```
7  #library "c" libc
6
5  print :: (string, ..string) -> void #foreign libc "printf"
4
3  greet :: (name: string) -> void {
2      print("Hello, %s\n", name)
1  }
8  █
1  main :: () -> void {
2      greet("Cocoaheads!")
3      print("Meet, Kai!")
4  }
```

Kai

```
kai hello.kai
./main                                        404ms
Hello, Cocoaheads!
Meet, Kai!
🯄
```

NORMAL   hello.kai   unix || kai   66%   8:1

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

Checking

Generates a set of annotations for the AST:

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

```swift
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)

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


define void @main() {
entry:
  br label %return


return:                                           ; preds = %entry
  ret void
}
```

There are lots of these AST Nodes

```swift
indirect enum AstNode {

    case invalid(SourceRange)

    case ident(String, SourceRange)
    case directive(String, args: [AstNode], SourceRange)

    case list([AstNode], SourceRange)

    case ellipsis(AstNode, SourceRange)

    case litInteger(Int64, SourceRange)
    case litFloat(Double, SourceRange)
    case litString(String, SourceRange)

    /// - Parameter type: `typeProc` node
    /// - Note: `type` holds reference to the args
    case litProc(type: AstNode, body: AstNode, SourceRange)

    /// - Note: Used to represent array literals (struct lits in the future)
    case litCompound(elements: [AstNode], SourceRange)

    case declValue(isRuntime: Bool, names: [AstNode], type: AstNode?, values: [AstNode], SourceRange)
    case declImport(path: AstNode, fullpath: String?, importName: AstNode?, SourceRange)
    case declLibrary(path: AstNode, fullpath: String?, libName: AstNode?, SourceRange)

    case exprSubscript(receiver: AstNode, value: AstNode, SourceRange)
    case exprCall(receiver: AstNode, args: [AstNode], SourceRange)
    case exprParen(AstNode, SourceRange)
    case exprUnary(String, expr: AstNode, SourceRange)
    case exprBinary(String, lhs: AstNode, rhs: AstNode, SourceRange)
    case exprTernary(cond: AstNode, AstNode, AstNode, SourceRange)
    case exprSelector(receiver: AstNode, member: AstNode, SourceRange)



    /// Essentially an expr which has it's rvalue thrown away
    case stmtExpr(AstNode)
    case stmtEmpty(SourceRange)
    case stmtAssign(String, lhs: [AstNode], rhs: [AstNode], SourceRange)
    case stmtBlock([AstNode], SourceRange)
    case stmtIf(cond: AstNode, body: AstNode, AstNode?, SourceRange)
    case stmtReturn([AstNode], SourceRange)
    case stmtFor(initializer: AstNode?, cond: AstNode?, post: AstNode?, body: AstNode, SourceRange)
    case stmtDefer(AstNode, SourceRange)
    case stmtBreak(SourceRange)
    case stmtContinue(SourceRange)

    /// - Parameter params:
    case typeProc(params: [AstNode], results: [AstNode], SourceRange)
    case typePointer(type: AstNode, SourceRange)
    case typeNullablePointer(type: AstNode, SourceRange)
    case typeArray(count: AstNode, type: AstNode, SourceRange)
}
```

