _) = color.rgba
```

Removing Argument Names

```
class Thing {
  init(location location: Thing?, name name: String,
       longDescription longDescription: String) { ... }
}
```

Removing Argument Names

```
class Thing {
  init(         location: Thing?,      name: String,
                          longDescription: String) { ... }
}
```

Removing Argument Names

```
class Thing {
  init(_        location: Thing?, _    name: String,
       _                 longDescription: String) { ... }
}
```

Removing Keyword Arguments

```
let wallWestOfHouse = Thing(westOfHouse, "wall",
        "The plaster has crumbled away, leaving the " +
        "wood beneath to rot.")

let pathWestOfHouse = Thing(westOfHouse, "garden path",
        "An overgrown path leads south around the " +
        "corner of the house.")
```

Special Objects

```
class Boards: Thing {
  // fill me in
}
let boards = Boards(westWallOfHouse, "boards",
          "They've been nailed directly to the wall.")
```

```
> PULL BOARDS
They come off with little effort.
```

Pulling Objects

```
// The parser will call this.
func performPull(object: Thing) {




}
```

```
> PULL BOARDS
They come off with little effort.
```

Pulling Objects

```
// The parser will call this.
func performPull(object: Thing) {
  if /* object is pullable */ {
    /* pull it */
  } else {
    /* complain */
  }
}
```

```
> PULL BOARDS
They come off with little effort.
```

Simple Protocols

Simple Protocols

```
protocol Pullable {
  func pull()
}
```

Adopting a Protocol

```
class Boards: Thing, Pullable {
}
```

Adopting a Protocol

```
class Boards: Thing, Pullable {
}
```

error: type 'Boards' does not conform to protocol 'Pullable'
note: protocol requires function 'pull()' with type '() -> ()'

Adopting a Protocol

```
class Boards: Thing, Pullable {
  func pull() {
    if location === westWallOfHouse {
      println("They come off with little effort.")
      location = westOfHouse
    } else {
      println("Think of the splinters!")
    }
  }
}
```

Adopting a Protocol

```
class Boards: Thing, Pullable {
  func pull() {
    if location === westWallOfHouse {
      println("They come off with little effort.")
      location = westOfHouse
    } else {
      println("Think of the splinters!")
    }
  }
}
```

Adopting a Protocol

```
class Boards: Thing, Pullable {
  func pull() {
    if location === westWallOfHouse {
      println("They come off with little effort.")
      location = westOfHouse
    } else {
      println("Think of the splinters!")
    }
  }
}
```

Adopting a Protocol

```
class Boards: Thing, Pullable {
  func pull() {
    if location === westWallOfHouse {
      println("They come off with little effort.")
      location = westOfHouse
    } else {
      println("Think of the splinters!")
    }
  }
}
```

Using a Protocol Value

```
func performPull(object: Thing) {
 if /* object is pullable */                {
    /* pull it */
  } else {
    /* complain */
  }
}
```

Using a Protocol Value

```
func performPull(object: Thing) {
 if let pullableObject = object as Pullable {
    /* pull it */
  } else {
    /* complain */
  }
}
```

Using a Protocol Value

```
func performPull(object: Thing) {
 if let pullableObject = object as Pullable {
    pullableObject.pull()
  } else {
    /* complain */
  }
}
```

Using a Protocol Value

```
func performPull(object: Thing) {
 if let pullableObject = object as Pullable {
    pullableObject.pull()
  } else {
    println("You aren't sure how to print a \(object.name).")
  }
}
```

String Interpolation

```
println("You aren't sure how to pull a \(object.name).")
```

```
> PULL PATH
You aren't sure how to pull a garden path.
```

String Interpolation

```
println("You aren't sure how to pull a \(object).")
```

String Interpolation

```
println("You aren't sure how to pull a \(object).")
```

```
> PULL PATH
You aren't sure how to pull a DemoApp.Thing
(has 2 children).
```

Special Protocols

```
LogicValue                          if logicValue {
Printable                           "\(printable)"
Sequence                            for x in sequence
IntegerLiteralConvertible           65536
FloatLiteralConvertible             1.0
StringLiteralConvertible            "abc"
ArrayLiteralConvertible             [ a, b, c ]
DictionaryLiteralConvertible        [ a: x, b: y ]
```

Printable

```
protocol Printable {
  var description: String { get }
}
```

Adopting Printable

```
extension Thing: Printable {
  var description: String { return name }
}
```

String Interpolation

```
println("You aren't sure how to pull a \(object).")
```

```
> PULL PATH
You aren't sure how to pull a garden path.
```

Decorating a Printed String

```
println("You aren't sure how to pull a \(object).")
```

```
> PULL MUD
You aren't sure how to pull a mud.
```

```
> PULL ORANGE
You aren't sure how to pull a orange.
```

Decorating a Printed String

```
extension Thing {
  var nameWithArticle: String {
    return "a " + name
  }
}

println("You aren't sure how to pull \(object.nameWithArticle).")
```

Decorating a Printed String

```
extension Thing {
  var nameWithArticle: String {
    return "a " + name
  }
}

println("You aren't sure how to pull \(an object).")
```

Decorating a Printed String

```
extension Thing {
  var nameWithArticle: String {
    return "a " + name
  }
}

println("You aren't sure how to pull \(an ~ object).")
```

Overloading an Operator

```
func ~ (decorator: ???,
        object: Thing) -> String {
}
```

Overloading an Operator

```
func ~ (decorator: ???,
        object: Thing) -> String {
}
```

error: operator implementation without matching operator declaration

Adding a New Operator

```
operator infix ~ {}

func ~ (decorator: ???,
        object: Thing) -> String {
}
```

Defining an Operator

```
operator infix ~ {}

func ~ (decorator: ???,
        object: Thing) -> String {
}
```

Handling "an"

```
println("You aren't sure how to pull \(an ~ object).")
```

Decorating a Printed String

```
func an(object: Thing) -> String {
  return object.nameWithArticle
}
```

Defining an Operator

```
operator infix ~ {}

func ~ (decorator: (Thing) -> String,
        object: Thing) -> String {

}

func an(object: Thing) -> String {
  return object.nameWithArticle
}
```

Defining an Operator

```
operator infix ~ {}

func ~ (decorator: (Thing) -> String,
        object: Thing) -> String {
  return decorator(object)
}

func an(object: Thing) -> String {
  return object.nameWithArticle
}
```

Decorating a Printed String

```
println("You aren't sure how to pull \(an ~ object).")
```

```
> PULL MUD
You aren't sure how to pull mud.
```

```
> PULL ORANGE
You aren't sure how to pull an orange.
```

West of House

> _

West of House

> SOUTH

Garden Path

What was once a tidy walled garden is now an overgrown thicket.
A path leads out a crack in the wall to the east.

> _

```
West of House
> SOUTH
Garden Path
What was once a tidy walled garden is now an overgrown thicket.
A path leads out a crack in the wall to the east.
> EAST
Forest
Tall oaks cast the forest floor into deep shadow.
> _
```

Defining a Place

```
enum Direction {
  case North, South, East, West
}

class Place: Thing {
  init(_ name: String, _ longDescription: String) { ... }

  var exits: Dictionary<Direction, Place>
}
```

Making Connections

```
let westOfHouse = Place("West of House",
              "You're standing in front of a quaint cottage, near the " +
              "edge of a broad forest. A path leads around to the south.")
let gardenPath = Place("Garden Path",
              "What was once a tidy walled garden is now an overgrown " +
              "thicket. A path leads out a crack in the wall to the east.")
let forest = Place("Forest",
              "Tall oaks cast the forest floor into deep shadow.")

westOfHouse.exits[.South] = gardenPath
gardenPath.exits[.North] = westOfHouse
gardenPath.exits[.East] = forest
```

Making Connections

```
let westOfHouse = Place("West of House",
               "You're standing in front of a quaint cottage, near the " +
               "edge of a broad forest. A path leads around to the south.")
let gardenPath = Place("Garden Path",
               "What was once a tidy walled garden is now an overgrown " +
               "thicket. A path leads out a crack in the wall to the east.")
let forest = Place("Forest",
               "Tall oaks cast the forest floor into deep shadow.")

westOfHouse[.South] = gardenPath
gardenPath[.North] = westOfHouse
gardenPath[.East] = forest
```

Subscripts

```
extension Place {





}
```

Subscripts

```
extension Place {
  subscript







}
```

Subscripts

```
extension Place {
  subscript(direction: Direction)





}
```

Subscripts

```
extension Place {
  subscript(direction: Direction) -> Place? {







  }
}
```

Subscripts

```
extension Place {
  subscript(direction: Direction) -> Place? {
    get {
      return exits[direction]
    }



  }
}
```

Subscripts

```
extension Place {
  subscript(direction: Direction) -> Place? {
    get {
      return exits[direction]
    }
    set(destination: Place?) {
      exits[direction] = destination
    }
  }
}
```

Making Connections

```
let westOfHouse = Place("West of House",
               "You're standing in front of a quaint cottage, near the " +
               "edge of a broad forest. A path leads around to the south.")
let gardenPath = Place("Garden Path",
               "What was once a tidy walled garden is now an overgrown " +
               "thicket. A path leads out a crack in the wall to the east.")
let forest = Place("Forest",
               "Tall oaks cast the forest floor into deep shadow.")

westOfHouse[.South] = gardenPath
gardenPath[.North] = westOfHouse
gardenPath[.East] = forest
```

A Word of Caution

A Word of Caution

Keep it natural

A Word of Caution

Keep it natural

Work by analogy

A Word of Caution

Keep it natural

Work by analogy

New idioms should pay for themselves

Swift Generics

Dave Abrahams
Operator Overlord

A public service of Apple DevTools

True Story (I Swear)

```
func peekString(interestingValue: String) {
  println("[peek] \(interestingValue)")
}
func peekInt(interestingValue: Int) {
  println("[peek] \(interestingValue)")
}
func peekFloat(interestingValue: Float) {
  println("[peek] \(interestingValue)")
}
```

True Story (I Swear)

```
func peekString(interestingValue: String) {
  println("[peek] \(interestingValue)")
}
func peekInt(interestingValue: Int) {
  println("[peek] \(interestingValue)")
}
func peekFloat(interestingValue: Float) {
  println("[peek] \(interestingValue)")
}
```

True Story (I Swear)

```
func peekString(interestingValue: String) {
  println("[peek] \(interestingValue)")
}
func peekInt(interestingValue: Int) {
  println("[peek] \(interestingValue)")
}
func peekFloat(interestingValue: Float) {
  println("[peek] \(interestingValue)")
}
-------------------------------------------------------------------------------
  peekString(window.title)                        // [peek] Xyzzy – Untitled
  peekInt(window.orderedIndex)                    // [peek] 3
  peekFloat(window.alphaValue)                    // [peek] 0.5
```

True Story (I Swear)

```
func peekString(interestingValue: String) {
  println("[peek] \(interestingValue)")
}
func peekInt(interestingValue: Int) {
  println("[peek] \(interestingValue)")
}
func peekFloat(interestingValue: Float) {
  println("[peek] \(interestingValue)")
}
--------------------------------------------------------------------------------
  peekString(window.title)                         // [peek] Xyzzy — Untitled
  peekInt(window.orderedIndex)                     // [peek] 3
  peekFloat(window.alphaValue)                     // [peek] 0.5
```

Function Overloading

```
func peek(interestingValue: String) {
  println("[peek] \(interestingValue)")
}
func peek(interestingValue: Int) {
  println("[peek] \(interestingValue)")
}
func peek(interestingValue: Float) {
  println("[peek] \(interestingValue)")
}
--------------------------------------------------------------------------------
  peek(window.title)                         // [peek] Xyzzy – Untitled
  peek(window.orderedIndex)                  // [peek] 3
  peek(window.alphaValue)                    // [peek] 0.5
```

