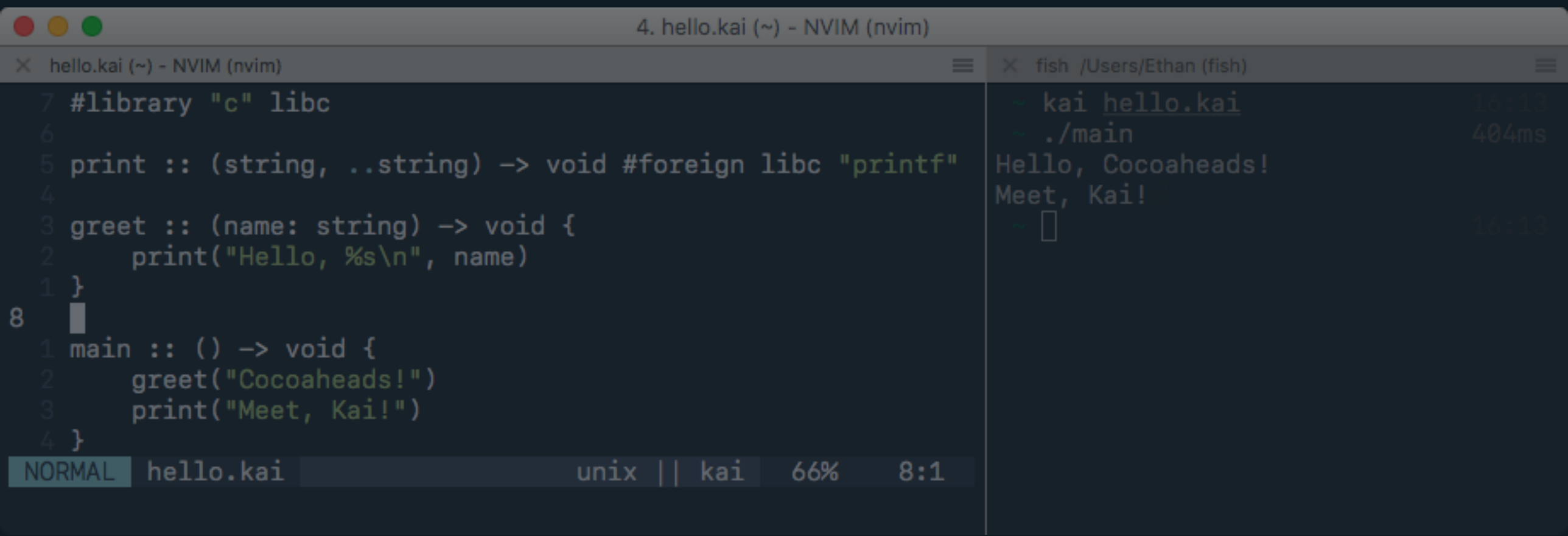
Kai


4. hello.kai (~) - NVIM (nvim)
hello.kai (~) - NVIM (nvim)
fish /Users/Ethan (fish)
#library "c" libc
print :: (string, ..string) -> void #foreign libc "printf"
greet :: (name: string) -> void {
print("Hello, %s\n", name)
}
main :: () -> void {
greet("Cocoaheads!")
print("Meet, Kai!")
}
NORMAL hello.kai unix || kai 66% 8:1
~ kai hello.kai
~ ./main 404ms
Hello, Cocoaheads!
Meet, Kai!
~

Brett R. Toomey

Vapor and iOS Developer @Nodes

@BrettRToomey

Compilers are hard

But it's not what you'd expect...

Lexing

Turns input text:

```
main :: () -> void {}
```

Into a stream of `Lexer.Token`:

```
["main", ":", ":", "(", ")", "->", "void", "{", "}"]
```

viewed as UnicodeScalars

stored as enums sometimes with associated values

Parsing

Turns a stream of `Lexer.Token`:

```
["main", ":", ":", "(", ")", "->", "void", "{", "}"]
```

Into an Abstract Syntax Tree:

```
(file '/Users/Brett/main.kai'
  (declCt names: 'main'
    (litProc type: '() -> void'
      (parameters)
      (results void)
      (stmtBlock))))
```