They all have locations

```swift
extension AstNode {

    var startLocation: SourceLocation {
        return location.lowerBound
    }

    var endLocation: SourceLocation {
        return location.upperBound
    }

    var location: SourceRange {

        switch self {
        case .invalid(let location),
             .ident(_, let location),
             .directive(_, _, let location),
             .list(_, let location),
             .ellipsis(_, let location),
             .litInteger(_, let location),
             .litFloat(_, let location),
             .litString(_, let location),
             .litProc(_, _, let location),
             .litCompound(_, let location),
             .declValue(_, _, _, _, let location),
             .declImport(_, _, _, let location),
             .declLibrary(_, _, _, let location),
             .exprUnary(_, _, let location),
             .exprBinary(_, _, _, let location),
             .exprParen(_, let location),
             .exprSelector(_, _, let location),
             .exprCall(_, _, let location),
             .exprSubscript(_, _, let location),
             .exprTernary(_, _, _, let location),
             .stmtEmpty(let location),
             .stmtAssign(_, _, _, let location),
             .stmtBlock(_, let location),
             .stmtIf(_, _, _, let location),
             .stmtReturn(_, let location),
             .stmtFor(_, _, _, _, let location),
             .stmtDefer(_, let location),
             .stmtBreak(let location),
             .stmtContinue(let location),
             .typeProc(_, _, let location),
             .typePointer(_, let location),
             .typeNullablePointer(_, let location),
             .typeArray(_, _, let location):

            return location

        case .stmtExpr(let expr):
            return expr.location
        }
    }
}
```

They all have a description

```swift
extension AstNode: CustomStringConvertible {

    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"
```

… And a Tree print format

```swift
func pretty(depth: Int = 0, includeParens: Bool = true, specialName: String? = nil) -> String {
    var unlabeled: [String] = []
    var labeled: [(String, String)] = []

    var children: [AstNode] = []

    // used to rename a node with a different short name ie: list node as parameters or results
    var renamedChildren: [(String, AstNode)] = []

    switch self {
    case .invalid(let location):
        labeled.append(("location", location.description))

    case .ident(let ident, _):
        unlabeled.append(ident)

    case .directive(let directive, _, _):
        unlabeled.append(directive)

    case .list(let nodes, _):

        if nodes.reduce(true, { $0.0 && $0.1.isIdent }) {
            unlabeled.append(nodes.commaSeparated)
        } else {
            children.append(contentsOf: nodes)
        }

    case .ellipsis(let expr, _):
        children.append(expr)

    case .litInteger(let val, _):
        unlabeled.append("'" + val.description + "'")

    case .litFloat(let val, _):
        unlabeled.append("'" + val.description + "'")

    case .litString(let val, _):
        unlabeled.append("\"" + val + "\"")

    case .litProc(let type, let body, _):
        labeled.append(("type", type.description))
        guard case .typeProc(let params, let results, _) = type else {
            break // NOTE(vdka): do we want to break?
        }

        let emptyList = AstNode.list([], SourceLocation.unknown ..< .unknown)
        var paramsList = emptyList
        for param in params {
            for decl in explode(param) {
                paramsList = append(paramsList, decl)
            }
        }

        var resultList = emptyList
        for result in results {
            for decl in explode(result) {
                resultList = append(resultList, decl)
            }
        }
        renamedChildren.append(("parameters", paramsList))
        renamedChildren.append(("results", resultList))

        children.append(body)

    case .litCompound(let elements, _):
        children.append(contentsOf: elements)

    case .exprUnary(let op, let expr, _):
        unlabeled.append("'" + op + "'")
        children.append(expr)

    case .exprBinary(let op, let lhs, let rhs, _):
        unlabeled.append("'" + op + "'")
        children.append(lhs)
        children.append(rhs)

    case .exprParen(let expr, _):
        children.append(expr)

    case .exprSelector(let receiver, let selector, _):
        children.append(receiver)
        children.append(selector)

    case .exprCall(let receiver, let args, _):
        unlabeled.append(receiver.description)
        children.append(contentsOf: args)

    case .exprSubscript(let receiver, let value, _):
        unlabeled.append(receiver.description)
        children.append(value)

    case .exprTernary(let cond, let trueBranch, let falseBranch, _):
```

… And a type annotated version of the same Tree print

In total `AST.swift` is just over **1000 lines** of code on it's own.

	Lexer	Parser	Checker	IRGen
LOC	347	616	1616	948
Difficulty	easy	tricky	hard	easy*

Helpers ~400 LOC
^ Total is ~5000 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

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

The plan is to support multiple languages

C header imports

This is a planned feature that `libClang` already provides functionality for.

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

Demo

Questions?