```
ar box: Any[]
ar count: Int
```

ar box: Any[]
ar count: Int

Using Any

```
func peek(interestingValue: String) {
  println("[peek] \(interestingValue)")
}
func peek(interestingValue: Int) {
  println("[peek] \(interestingValue)")
}
func peek(interestingValue: Float) {
  println("[peek] \(interestingValue)")
}
-------------------------------------------------------------------------------------
  peek(window.title)                            // [peek] Xyzzy — Untitled
  peek(window.orderedIndex)                     // [peek] 3
  peek(window.alphaValue)                       // [peek] 0.5
```

Using Any

```
func peek(interestingValue: Any) {
  println("[peek] \(interestingValue)")
}
```

```
peek(window.title)                          // [peek] Xyzzy — Untitled
peek(window.orderedIndex)                   // [peek] 3
peek(window.alphaValue)                     // [peek] 0.5
```

Using Any

```
func peek(interestingValue: Any) {
  println("[peek] \(interestingValue)")
}
```

```
------------------------------------------------------------------------------------------
  peek(window.title)                                    // [peek] Xyzzy — Untitled
  peek(window.orderedIndex)                             // [peek] 3
  peek(window.alphaValue)                               // [peek] 0.5
  peek(window.frame.origin)                             // [peek] (4.0,5.0)
```

Using Any

```
func peek(interestingValue: Any) {
  println("[peek] \(interestingValue)")
}
```

```
  peek(window.title)                          // [peek] 🐶🐮 - Moof
  peek(window.orderedIndex)                   // [peek] 3
  peek(window.alphaValue)                     // [peek] 0.5
  peek(window.frame.origin)                   // [peek] (4.0,5.0)
```

Using Any

```
func peek(interestingValue: Any) {
  println("[peek] \(interestingValue)")
}

extension String {
  var sansEmoji: String { … }
}
```

```
peek(window.title)                          // [peek] 🐶🐮 – Moof
```

True Story (I Swear)

```
func peek(interestingValue: Any) {
  println("[peek] \(interestingValue)")
}

extension String {
  var sansEmoji: String { … }
}
-------------------------------------------------------------------------------
  window.title = window.document.fileURL
    .lastPathComponent.capitalizedString.sansEmoji
    + " – " + window.document.displayName

  peek(window.title)                              // [peek] 🐶🐮 – Moof
```

True Story (I Swear)

```
func peek(interestingValue: Any) {
  println("[peek] \(interestingValue)")
}

extension String {
  var sansEmoji: String { … }
}
-------------------------------------------------------------------------------
  window.title = window.document.fileURL
    .lastPathComponent.capitalizedString.sansEmoji
    + " – " + window.document.displayName

  peek(window.title)                          // [peek] 🐶🐮 – Moof
```

True Story (I Swear)

```
func peek(interestingValue: Any) {
  println("[peek] \(interestingValue)")
}

extension String {
  var sansEmoji: String { … }
}
```

```
  let lastPathComponent = window.document.fileURL.lastPathComponent
  peek(lastPathComponent)
  window.title = lastPathComponent.capitalizedString.sansEmoji
    + " – " + window.document.displayName
```

True Story (I Swear)

```
func peek(interestingValue: Any) -> Any {
  println("[peek] \(interestingValue)")
  return interestingValue
}
extension String {
  var sansEmoji: String { … }
}
```

```
  window.title = window.document.fileURL
    .lastPathComponent.capitalizedString.sansEmoji
    + " — " + window.document.displayName
```

True Story (I Swear)

```
func peek(interestingValue: Any) -> Any {
  println("[peek] \(interestingValue)")
  return interestingValue
}
extension String {
  var sansEmoji: String { … }
}
```

```
  window.title = peek(window.document.fileURL
    .lastPathComponent).capitalizedString.sansEmoji
    + " – " + window.document.displayName
```

True Story (I Swear)

```
func peek(interestingValue: Any) -> Any {
  println("[peek] \(interestingValue)")
  return interestingValue
}
extension String {
  var sansEmoji: String { … }
}
```

```
  window.title = peek(window.document.fileURL
    .lastPathComponent).capitalizedString.sansEmoji
    + " — " + window.document.displayName
```

error: 'Any' does not have a member named 'capitalizedString'

True Story (I Swear)

```
func peek(interestingValue: Any) -> Any {
  println("[peek] \(interestingValue)")
  return interestingValue
}
extension String {
  var sansEmoji: String { … }
}
```

```
  window.title = peek(window.document.fileURL
    .lastPathComponent).capitalizedString.sansEmoji
    + " - " + window.document.displayName
```

error: 'Any' does not have a member named 'capitalizedString'

True Story (I Swear)

```
func peek(interestingValue: Any) -> Any {
  println("[peek] \(interestingValue)")
  return interestingValue
}
extension String {
  var sansEmoji: String { … }
}
```

```
  window.title = (peek(window.document.fileURL
    .lastPathComponent) as String).capitalizedString.sansEmoji
    + " – " + window.document.displayName
```

True Story (I Swear)

```
func peek<T>(interestingValue: T) -> T {
  println("[peek] \(interestingValue)")
  return interestingValue
}
extension String {
  var sansEmoji: String { … }
}
```

```
  window.title = peek(window.document.fileURL
    .lastPathComponent).capitalizedString.sansEmoji
    + " — " + window.document.displayName
```

Protocol Types vs. Swift Generics

Protocol types like Any or Pullable throw away type information

- Great for Dynamic Polymorphism

Swift Generics conserve type information

- Great for type safety
- Great for performance

Type Relationships

```swift
// Exchange the values of x and y
func swap<T>(inout x: T, inout y: T) {
  let tmp = x
  x = y
  y = tmp
}
```

Type Relationships

```
// Exchange the values of x and y
func swap<T>(inout x: T, inout y: T) {
  let tmp = x
  x = y
  y = tmp
}

var studentCount = 42
var teacherCount = 7
swap(&studentCount, &teacherCount) // OK
```

Type Relationships

```
// Exchange the values of x and y
func swap<T>(inout x: T, inout y: T) {
  let tmp = x
  x = y
  y = tmp
}

var studentCount = 42
var teacherCount = 7
swap(&studentCount, &teacherCount) // OK

var schoolName = “Homestead High School"
swap(&studentCount, &schoolName)
```

Type Relationships

```
// Exchange the values of x and y
func swap<T>(inout x: T, inout y: T) {
  let tmp = x
  x = y
  y = tmp
}

var studentCount = 42
var teacherCount = 7
swap(&studentCount, &teacherCount) // OK

var schoolName = “Homestead High School"
swap(&studentCount, &schoolName)
```

error: 'Int' is not identical to 'String'

Finding a String in an Array

```
// Return the  first index of sought in array, or nil if not found
func indexOfString(sought: String, inArray array: String[]) -> Int? {
  for i in 0..array.count {
    if array[i] == sought {
      return i
    }
  }
  return nil
}
```

Finding a String in an Array

```
// Return the  first index of sought in array, or nil if not found
func indexOfString(sought: String, inArray array: String[]) -> Int? {
  for i in 0..array.count {
    if array[i] == sought {
      return i
    }
  }
  return nil
}
```

Finding an Item in an Array

```
// Return the  first index of sought in array, or nil if not found
func indexOf<T>(sought: T, inArray array: T[]) -> Int? {
  for i in 0..array.count {
    if array[i] == sought {
      return i
    }
  }
  return nil
}
```

Finding an Item in an Array

```
// Return the  first index of sought in array, or nil if not found
func indexOf<T>(sought: T, inArray array: T[]) -> Int? {
  for i in 0..array.count {
    if array[i] == sought {
      return i
    }
  }
  return nil
}
```

Finding an Item in an Array

```
// Return the  first index of sought in array, or nil if not found
func indexOf<T>(sought: T, inArray array: T[]) -> Int? {
  for i in 0..array.count {
    if array[i] == sought {
      return i
    }
  }
  return nil
}
```

error: could not find an overload for '=='
that accepts the supplied arguments

Protocol Constraints

```
// Return the  first index of sought in array, or nil if not found
func indexOf<T : Equatable>(sought: T, inArray array: T[]) -> Int? {
  for i in 0..array.count {
    if array[i] == sought {
      return i
    }
  }
  return nil
}
```

Protocol Constraints

```
// Return the  first index of sought in array, or nil if not found
func indexOf<T : Equatable>(sought: T, inArray array: T[]) -> Int? {
  for i in 0..array.count {
    if array[i] == sought {
      return i
    }
  }
  return nil
}
```

Protocol Constraints

```
// Return the  first index of sought in array, or nil if not found
func indexOf<T : Equatable>(sought: T, inArray array: T[]) -> Int? {
  for i in 0..array.count {
    if array[i] == sought {
      return i
    }
  }
  return nil
}
```

Inside Equatable

```
// Equality comparison protocol. Requires: == is an equivalence relation
protocol Equatable {
  func == (lhs: Self, rhs: Self) -> Bool
}
```

Inside Equatable

```
// Equality comparison protocol. Requires: == is an equivalence relation
protocol Equatable {
  func == (lhs: Self, rhs: Self) -> Bool
}
```

Inside Equatable

```
// Equality comparison protocol. Requires: == is an equivalence relation
protocol Equatable {
  func == (lhs: Self, rhs: Self) -> Bool
}
```

Implementing Equatable

```
// Equality comparison protocol. Requires: == is an equivalence relation
protocol Equatable {
  func == (lhs: Self, rhs: Self) -> Bool
}
```

```
struct Temperature : Equatable {
  let value: Int = 0
}

func == (lhs: Temperature, rhs: Temperature) -> Bool {
  return lhs.value == rhs.value
}
```

Implementing Equatable

```
// Equality comparison protocol. Requires: == is an equivalence relation
protocol Equatable {
  func == (lhs: Self, rhs: Self) -> Bool
}
```

```
struct Temperature : Equatable {
  let value: Int = 0
}

func == (lhs: Temperature, rhs: Temperature) -> Bool {
  return lhs.value == rhs.value
}
```

Implementing Equatable

```
// Equality comparison protocol. Requires: == is an equivalence relation
protocol Equatable {
  func == (lhs: Self, rhs: Self) -> Bool
}
```

```
struct Temperature : Equatable {
  let value: Int = 0
}

func == (lhs: Temperature, rhs: Temperature) -> Bool {
  return lhs.value == rhs.value
}
```

Implementing Equatable

```
// Equality comparison protocol. Requires: == is an equivalence relation
protocol Equatable {
  func == (lhs: Self, rhs: Self) -> Bool
}
```

```
struct Temperature : Equatable {
  let value: Int = 0
}

func == (lhs: Temperature, rhs: Temperature) -> Bool {
  return lhs.value == rhs.value
}
```

Where's "Not-Equal?"

```
// Equality comparison protocol. Requires: == is an equivalence relation
protocol Equatable {
  func == (lhs: Self, rhs: Self) -> Bool
}

func != <T: Equatable>(lhs: T, rhs: T) -> Bool {
  return !(lhs == rhs)
}
```

Where's "Not-Equal?"