There are lots of these AST Nodes

5. AST.swift (~/Source/vdka/Kai/Sources/kai) - NVIM (nvim)

```
25 indirect enum AstNode {
24
23     case invalid(SourceRange)
22
21     case ident(String, SourceRange)
20     case directive(String, args: [AstNode], SourceRange)
19
18     case list([AstNode], SourceRange)
17
16     case ellipsis(AstNode, SourceRange)
15
14     case litInteger(Int64, SourceRange)
13     case litFloat(Double, SourceRange)
12     case litString(String, SourceRange)
11
10     /// - Parameter type: `typeProc` node
 9     /// - Note: `type` holds reference to the args
 8     case litProc(type: AstNode, body: AstNode, SourceRange)
 7
 6     /// - Note: Used to represent array literals (struct lits in the future)
 5     case litCompound(elements: [AstNode], SourceRange)
 4
 3     case declValue(isRuntime: Bool, names: [AstNode], type: AstNode?, values: [AstNode], SourceRange)
 2     case declImport(path: AstNode, fullpath: String?, importName: AstNode?, SourceRange)
 1     case declLibrary(path: AstNode, fullpath: String?, libName: AstNode?, SourceRange)
56
 1     case exprSubscript(receiver: AstNode, value: AstNode, SourceRange)
 2     case exprCall(receiver: AstNode, args: [AstNode], SourceRange)
 3     case exprParen(AstNode, SourceRange)
 4     case exprUnary(String, expr: AstNode, SourceRange)
 5     case exprBinary(String, lhs: AstNode, rhs: AstNode, SourceRange)
 6     case exprTernary(cond: AstNode, AstNode, AstNode, SourceRange)
 7     case exprSelector(receiver: AstNode, member: AstNode, SourceRange)
 8
 9
10
11     /// Essentially an expr which has it's rvalue thrown away
12     case stmtExpr(AstNode)
13     case stmtEmpty(SourceRange)
14     case stmtAssign(String, lhs: [AstNode], rhs: [AstNode], SourceRange)
15     case stmtBlock([AstNode], SourceRange)
16     case stmtIf(cond: AstNode, body: AstNode, AstNode?, SourceRange)
17     case stmtReturn([AstNode], SourceRange)
18     case stmtFor(initializer: AstNode?, cond: AstNode?, post: AstNode?, body: AstNode, SourceRange)
19     case stmtDefer(AstNode, SourceRange)
20     case stmtBreak(SourceRange)
21     case stmtContinue(SourceRange)
22
23     /// - Parameter params:
24     case typeProc(params: [AstNode], results: [AstNode], SourceRange)
25     case typePointer(type: AstNode, SourceRange)
26     case typeNullablePointer(type: AstNode, SourceRange)
27     case typeArray(count: AstNode, type: AstNode, SourceRange)
28 }
NORMAL  AST.swift                                   unix || swift   5%   56:1
```

They all have locations

5. AST.swift (~/Source/vdka/Kai/Sources/kai) - NVIM (nvim)

```
  1 extension AstNode {
109 
  1     var startLocation: SourceLocation {
  2         return location.lowerBound
  3     }
  4 
  5     var endLocation: SourceLocation {
  6         return location.upperBound
  7     }
  8 
  9     var location: SourceRange {
 10 
 11         switch self {
 12         case .invalid(let location),
 13              .ident(_, let location),
 14              .directive(_, _, let location),
 15              .list(_, let location),
 16              .ellipsis(_, let location),
 17              .litInteger(_, let location),
 18              .litFloat(_, let location),
 19              .litString(_, let location),
 20              .litProc(_, _, let location),
 21              .litCompound(_, let location),
 22              .declValue(_, _, _, _, let location),
 23              .declImport(_, _, _, let location),
 24              .declLibrary(_, _, _, let location),
 25              .exprUnary(_, _, let location),
 26              .exprBinary(_, _, _, let location),
 27              .exprParen(_, let location),
 28              .exprSelector(_, _, let location),
 29              .exprCall(_, _, let location),
 30              .exprSubscript(_, _, let location),
 31              .exprTernary(_, _, _, let location),
 32              .stmtEmpty(let location),
 33              .stmtAssign(_, _, _, let location),
 34              .stmtBlock(_, let location),
 35              .stmtIf(_, _, _, let location),
 36              .stmtReturn(_, let location),
 37              .stmtFor(_, _, _, _, let location),
 38              .stmtDefer(_, let location),
 39              .stmtBreak(let location),
 40              .stmtContinue(let location),
 41              .typeProc(_, _, let location),
 42              .typePointer(_, let location),
 43              .typeNullablePointer(_, let location),
 44              .typeArray(_, _, let location):
 45 
 46              return location
 47 
 48         case .stmtExpr(let expr):
 49             return expr.location
 50         }
 51     }
 52 }
NORMAL AST.swift                                                unix || swift   10%  109:1
```