```
// Equality comparison protocol. Requires: == is an equivalence relation
protocol Equatable {
  func == (lhs: Self, rhs: Self) -> Bool
}

func != <T : Equatable>(lhs: T, rhs: T) -> Bool {
  return !(lhs == rhs)
}
```

Example

Approximating φ (phi), a.k.a. the Golden Mean

Approximating φ

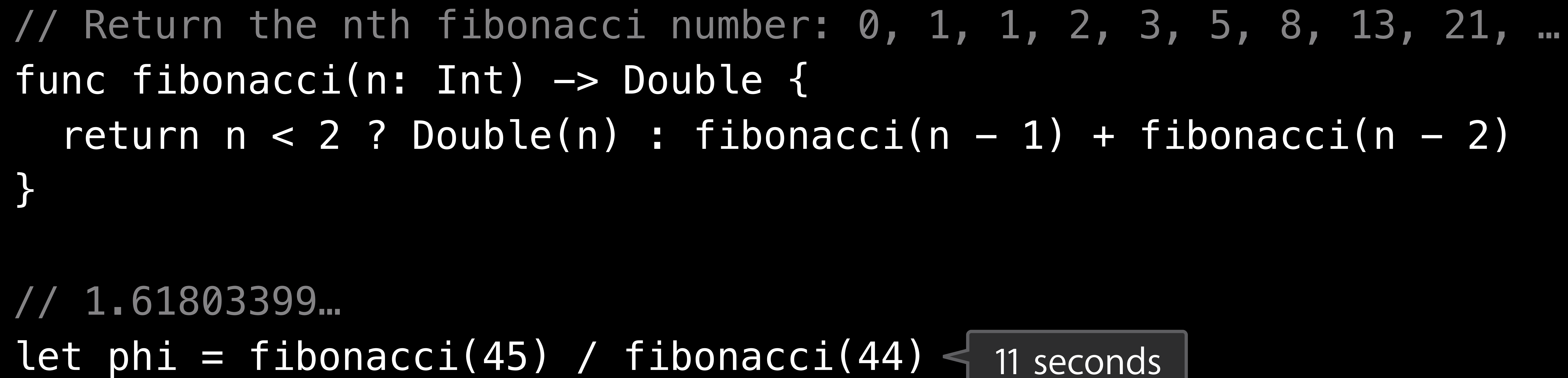
```
// Return the nth fibonacci number: 0, 1, 1, 2, 3, 5, 8, 13, 21, …
func fibonacci(n: Int) -> Double {
  return n < 2 ? Double(n) : fibonacci(n - 1) + fibonacci(n - 2)
}
```

Approximating φ

```
// Return the nth fibonacci number: 0, 1, 1, 2, 3, 5, 8, 13, 21, …
func fibonacci(n: Int) -> Double {
  return n < 2 ? Double(n) : fibonacci(n - 1) + fibonacci(n - 2)
}

// 1.61803399…
let phi = fibonacci(45) / fibonacci(44)
```

Approximating φ



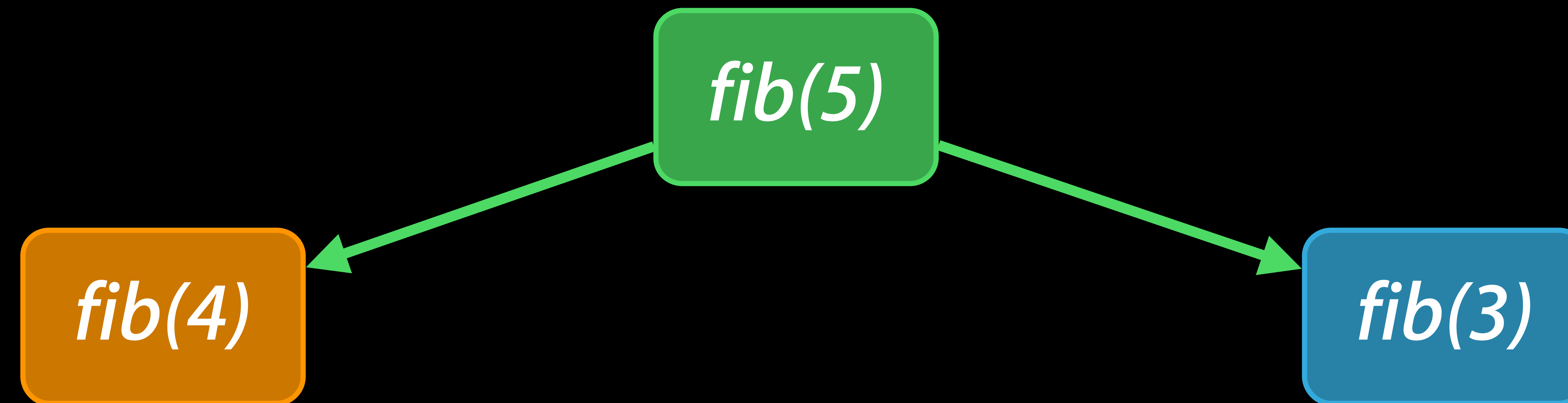
Recursive Fibonacci

Call Graph

fib(5)

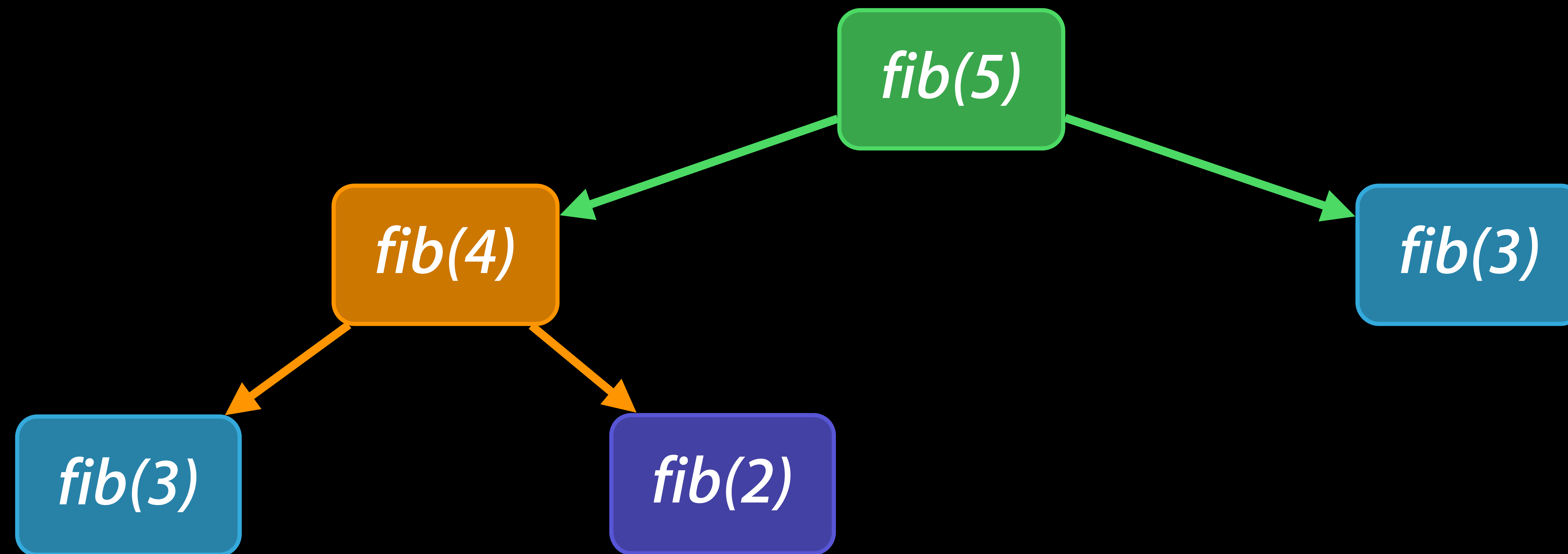
Recursive Fibonacci

Call Graph



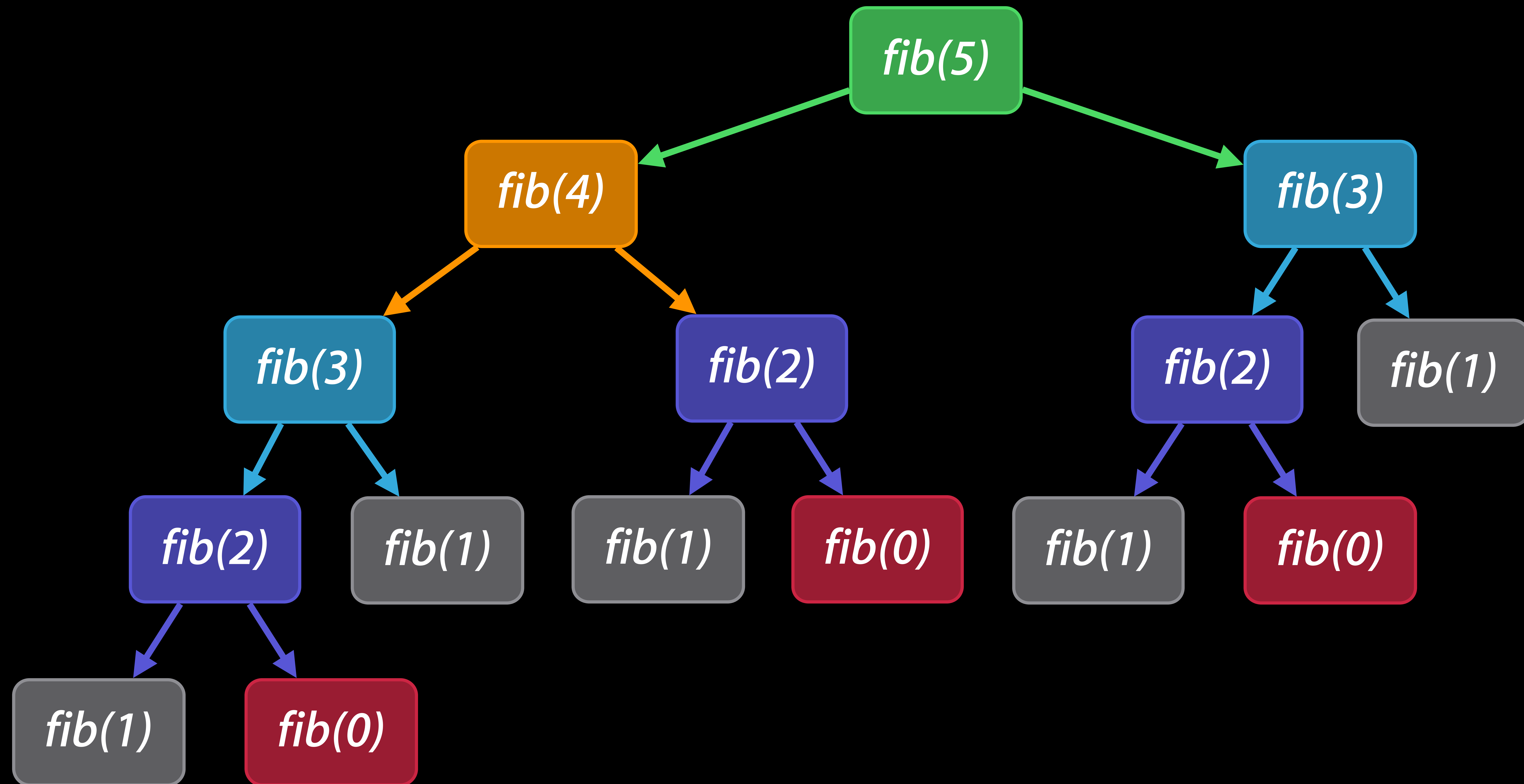
Recursive Fibonacci

Call Graph



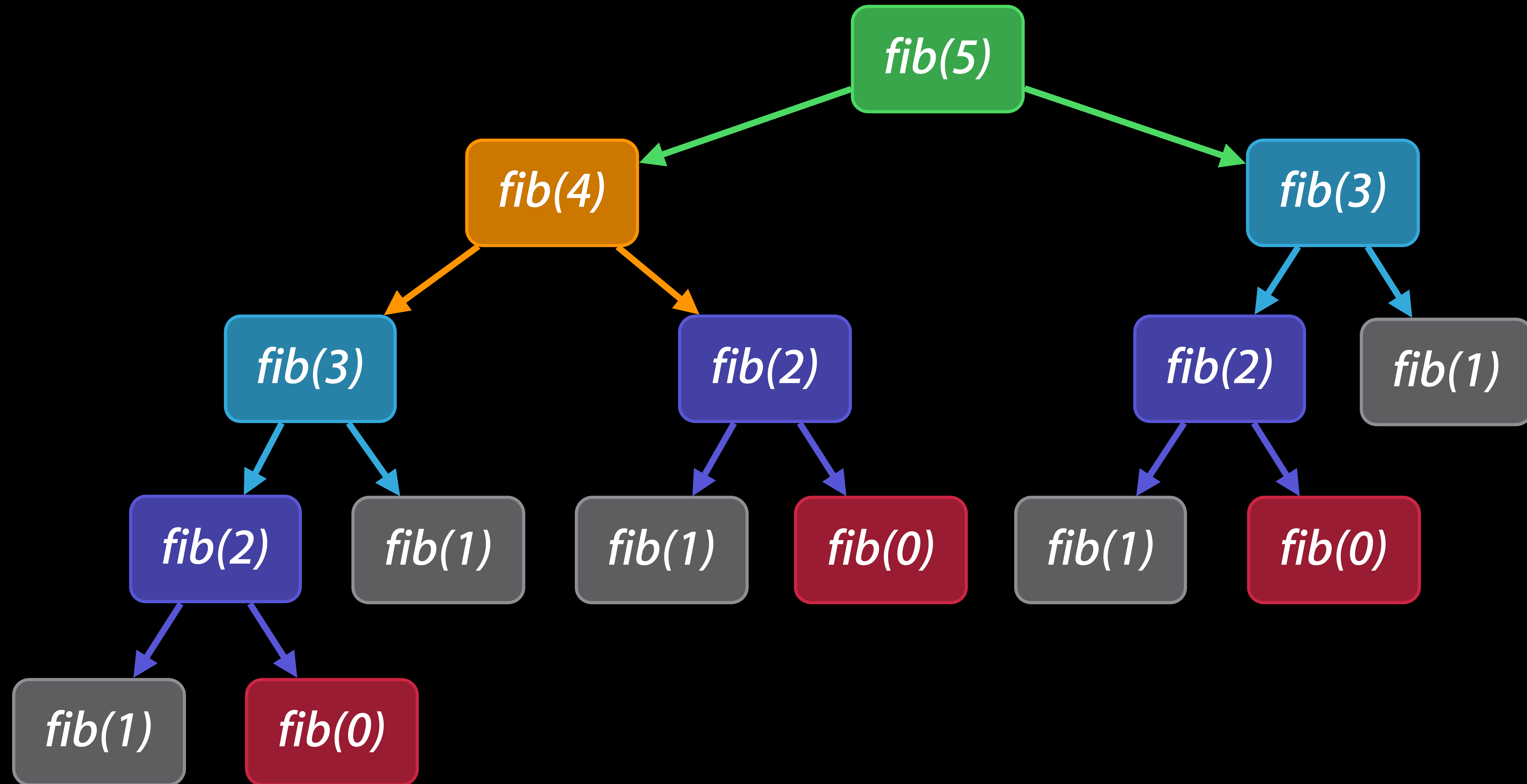
Recursive Fibonacci

Call Graph



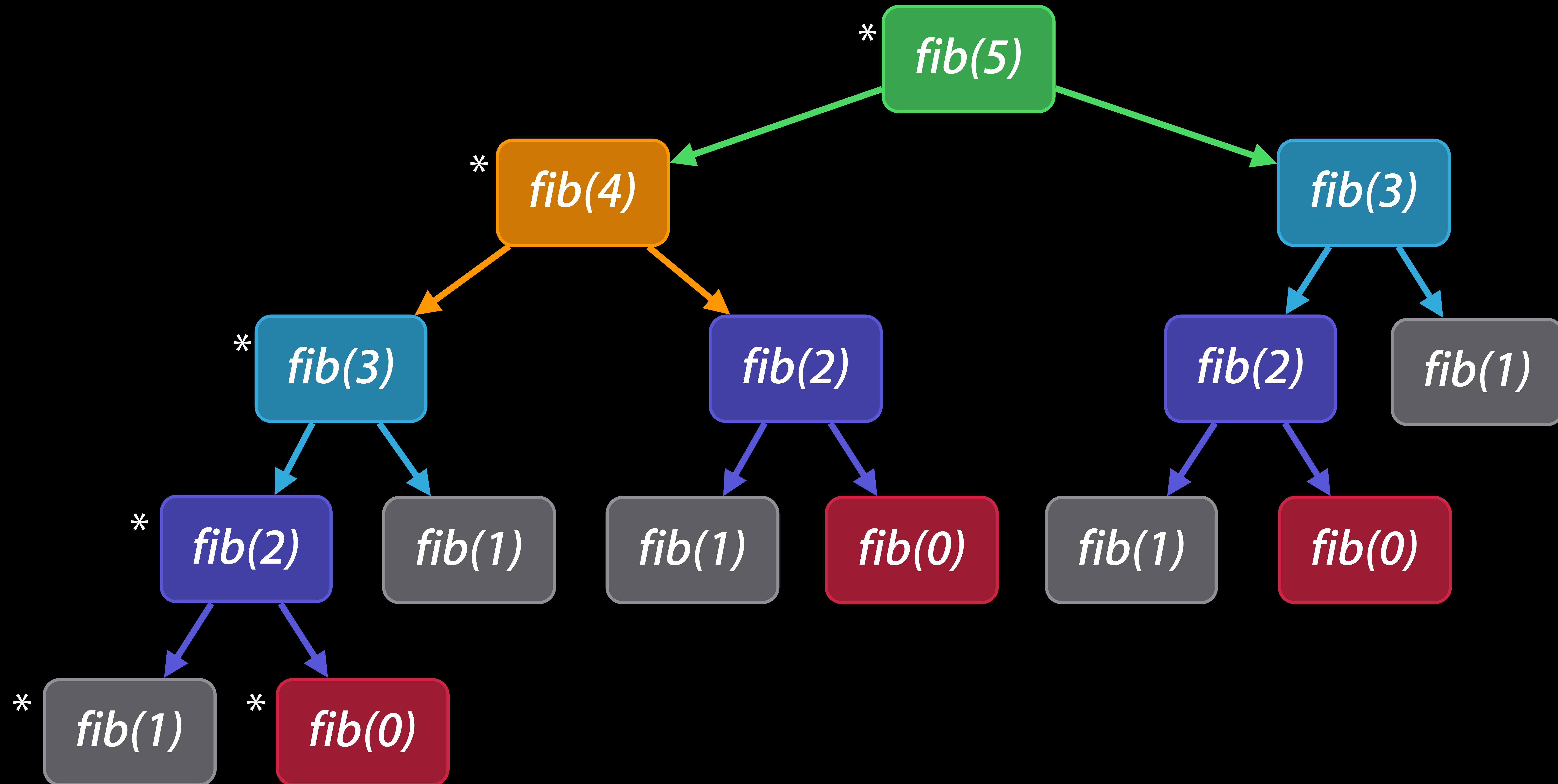
Memoized Fibonacci

Call Graph



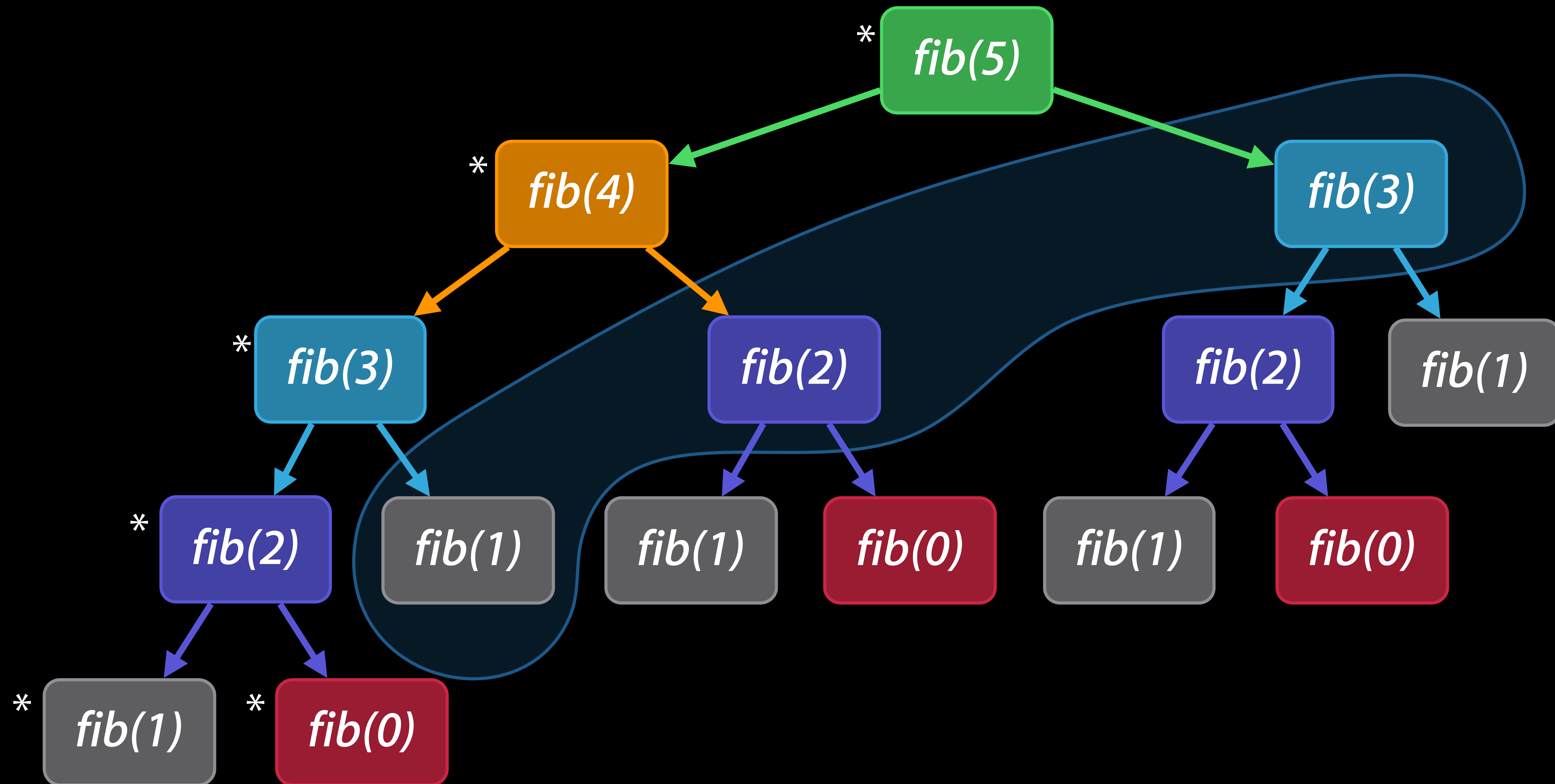
Memoized Fibonacci

Call Graph



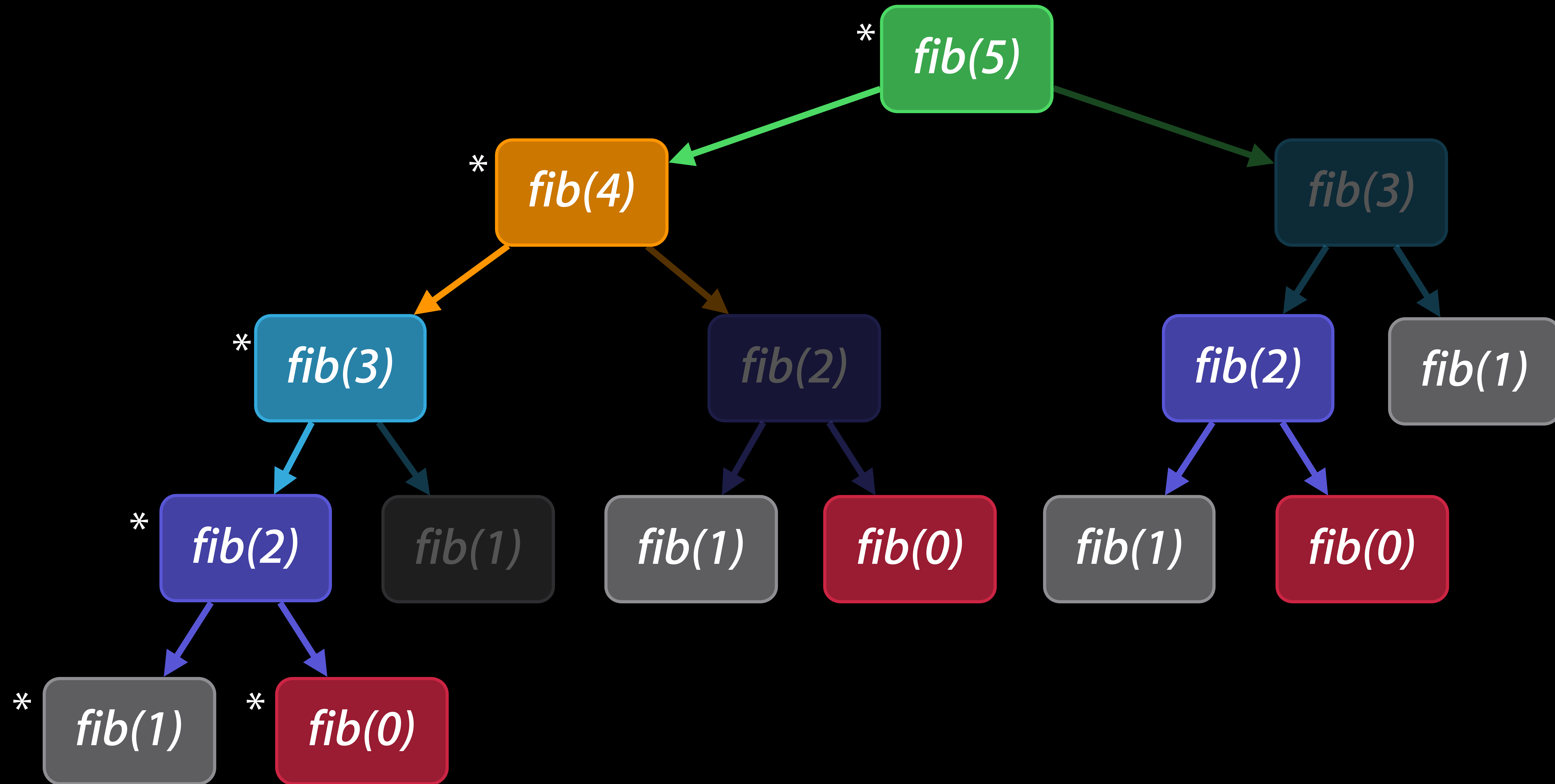
Memoized Fibonacci

Call Graph



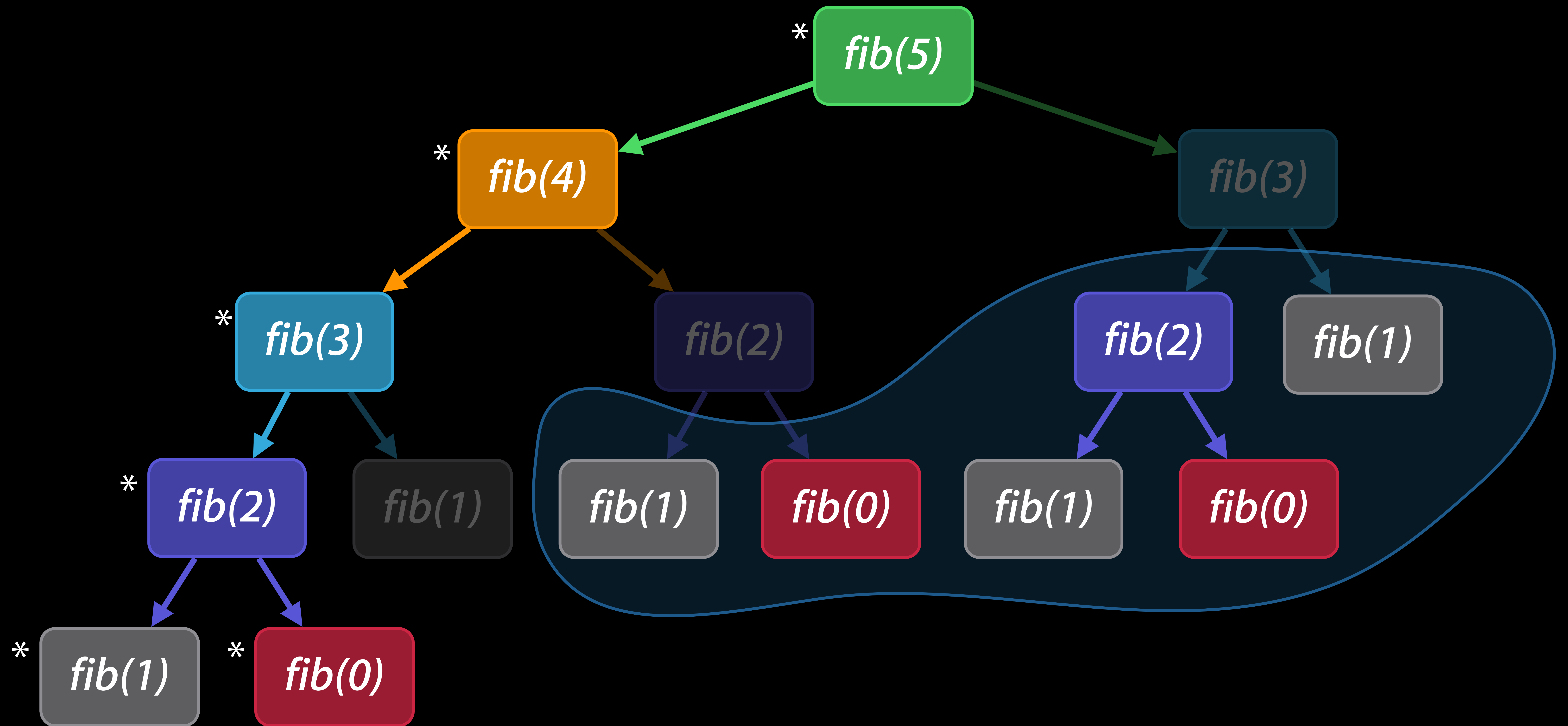
Memoized Fibonacci

Call Graph



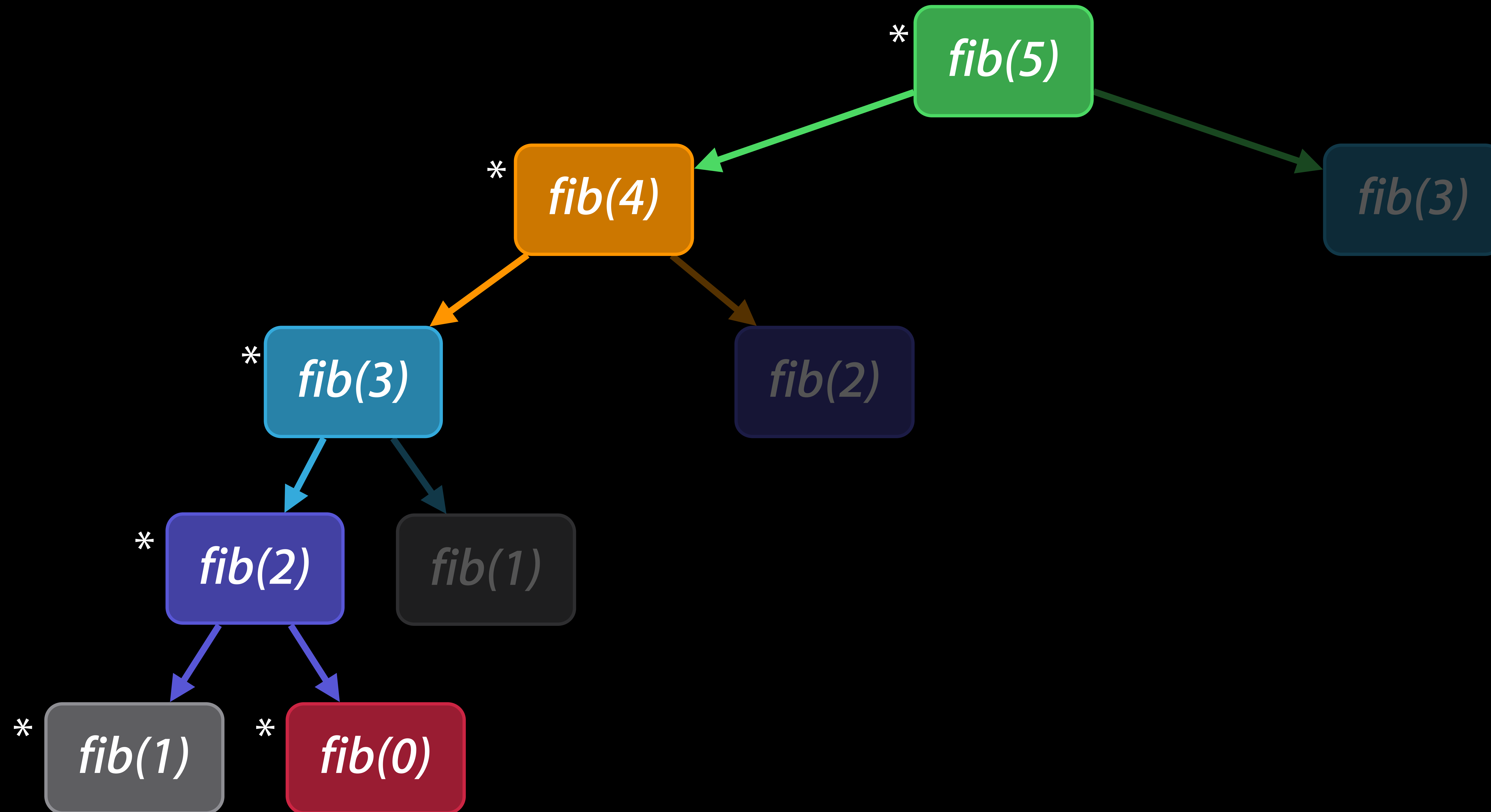
Memoized Fibonacci

Call Graph



Memoized Fibonacci

Call Graph



Manual Memoization