They all have a description

5. AST.swift + (~/Source/vdka/Kai/Sources/kai) - NVIM (nvim)

```
extension AstNode: CustomStringConvertible {
397
    var description: String {

        switch self {
        case .invalid(let location):
            return "<invalid at \(location)>"

        case .ident(let ident, _):
            return ident

        case .directive(let directive, let args, _):
            return "\(directive) \(args.commaSeparated)"

        case .list(let nodes, _):
            return nodes.description

        case .ellipsis(let expr, _):
            return "..\(expr)"

        case .litInteger(let i, _):
            return i.description

        case .litFloat(let d, _):
            return d.description

        case .litString(let s, _):
            return "\"\(s)\""

        case .litProc(let type, let body, _):
            return "\(type) \(body)"

        case .litCompound(let elements, _):
            return elements.map({ $0.description }).joined(separator: ", ")

        case .declValue(let isRuntime, let names, let type, let values, _):
            let declChar = isRuntime ? "=" : ":"

            if values.isEmpty {
                return "\(names.commaSeparated) : \(type!)"
            } else if let type = type {
                return "\(names.commaSeparated) : \(type) \(declChar) \(values.commaSeparated)"
            }
            return "\(names.commaSeparated) :\(declChar) \(values.commaSeparated)"

        case .declImport(let path, _, let importName, _):
            if let importName = importName {
                return "#import \(path) \(importName)"
            }
            return "#import \(path)"

        case .declLibrary(let path, _, let libName, _):
            if let libName = libName {
                return "#library \(path) \(libName)"
            }
            return "#library \(path)"

        case .exprCall(let receiver, let args, _):
            return "\(receiver)(\(args.commaSeparated))"

        case .exprSubscript(let receiver, let value, _):
            return "\(receiver)[\(value)]"

        case .exprParen(let expr, _):
            return "(\(expr))"

        case .exprUnary(let op, let expr, _):
            return "\(op)\(expr)"

        case .exprBinary(let op, let lhs, let rhs, _):
            return "\(lhs) \(op) \(rhs)"

        case .exprTernary(let cond, let then, let el, _):
            return "\(cond) ? \(then) : \(el)"

        case .exprSelector(let receiver, let member, _):
            return "\(receiver).\(member)"

        case .stmtExpr(let expr):
            return expr.description

        case .stmtEmpty(_):
            return ";" // NOTE(vdka): Is this right?

        case .stmtAssign(let op, let lhs, let rhs, _):
            return "\(lhs.commaSeparated) \(op) \(rhs.commaSeparated)"

        case .stmtBlock:
            return "{ /* ... */ }" // NOTE(vdka): Is this good?

        case .stmtIf:
            return "if"
NORMAL  AST.swift | +                                   unix || swift   36%  397:1
```

… And a Tree print format

5. AST.swift + (~/Source/vdka/Kai/Sources/kai) - NVIM (nvim)

```
572     func pretty(depth: Int = 0, includeParens: Bool = true, specialName: String? = nil) -> String {
1
2           var unlabeled: [String] = []
3           var labeled: [(String, String)] = []
4
5           var children: [AstNode] = []
6
7           // used to emit a node with a different short name ie: list node as parameters or results
8           var renamedChildren: [(String, AstNode)] = []
9
10          switch self {
11          case .invalid(let location):
12              labeled.append(("location", location.description))
13
14          case .ident(let ident, _):
15              unlabeled.append(ident)
16
17          case .directive(let directive, _, _):
18              unlabeled.append(directive)
19
20          case .list(let nodes, _):
21
22              if nodes.reduce(true, { $0.0 && $0.1.isIdent }) {
23                  unlabeled.append(nodes.commaSeparated)
24              } else {
25                  children.append(contentsOf: nodes)
26              }
27
28          case .ellipsis(let expr, _):
29              children.append(expr)
30
31          case .litInteger(let val, _):
32              unlabeled.append("'" + val.description + "'")
33
34          case .litFloat(let val, _):
35              unlabeled.append("'" + val.description + "'")
36
37          case .litString(let val, _):
38              unlabeled.append("\"" + val + "\"")
39
40          case .litProc(let type, let body, _):
41              labeled.append(("type", type.description))
42              guard case .typeProc(let params, let results, _) = type else {
43                  break // NOTE(vdka): Do we want to break?
44              }
45
46              let emptyList = AstNode.list([], SourceLocation.unknown ..< .unknown)
47              var paramsList = emptyList
48              for param in params {
49                  for decl in explode(param) {
50                      paramsList = append(paramsList, decl)
51                  }
52              }
53
54              var resultList = emptyList
55              for result in results {
56                  for decl in explode(result) {
57                      resultList = append(resultList, decl)
58                  }
59              }
60              renamedChildren.append(("parameters", paramsList))
61              renamedChildren.append(("results", resultList))
62
63              children.append(body)
64
65          case .litCompound(let elements, _):
66              children.append(contentsOf: elements)
67
68          case .exprUnary(let op, let expr, _):
69              unlabeled.append("'" + op + "'")
70              children.append(expr)
71
72          case .exprBinary(let op, let lhs, let rhs, _):
73              unlabeled.append("'" + op + "'")
74              children.append(lhs)
75              children.append(rhs)
76
77          case .exprParen(let expr, _):
78              children.append(expr)
79
80          case .exprSelector(let receiver, let selector, _):
81              children.append(receiver)
82              children.append(selector)
83
84          case .exprCall(let receiver, let args, _):
85              unlabeled.append(receiver.description)
86              children.append(contentsOf: args)
87
88          case .exprSubscript(let receiver, let value, _):
89              unlabeled.append(receiver.description)
90              children.append(value)
91
92          case .exprTernary(let cond, let trueBranch, let falseBranch, _):
NORMAL  AST.swift | +                                                    unix || swift   52%  572:99
```