```
// Return the nth fibonacci number: 0, 1, 1, 2, 3, 5, 8, 13, 21, …
func fibonacci(n: Int) -> Double {
  return n < 2 ? Double(n) : fibonacci(n - 1) + fibonacci(n - 2)
}
```

Manual Memoization

```
var fibonacciMemo = Dictionary<Int, Double>()     // implementation detail

// Return the nth fibonacci number: 0, 1, 1, 2, 3, 5, 8, 13, 21, …
func fibonacci(n: Int) -> Double {
  return n < 2 ? Double(n) : fibonacci(n - 1) + fibonacci(n - 2)
}
```

Manual Memoization

```
var fibonacciMemo = Dictionary<Int, Double>()     // implementation detail

// Return the nth fibonacci number: 0, 1, 1, 2, 3, 5, 8, 13, 21, …
func fibonacci(n: Int) -> Double {
  if let result = fibonacciMemo[n] {
    return result
  }
  let result = n < 2 ? Double(n) : fibonacci(n - 1) + fibonacci(n - 2)
  fibonacciMemo[n] = result
  return result
}
```

Manual Memoization

```
var fibonacciMemo = Dictionary<Int, Double>()     // implementation detail

// Return the nth fibonacci number: 0, 1, 1, 2, 3, 5, 8, 13, 21, …
func fibonacci(n: Int) -> Double {
  if let result = fibonacciMemo[n] {
    return result
  }
  let result = n < 2 ? Double(n) : fibonacci(n - 1) + fibonacci(n - 2)
  fibonacciMemo[n] = result
  return result
}

// 1.61803399…
let phi = fibonacci(45) / fibonacci(44)
```

Manual Memoization

```
var fibonacciMemo = Dictionary<Int, Double>()     // implementation detail

// Return the nth fibonacci number: 0, 1, 1, 2, 3, 5, 8, 13, 21, …
func fibonacci(n: Int) -> Double {
  if let result = fibonacciMemo[n] {
    return result
  }
  let result = n < 2 ? Double(n) : fibonacci(n - 1) + fibonacci(n - 2)
  fibonacciMemo[n] = result
  return result
}

// 1.61803399…
let phi = fibonacci(45) / fibonacci(44)
```

0.1 seconds = 100x speedup

Manual Memoization

```
var fibonacciMemo = Dictionary<Int, Double>()     // implementation detail

// Return the nth fibonacci number: 0, 1, 1, 2, 3, 5, 8, 13, 21, …
func fibonacci(n: Int) -> Double {
  if let result = fibonacciMemo[n] {
    return result
  }
  let result = n < 2 ? Double(n) : fibonacci(n - 1) + fibonacci(n - 2)
  fibonacciMemo[n] = result
  return result
}

// 1.61803399…
let phi = fibonacci(45) / fibonacci(44)
```

0.1 seconds = 100x speedup

Automatic Memoization

```
// Return the nth fibonacci number: 0, 1, 1, 2, 3, 5, 8, 13, 21, …
let fibonacci: (Int)->Double = memoize {
  fibonacci, n in
  n < 2 ? Double(n) : fibonacci(n - 1) + fibonacci(n - 2)
}
```

Automatic Memoization

```
// Return the nth fibonacci number: 0, 1, 1, 2, 3, 5, 8, 13, 21, …
let fibonacci: (Int)->Double = memoize {
  fibonacci, n in
  n < 2 ? Double(n) : fibonacci(n - 1) + fibonacci(n - 2)
}
```

Automatic Memoization

```
// Return the nth fibonacci number: 0, 1, 1, 2, 3, 5, 8, 13, 21, …
let fibonacci: (Int)->Double = memoize {
  fibonacci, n in
  n < 2 ? Double(n) : fibonacci(n – 1) + fibonacci(n – 2)
}
```

Automatic Memoization

```
// Return the nth fibonacci number: 0, 1, 1, 2, 3, 5, 8, 13, 21, …
let fibonacci: (Int)->Double = memoize {
  fibonacci, n in
  n < 2 ? Double(n) : fibonacci(n - 1) + fibonacci(n - 2)
}
```

Automatic Memoization

```
// Return the nth fibonacci number: 0, 1, 1, 2, 3, 5, 8, 13, 21, …
let fibonacci = memoize {
  fibonacci, n in
  n < 2 ? Double(n) : fibonacci(n - 1) + fibonacci(n - 2)
}
```

Automatic Memoization

```
// Return the nth fibonacci number: 0, 1, 1, 2, 3, 5, 8, 13, 21, …
let fibonacci = memoize {
  fibonacci, n in
  n < 2 ? Double(n) : fibonacci(n – 1) + fibonacci(n – 2)
}

// Parse a text representation of a property list, returning an NSString,
// NSData, NSArray, or NSDictionary object, according to the topmost element
let parsePropertyList = memoize {
  (_, s: String) in
  s.propertyList()
}
```

Automatic Memoization, Take One

```
func memoize<T: Hashable, U>( body: (T)->U ) -> (T)->U {
  var memo = Dictionary<T, U>()
  return { x in
    if let q = memo[x] { return q }
    let r = body(x)
    memo[x] = r
    return r
  }
}
```

Automatic Memoization, Take One

```
func memoize<T: Hashable, U>( body: (T)->U ) -> (T)->U {
  var memo = Dictionary<T, U>()
  return { x in
    if let q = memo[x] { return q }
    let r = body(x)
    memo[x] = r
    return r
  }
}
```

Automatic Memoization, Take One

```
func memoize<T: Hashable, U>( body: (T)->U ) -> (T)->U {
  var memo = Dictionary<T, U>()
  return { x in
    if let q = memo[x] { return q }
    let r = body(x)
    memo[x] = r
    return r
  }
}
```

Automatic Memoization, Take One

```
func memoize<T: Hashable, U>( body: (T)->U ) -> (T)->U {
  var memo = Dictionary<T, U>()
  return { x in
    if let q = memo[x] { return q }
    let r = body(x)
    memo[x] = r
    return r
  }
}
```

Automatic Memoization, Take One

```
func memoize<T: Hashable, U>( body: (T)->U ) -> (T)->U {
  var memo = Dictionary<T, U>()
  return { x in
    if let q = memo[x] { return q }
    let r = body(x)
    memo[x] = r
    return r
  }
}
```

Automatic Memoization, Take One

```
func memoize<T: Hashable, U>( body: (T)->U ) -> (T)->U {
  var memo = Dictionary<T, U>()
  return { x in
    if let q = memo[x] { return q }
    let r = body(x)
    memo[x] = r
    return r
  }
}
```

Automatic Memoization, Take One

```
func memoize<T: Hashable, U>( body: (T)->U ) -> (T)->U {
  var memo = Dictionary<T, U>()
  return { x in
    if let q = memo[x] { return q }
    let r = body(x)
    memo[x] = r
    return r
  }
}
```

Automatic Memoization, Take One

```
func memoize<T: Hashable, U>( body: (T)->U ) -> (T)->U {
  var memo = Dictionary<T, U>()
  return { x in
    if let q = memo[x] { return q }
    let r = body(x)
    memo[x] = r
    return r
  }
}
```

Automatic Memoization, Take One

```
func memoize<T: Hashable, U>( body: (T)->U ) -> (T)->U {
  var memo = Dictionary<T, U>()
  return { x in
    if let q = memo[x] { return q }
    let r = body(x)
    memo[x] = r
    return r
  }
}

let parsePropertyList = memoize { (s: String) in s.propertyList() }
```

Automatic Memoization, Take One

```
func memoize<T: Hashable, U>( body: (T)->U ) -> (T)->U {
  var memo = Dictionary<T, U>()
  return { x in
    if let q = memo[x] { return q }
    let r = body(x)
    memo[x] = r
    return r
  }
}

let factorial = memoize { x in x == 0 ? 1 : x * factorial(x - 1) }
```

Automatic Memoization, Take One

```
func memoize<T: Hashable, U>( body: (T)->U ) -> (T)->U {
  var memo = Dictionary<T, U>()
  return { x in
    if let q = memo[x] { return q }
    let r = body(x)
    memo[x] = r
    return r
  }
}

let factorial = memoize { x in x == 0 ? 1 : x * factorial(x - 1) }
```

error: variable used within its own initial value

Automatic Memoization, Take One

```
func memoize<T: Hashable, U>( body: (T)->U ) -> (T)->U {
  var memo = Dictionary<T, U>()
  return { x in
    if let q = memo[x] { return q }
    let r = body(x)
    memo[x] = r
    return r
  }
}

let factorial = memoize { x in x == 0 ? 1 : x * factorial(x - 1) }
```

Automatic Memoization, Take One

```
func memoize<T: Hashable, U>( body: (T)->U ) -> (T)->U {
  var memo = Dictionary<T, U>()
  return { x in
    if let q = memo[x] { return q }
    let r = body(x)
    memo[x] = r
    return r
  }
}

var factorial: (Int)->Int = { $0 }
factorial = memoize { x in x == 0 ? 1 : x * factorial(x - 1) }
```

Automatic Memoization, Take Two

```
func memoize<T: Hashable, U>( body: (T)->U ) -> (T)->U {
  var memo = Dictionary<T, U>()
  return { x in
    if let q = memo[x] { return q }
    let r = body(x)
    memo[x] = r
    return r
  }
}

let factorial = memoize { x in x == 0 ? 1 : x * factorial(x - 1) }
```

error: variable used within its own initial value

Automatic Memoization, Take Two

```
func memoize<T: Hashable, U>( body: (T)->U ) -> (T)->U {
  var memo = Dictionary<T, U>()
  return { x in
    if let q = memo[x] { return q }
    let r = body(x)
    memo[x] = r
    return r
  }
}

let factorial = memoize { x in x == 0 ? 1 : x * factorial(x - 1) }
```

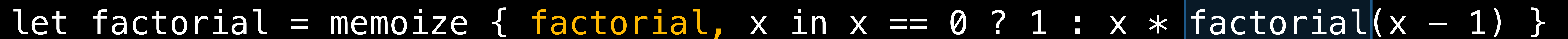
Automatic Memoization, Take Two

```
func memoize<T: Hashable, U>( body: (T)->U ) -> (T)->U {
  var memo = Dictionary<T, U>()
  return { x in
    if let q = memo[x] { return q }
    let r = body(x)
    memo[x] = r
    return r
  }
}

let factorial = memoize { factorial, x in x == 0 ? 1 : x * factorial(x - 1) }
```

Automatic Memoization, Take Two

```
func memoize<T: Hashable, U>( body: (T)->U ) -> (T)->U {
  var memo = Dictionary<T, U>()
  return { x in
    if let q = memo[x] { return q }
    let r = body(x)
    memo[x] = r
    return r
  }
}
```



Automatic Memoization, Take Two

```
func memoize<T: Hashable, U>( body: (T)->U ) -> (T)->U {
  var memo = Dictionary<T, U>()
  return { x in
    if let q = memo[x] { return q }
    let r = body(x)
    memo[x] = r
    return r
  }
}

let factorial = memoize { factorial, x in x == 0 ? 1 : x * factorial(x - 1) }
```

Automatic Memoization, Take Two

```
func memoize<T: Hashable, U>( body: ((T)->U, T)->U ) -> (T)->U {
  var memo = Dictionary<T, U>()
  return { x in
    if let q = memo[x] { return q }
    let r = body(x)
    memo[x] = r
    return r
  }
}

let factorial = memoize { factorial, x in x == 0 ? 1 : x * factorial(x - 1) }
```

Automatic Memoization, Take Two

```
func memoize<T: Hashable, U>( body: ((T)->U, T)->U ) -> (T)->U {
  var memo = Dictionary<T, U>()
  return { x in
    if let q = memo[x] { return q }
    let r = body(x)
    memo[x] = r
    return r
  }
}

let factorial = memoize { factorial, x in x == 0 ? 1 : x * factorial(x - 1) }
```

Automatic Memoization, Take Two

```
func memoize<T: Hashable, U>( body: ((T)->U, T)->U ) -> (T)->U {
  var memo = Dictionary<T, U>()
  var result: ((T)->U)!
  result = { x in
    if let q = memo[x] { return q }
    let r = body(x)
    memo[x] = r
    return r
  }
  return result
}

let factorial = memoize { factorial, x in x == 0 ? 1 : x * factorial(x - 1) }
```

Automatic Memoization, Take Two

```
func memoize<T: Hashable, U>( body: ((T)->U, T)->U ) -> (T)->U {
  var memo = Dictionary<T, U>()
  var result: ((T)->U)!
  result = { x in
    if let q = memo[x] { return q }
    let r = body(x)
    memo[x] = r
    return r
  }
  return result
}

let factorial = memoize { factorial, x in x == 0 ? 1 : x * factorial(x - 1) }
```

Automatic Memoization, Take Two

```
func memoize<T: Hashable, U>( body: ((T)->U, T)->U ) -> (T)->U {
  var memo = Dictionary<T, U>()
  var result: ((T)->U)!
  result = { x in
    if let q = memo[x] { return q }
    let r = body(result, x)
    memo[x] = r
    return r
  }
  return result
}

let factorial = memoize { factorial, x in x == 0 ? 1 : x * factorial(x - 1) }
```

Automatic Memoization, Take Two

```
func memoize<T: Hashable, U>( body: ((T)->U, T)->U ) -> (T)->U {
  var memo = Dictionary<T, U>()
  var result: ((T)->U)!
  result = { x in
    if let q = memo[x] { return q }
    let r = body(result, x)
    memo[x] = r
    return r
  }
  return result
}

let factorial = memoize { factorial, x in x == 0 ? 1 : x * factorial(x - 1) }
```

Synergy of Powerful Features

Synergy of Powerful Features

Type deduction for concision:

```
let fibonacci = memoize { fibonacci, n in
```

Synergy of Powerful Features

Type deduction for concision

```
let fibonacci = memoize { fibonacci, n in
```

Trailing closure syntax for expressivity

```
let fibonacci = memoize { fibonacci, n in
```

Synergy of Powerful Features

Type deduction for concision:

```
let fibonacci = memoize { fibonacci, n in
```

Trailing closure syntax for expressivity

```
let fibonacci = memoize { fibonacci, n in
```

Generic functions for generality with safety *and* performance

```
func memoize<T: Hashable, U>( body: ((T)->U, T)->U ) -> (T)->U
```

Generic Types

Swift has ‘em

Generic Types are Everywhere

Array<T> and Dictionary<K,V>: generic structs

Optional<T>: a generic enum

Generic classes are possible, too

A Simple Stack of Strings

```
struct StringStack {
  mutating func push(x: String) {
    items += x
  }
  mutating func pop() -> String {
    return items.removeLast()
  }
  var items: String[]
}
```

A Simple Stack of Strings

```
struct StringStack {
  mutating func push(x: String) {
    items += x
  }
  mutating func pop() -> String {
    return items.removeLast()
  }
  var items: String[]
}
```

A Simple Generic Stack

```
struct Stack<T> {
  mutating func push(x: T) {
    items += x
  }
  mutating func pop() -> T {
    return items.removeLast()
  }
  var items: T[]
}
```

A Simple Generic Stack

```
struct Stack<T> {
  mutating func push(x: T) {
    items += x
  }
  mutating func pop() -> T {
    return items.removeLast()
  }
  var items: T[]
}

var intStack = Stack<Int>()
intStack.push(42)
```

A Simple Generic Stack

```
struct Stack<T> {
  mutating func push(x: T) {
    items += x
  }
  mutating func pop() -> T {
    return items.removeLast()
  }
  var items: T[]
}

var intStack = Stack<Int>()
intStack.push(42)
var windowStack = Stack<NSWindow>()
```

A Simple Generic Stack

```
struct Stack<T> {
  mutating func push(x: T) {
    items += x
  }
  mutating func pop() -> T {
    return items.removeLast()
  }
  var items: T[]
}

var intStack = Stack<Int>()
intStack.push(42)
var windowStack = Stack<NSWindow>()
```

A Simple Generic Stack

```
struct Stack<T> {
  mutating func push(x: T) {
    items += x
  }
  mutating func pop() -> T {
    return items.removeLast()
  }
  var items: T[]
}

var intStack = Stack<Int>()
intStack.push(42)
var windowStack = Stack<NSWindow>()
```

```
func peekStack(s: Stack<T>) {
  for x in s { println(x) }
}
```

A Simple Generic Stack

```
struct Stack<T> {
  mutating func push(x: T) {
    items += x
  }
  mutating func pop() -> T {
    return items.removeLast()
  }
  var items: T[]
}

var intStack = Stack<Int>()
intStack.push(42)
var windowStack = Stack<NSWindow>()
```

error: 'Stack<T>' does not conform to protocol 'Sequence'

```
func peekStack(s: Stack<T>) {
  for x in s { println(x) }
}
```

Under the Hood

Inside the Swift `for...in` Loop

You write

Under the Hood

Inside the Swift `for…in` Loop

You write

```
for x in someSequence {
  ...
}
```

Under the Hood

Inside the Swift `for…in` Loop

You write

```
for x in someSequence {
  ...
}
```

Internally, Swift *re*writes

Under the Hood

Inside the Swift `for…in` Loop

You write

```
for x in someSequence {
  ...
}
```

Internally, Swift *re*writes

```
var __g = someSequence.generate()
while let x = __g.next() {
  ...
}
```

Under the Hood

Inside the Swift for…in Loop

You write

```
for x in someSequence {
  ...
}
```

Internally, Swift *re*writes

```
var __g = someSequence.generate()
while let x = __g.next() {
  ...
}
```

Under the Hood

Inside the Swift for...in Loop

You write

```
for x in someSequence {
  ...
}
```

Internally, Swift *re*write

Create a Generator

```
var __g = someSequence.generate()
while let x = __g.next() {
  ...
}
```

Under the Hood

Inside the Swift for…in Loop

You write

```
for x in someSequence {
  ...
}
```

Internally, Swift *re*writes

```
var __g = someSequence.generate()
while let x = __g.next() {
  ...
}
```

Under the Hood

Inside the Swift for...in Loop

You write:

```
for x in someSequence {
  ...
}
```

Internally, Swift *re*writes

```
var __g = someSequence.generate()
while let x = __g.next() {
  ...
}
```

Under the Hood

Inside the Swift for…in Loop

You write:

```
for x in someSequence {
  ...
}
```

Internally, Swift *rewrites*

```
var __g = someSequence.generate()
while let x = __g.next() {
  ...
}
```

Return the next element wrapped in an optional (or nil if done)

Under the Hood

Inside the Swift for...in Loop

```
protocol Generator {
  typealias Element
  mutating func next() -> Element?
}
```

```
var __g = someSequence.generate()
while let x = __g.next() {
  ...
}
```

Under the Hood

Inside the Swift for...in Loop

```
protocol Generator {
  typealias Element
  mutating func next() -> Element?
}
```

```
var __g = someSequence.generate()
while let x = __g.next() {
  ...
}
```

Under the Hood

Inside the Swift for...in Loop

```
protocol Generator {
  typealias Element
  mutating func next() -> Element?
}
```

An "associated type" requirement

```
var __g = someSequence.generate()
while let x = __g.next() {
  ...
}
```

Under the Hood

Inside the Swift for...in Loop

```
protocol Generator {
  typealias Element
  mutating func next() -> Element?
}
```

An "associated type" requirement

```
var __g = someSequence.generate()
while let x = __g.next() {
  ...
}
```

A Generator for Stack<T>

```
protocol Generator {
  typealias Element
  mutating func next() -> Element?
}
```

A Generator for Stack<T>

```
protocol Generator {
  typealias Element
  mutating func next() -> Element?
}
```

```
struct StackGenerator<T> : Generator {
  typealias Element = T
  mutating func next() -> T? {
    if items.isEmpty { return nil }
    let ret = items[0]
    items = items[1..items.count]
    return ret
  }

  var items: Slice<T>
}
```

A Generator for Stack<T>

```
protocol Generator {
  typealias Element
  mutating func next() -> Element?
}
```

```
struct StackGenerator<T> : Generator {
  typealias Element = T
  mutating func next() -> T? {
    if items.isEmpty { return nil }
    let ret = items[0]
    items = items[1..items.count]
    return ret
  }

  var items: Slice<T>
}
```

A Generator for Stack<T>

```
protocol Generator {
  typealias Element
  mutating func next() -> Element?
}
```

```
struct StackGenerator<T> : Generator {
  typealias Element = T
  mutating func next() -> T? {
    if items.isEmpty { return nil }
    let ret = items[0]
    items = items[1..items.count]
    return ret
  }

  var items: Slice<T>
}
```

A Generator for Stack<T>

```
protocol Generator {
  typealias Element
  mutating func next() -> Element?
}
```

```
struct StackGenerator<T> : Generator {

  mutating func next() -> T? {
    if items.isEmpty { return nil }
    let ret = items[0]
    items = items[1..items.count]
    return ret
  }

  var items: Slice<T>
}
```

A Generator for Stack<T>

```
protocol Generator {
  typealias Element
  mutating func next() -> Element?
}
```

```
struct StackGenerator<T> : Generator {

  mutating func next() -> T? {
    if items.isEmpty { return nil }
    let ret = items[0]
    items = items[1..items.count]
    return ret
  }

  var items: Slice<T>
}
```

A Generator for Stack<T>

```
protocol Generator {
  typealias Element
  mutating func next() -> Element?
}
```

```
struct StackGenerator<T> : Generator {

  mutating func next() -> T? {
    if items.isEmpty { return nil }
    let ret = items[0]
    items = items[1..items.count]
    return ret
  }

  var items: Slice<T>
}
```

A Generator for Stack<T>

```
protocol Generator {
  typealias Element
  mutating func next() -> Element?
}
```

```
struct StackGenerator<T> : Generator {

  mutating func next() -> T? {
    if items.isEmpty { return nil }
    let ret = items[0]
    items = items[1..items.count]
    return ret
  }

  var items: Slice<T>
}
```

A Generator for Stack<T>

```
protocol Generator {
  typealias Element
  mutating func next() -> Element?
}
```

```
struct StackGenerator<T> : Generator {

  mutating func next() -> T? {
    if items.isEmpty { return nil }
    let ret = items[0]
    items = items[1..items.count]
    return ret
  }

  var items: Slice<T>
}
```

A Generator for Stack<T>

```
protocol Generator {
  typealias Element
  mutating func next() -> Element?
}
```

```
struct StackGenerator<T> : Generator {

  mutating func next() -> T? {
    if items.isEmpty { return nil }
    let ret = items[0]
    items = items[1..items.count]
    return ret
  }

  var items: Slice<T>
}
```

A Generator for Stack<T>

```
protocol Generator {
  typealias Element
  mutating func next() -> Element?
}
```

```
struct StackGenerator<T> : Generator {

  mutating func next() -> T? {
    if items.isEmpty { return nil }
    let ret = items[0]
    items = items[1..items.count]
    return ret
  }

  var items: Slice<T>
}
```

A Generator for Stack<T>

```
protocol Generator {
  typealias Element
  mutating func next() -> Element?
}
```

```
struct StackGenerator<T> : Generator {

  mutating func next() -> T? {
    if items.isEmpty { return nil }
    let ret = items[0]
    items = items[1..items.count]
    return ret
  }

  var items: Slice<T>
}
```

A Generator for Stack<T>

```
protocol Generator {
  typealias Element
  mutating func next() -> Element?
}
```

```
struct StackGenerator<T> : Generator {

  mutating func next() -> T? {
    if items.isEmpty { return nil }
    let ret = items[0]
    items = items[1..items.count]
    return ret
  }

  var items: Slice<T>
}
```

A Generator for Stack<T>

```
protocol Generator {
  typealias Element
  mutating func next() -> Element?
}
```

```
struct StackGenerator<T> : Generator {

  mutating func next() -> T? {
    if items.isEmpty { return nil }
    let ret = items[0]
    items = items[1..items.count]
    return ret
  }

  var items: Slice<T>
}
```

The Sequence Protocol

```
protocol Generator {
  typealias Element
  mutating func next() -> Element?
}

protocol Sequence {
  typealias GeneratorType : Generator
  func generate() -> GeneratorType
}
```

The Sequence Protocol

```
protocol Generator {
  typealias Element
  mutating func next() -> Element?
}

protocol Sequence {
  typealias GeneratorType : Generator
  func generate() -> GeneratorType
}
```

The Sequence Protocol

```
protocol Generator {
  typealias Element
  mutating func next() -> Element?
}

protocol Sequence {
  typealias GeneratorType : Generator
  func generate() -> GeneratorType
}
```

Stack<T>: Implementing Sequence

```
protocol Generator {
  typealias Element
  mutating func next() -> Element?
}

protocol Sequence {
  typealias GeneratorType : Generator
  func generate() -> GeneratorType
}
```

Stack<T>: Implementing Sequence

```
protocol Sequence {
  typealias GeneratorType : Generator
  func generate() -> GeneratorType
}
```

Stack<T>: Implementing Sequence

```
protocol Sequence {
  typealias GeneratorType : Generator
  func generate() -> GeneratorType
}
```

```
extension Stack : Sequence {
  func generate() -> StackGenerator<T> {
    return StackGenerator( items[0..itemCount] )
  }
}
```

Stack<T>: Implementing Sequence