... And a type annotated version of the same Tree print

In total `AST.swift` is just over **1200 lines** of code on it's own.

Checking

Generates a set of annotations for the AST:

```
(file '/Users/Brett/main.kai'
  (declCt names: 'main' types: '() -> void'
    (litProc type: '() -> void'
      (parameters )
      (results void)
      (stmtBlock))))
```

```
extension AstNode: Hashable {
    var hashValue: Int {
        // Because Int is the platform native size, and so are pointers the result is
        //   that the hashValue *should* be the pointer address.
        // Thanks to this we have instance identity as the hashValue.
        return unsafeBitCast(self, to: Int.self)
    }
}


struct Info {
    var entities:    [Entity:  DeclInfo] = [:]
    var definitions:  [AstNode: Entity]   = [:] // Key: AstNode.ident
    var decls:       [AstNode: DeclInfo] = [:] // Key: AstNode.declValue
    var types:       [AstNode: Type]     = [:] // Key: Any AstNode that can be a type
    var uses:        [AstNode: Entity]   = [:] // Key: AstNode.ident
    var scopes:      [AstNode: Scope]    = [:] // Key: Any AstNode
    var casts:       Set<AstNode>        = [ ] // Key: AstNode.call
}
```

IRGeneration (Using LLVM)

```
; ModuleID = '/Users/Brett/main.kai'
source_filename = "/Users/Brett/main.kai"

define void @main() {
entry:
  %result = alloca i32
  %y = alloca i32
  br label %return

return:                                           ; preds = %entry
  %result1 = load i32, i32* %result
  ret %result1
}
```

Linking (Using Clang)

— Generate object files
— Link objects and dependencies
— Generate executable

	Lexer	Parser	Checker	IRGen	Linking
LOC	627	965	2762	1751	10
Diff.	easy	tricky	hard	easy*	lol

Helpers ~400 LOC (2500 Swift de-mangler)
^ Total is ~6200 SLOC (9000 LOC)

Planned features

Powerful Foreign Function Interface

Language embedded linking support

```
#library "c" libc
#library "/usr/local/lib/libglfw3.dylib" glfw
#library "OpenGL.framework" gl
#library "libraylib.a" raylib
```

Foreign symbol declaration

```
10 initWindow :: (i32, i32, title: string) -> void #foreign raylib "InitWindow"
11 closeWindow :: () -> void #foreign raylib "CloseWindow"
12 setTargetFPS :: (i32) -> void #foreign raylib "SetTargetFPS"
13 windowShouldClose :: () -> bool #foreign raylib "WindowShouldClose"
14 beginDrawing :: () -> void #foreign raylib "BeginDrawing"
15 endDrawing :: () -> void #foreign raylib "EndDrawing"
16 clearBackground :: (color: u32) -> void #foreign raylib "ClearBackground"
17 drawText :: (string, x: i32, y: i32, size: i32, color: u32) -> void #foreign raylib "DrawText"
```

The plan is to support multiple languages

C header imports

Using **#import "dlopen.h"`** you will have an interface file generated, similar to the one above automatically for you.

Demo

Questions?

@BrettRToomey

Cool Hangouts

— http://git.kai-lang.org
— http://docs.kai-lang.org
— https://llvmswift-slack.herokuapp.com