```
protocol Sequence {
  typealias GeneratorType : Generator
  func generate() -> GeneratorType
}
```

```
extension Stack : Sequence {
  func generate() -> StackGenerator<T> {
    return StackGenerator( items[0..itemCount] )
  }
}
```

Stack<T>: Implementing Sequence

```
protocol Sequence {
  typealias GeneratorType : Generator
  func generate() -> GeneratorType
}
```

```
extension Stack : Sequence {
  func generate() -> StackGenerator<T> {
    return StackGenerator( items[0..itemCount] )
  }
}
```

Stack<T>: Implementing Sequence

```
protocol Sequence {
  typealias GeneratorType : Generator
  func generate() -> GeneratorType
}
```

```
extension Stack : Sequence {
  func generate() -> StackGenerator<T> {
    return StackGenerator( items[0..itemCount] )
  }
}
```

Stack<T>: Implementing Sequence

```
protocol Sequence {
  typealias GeneratorType : Generator
  func generate() -> GeneratorType
}
```

```
extension Stack : Sequence {
  func generate() -> StackGenerator<T> {
    return StackGenerator( items[0..itemCount] )
  }
}
```

Stack<T>: Implementing Sequence

```
protocol Sequence {
  typealias GeneratorType : Generator
  func generate() -> GeneratorType
}
```

```
extension Stack : Sequence {
  func generate() -> StackGenerator<T> {
    return StackGenerator( items[0..itemCount] )
  }
}

func peekStack(s: Stack<T>) {
  for x in s { println(x) }
}
```

Works!

Summary of Swift Generics and Protocols

Protocols are your hooks into the Swift core language

Swift generics combine abstraction, safety, and performance in new ways

Read, experiment, and have fun. There's plenty to discover!

The Swift Model

John McCall
Syntax Artist

The Minimal Model

- Statically compiled
- Small runtime

Simple Interoperation

Transparent interaction with C and Objective C

Can deploy to previous versions of iOS and Mac OS X

Predictable Behavior

You control the code that runs

Comprehensible compilation with inspectable results

No non-deterministic JITs or garbage collection pauses

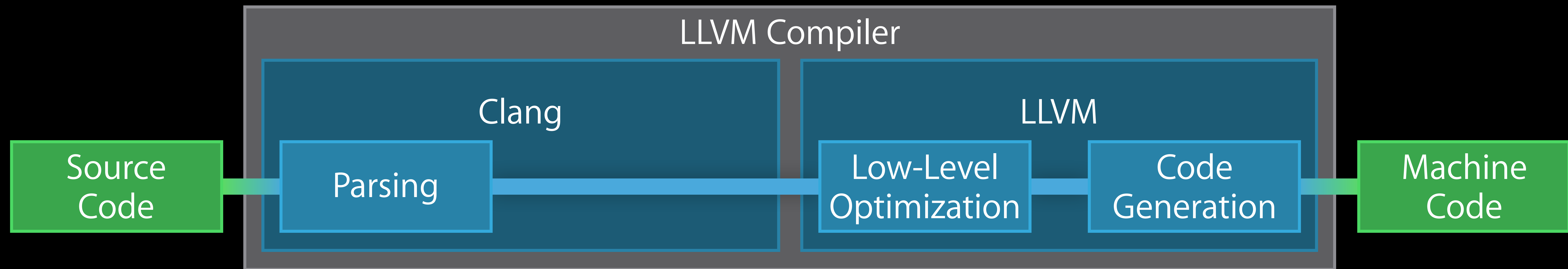
Efficient Execution

Native code instantly ready to run

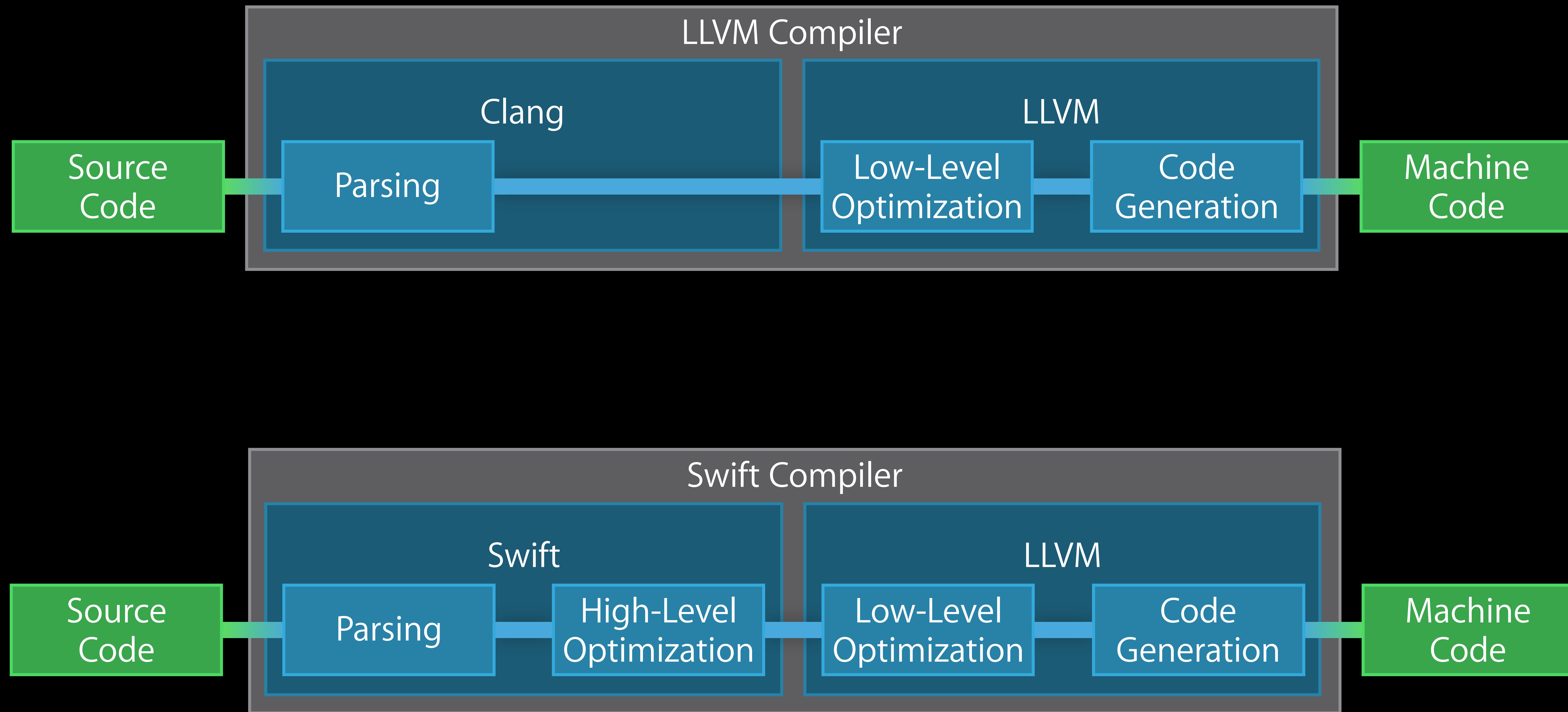
No artificial abstraction barriers

Predictable model enables bare-to-the-metal programming

Compiler Architecture



Compiler Architecture



High-Level Optimization

Removing abstraction penalties

Generic specialization

Devirtualization

Reducing Abstraction Penalties

```
struct Newtons { var value: Float }

func + (a: Newtons, b: Newtons)
  -> Newtons {
  return Newtons(
    value: a.value + b.value)
}
```

```
struct Pounds { var value: Float }

func + (a: Pounds, b: Pounds)
  -> Pounds {
  return Pounds(
    value: a.value + b.value)
}
```

Reducing Abstraction Penalties

Global analysis of your app

Using a struct has no runtime penalty

Even Int and Float are structs

Generic Specialization

```
func swap<T>(inout x: T, inout y: T) {
  let tmp: T = x
  x = y
  y = tmp
}
```

Generic Specialization

```
func swap(inout x: Thing, inout y: Thing) {
  let tmp: Thing
  x = y
  y = tmp
}
```

Generic Specialization

```
func swap(inout x: Thing, inout y: Thing) {
  let tmp: Thing
  x = y
  y = tmp
}
```

Generic Specialization

```
func swap<T>(inout x: T, inout y: T) {
  let tmp: T = x
  x = y
  y = tmp
}
```

```
func swap(inout x: Thing, inout y: Thing) {
  let tmp: Thing
  x = y
  y = tmp
}
```

Generic Specialization

Swift can run generic code directly

Optimizer can produce specialized versions of generic code at will

- Separate compilation of generics
- Faster compiles
- Flexibility to trade code size for speed

Devirtualization

Resolving dynamic method calls at compile-time

- If Swift can see where you constructed the object
- If Swift knows that a class doesn't have any subclasses
- If you've marked a method with the @final attribute

High-Level Optimization

ARC optimization

Enum analysis

Alias analysis

Value propagation

Library optimizations on strings, arrays, etc.

Safety First

Subtle and unexpected behavior is usually also a security problem

Swift provides tools to control that:

- Statically, to encourage you to handle unexpected cases
- Dynamically, to prevent errors from propagating dangerously

Arithmetic Errors

```
func finishTransaction(cart: Cart) {
  var total: Int = 0
  for (item, quantity) in cart {
    total += item.price * quantity
  }
  collectPayment(total)
}
```

Arithmetic Errors

Title	Price (in cents)	Quantity	Item Total (in cents)
Gulliver's Travels	3999	300	1199700
A Modest Proposal	2499	300	749700
A Tale of a Tub	3499	300	1049700
The Bickerstaff-Partridge Papers	1999	300	599700
The Journal to Stella	3999	300	1199700
		Total	**4798500**

Arithmetic Errors

Title	Price (in cents)	Quantity	Item Total (in cents)
Gulliver's Travels	3999	270000	1079730000
A Modest Proposal	2499	270000	674730000
A Tale of a Tub	3499	270000	944730000
The Bickerstaff-Partridge Papers	1999	260000	519740000
The Journal to Stella	3999	270000	1079730000
		Total	**4,298,660,000**

Arithmetic Errors

Title	Price (in cents)	Quantity	Item Total (in cents)
Gulliver's Travels	3999	270000	1079730000
A Modest Proposal	2499	270000	674730000
A Tale of a Tub	3499	270000	944730000
The Bickerstaff-Partridge Papers	1999	260000	519740000
The Journal to Stella	3999	270000	1079730000
		Total	**$36,927.04**

Arithmetic Errors

Standard integer operators (+, -, *, /) fail on overflow or invalid input

Masking operators (&+, &-, &*) safely wrap around

Build Kinds

Build Kinds

```
-Onone          // optimization off, safety checks on
-O              // optimization on,  safety checks on
-Ofast       // optimization on,  safety checks off
```

Conclusion

In Summary

Take control of the basic language

Use generic programming to write cleaner code

The Swift compiler will make it fast and safe

More Information

Dave DeLong
Developer Tools Evangelist
delong@apple.com

Jake Behrens
App Frameworks Evangelist
behrens@apple.com

Swift Language Documentation
http://developer.apple.com/swift

Apple Developer Forums
http://devforums.apple.com

Related Sessions

Session	Location	Time
Introduction to Swift	Presidio	Tuesday 2:00PM
Integrating Swift with Objective-C	Presidio	Wednesday 9:00AM
Swift Playgrounds	Presidio	Wednesday 11:30AM
Intermediate Swift	Presidio	Wednesday 2:00PM
Swift Interoperability in Depth	Presidio	Wednesday 3:15PM
Introduction to LLDB and the Swift REPL	Mission	Thursday 10:15AM
Advanced Swift Debugging in LLDB	Mission	Friday 9:00AM

Labs

Swift Lab	Tools Lab A	Daily 9:00AM
Swift Lab	Tools Lab A	Daily 2:00PM

WWDC14