(g)ROOT
Final Report

Samuel Russo Amy Bui Eliza Encherman
Zachary Goldstein Nickolas Gravel

May 6, 2022

Contents

1 Acknowledgments

Project Mentor: Mert Erden

Professor: Richard Townsend

2 Resources

Implementing functional languages: a tutorial. Simon Peyton Jones, David R Lester. Prentice Hall, 1992.

Modern Compiler Implementation in ML. Andrew W. Appel. Cambridge University Press, 2004.

Compilers: Principles, Techniques, and Tools, 2nd Edition. Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. Addison-Wesley, 2006.

Engineering a Compiler, 2th Edition. Keith D. Cooper and Linda Torczon. Morgan Kaufmann, 2012.

OCaml Manual

3 Introduction

g(ROOT) is a general-purpose functional programming language, with a pared-down and minimalistic syntax that makes it useful for educational purposes, such as transitioning imperative programmers to functional languages. The language has a LISP-like syntax and offers lambda expressions, higher-order functions, inferred types, and allows variable redefinitions.

g(ROOT) utilizes the Hindley-Milner algorithm in order to infer the types of expressions, as well as typing and variable environments in order to accomplish closure conversion to resolve lambda expressions (closures use the original values of any free variables at the time the closure was created, rather than using the most recent values of any free variables when the closure is used).

The project originally set out to give the language a built-in n-ary tree data structure as a primitive type. This is where the name and mascot "groot" originates. However, with the challenges of compiling seemingly inherent functional language features and the time constraints, trees are left as a future feature to be implemented.

This document will provide an in-depth language reference manual for g(ROOT), a language tutorial, and details of our implementations and key takeaways from this project.

4 Language Tutorial

4.1 Environment Setup

To run our language, please have the following packages installed:

- `opam-2.1`
- `llvm-13`
- `ocaml-4.13`
- `ocamlbuild-`
- `llc-`
- `gcc-`

If there are compilation issues that may be due to versioning, we have a docker image that may help.

4.2 Compile and Run

Currently does not support linking, so all source code must be in the same file. gROOT file extension is ***.gt***

Compile the g(ROOT) compiler with ONE of these two commands:

- `make`
- `make toplevel.native`

Compile and create executable and intermediate files for any given source file in our language with the format [filename].gt
Intermediate files are stored in toplevel as tmp.ll and tmp.s, respectively

- `make [filename].exe`

Run [filename].exe executable

- `./[filename].exe`

4.3 Using the Language

g(ROOT) uses Lisp-style syntax and allows higher-order functions. All expressions except literals and variable evaluations must be enclosed in parentheses. Whitespace serves no syntactic purpose except to separate tokens and aid in styling.

There is an inbuilt standard basis consisting of the basic algebraic and boolean operations and comparisons, and both named and anonymous user-defined functions can be created using `val` `lambda` and `lambda`, respectively.

Please reference the g(ROOT) Lexical Conventions for more details.

Simple example code:

```
(val x 42)
(printi x)
```

The above compiled code outputs:

```
$ 42
```

Complex example code:

```
(val letterGrade (lambda (test)
  (if (> 90 test)
      'A'
      (if (> 80 test)
          'B'
          (if (> 70 test)
              'C'
              'D'))))))

(printc (letterGrade 89))     (; this prints B ;)

(val computeGrade (lambda (test test2 test3)
  (let ([sum (+ (+ test test2) test3)])
      (let ([avg (/ sum 3)])
          (letterGrade avg)))))

(printc (computeGrade 88 90 91)) (; this prints B ;)

(val letterGrade (lambda (test)
  (if (> 85 test)
      'A'
      (if (> 75 test)
          'B'
          (if (> 65 test)
              'C'
              'D'))))))

(printc (letterGrade 89))        (; this prints A ;)
(printc (computeGrade 88 90 91)) (; this prints B ;)
```

The above compiled code outputs:

```
$ B
$ B
$ A
$ B
```

Definitions and expressions are processed in order, and both local and toplevel definitions exist - local definitions take priority in-scope and disappear once the block in which they were defined is finished. Local definitions defined simultaneously cannot reference each

other (let* does not exist), but nested let statements can get around this, as seen by using sum to define avg.

Redefinitions are allowed, and later usages use the updated values, though if a variable was used in the body of the closure, the closure will always use that value even if the variable is redefined later, hence why the last line prints “B” - computeGrade still uses the earlier definition of letterGrade.

5 Language Reference Manual

5.1 How to read manual

The syntax of the language will be given in BNF-like notation. Non-terminal symbol will be in italic font *like-this*, square brackets [...] denote optional components, curly braces { ... } denote zero or more repetitions of the enclosed component, and parentheses (...) denote a grouping. Note the font, as `[ ... ]` and `( ... )` are syntax requirements later in the manual.

5.2 Lexical Convention

5.2.1 Blanks

The following characters are considered as **blanks**: space, horizontal tab ('`\t`'), newline character ('`\n`'), and carriage return ('`\r`').

Blanks separate adjacent identifiers, literals, expressions, and keywords. They are otherwise ignored.

5.2.2 Comments

Comments are introduced with two adjact characters `(;` and terminated by two adjacent characters `;)`. Multiline comments are allowed with this. Single line comments using `;;` are also allowed for ease.

```
(; This is a comment. ;)

;; This is another comment

(; This is a
   multi-lined comment. ;)
```

5.2.3 Identifiers

Identifiers are sequences of letters, digits, and ASCII characters, starting with any character that isn't the underscore. Letters will refer to the below ranges of ASCII characters. *Identifiers may not start with an underscore character*, and may not be any of the reserved character sequences.

⟨*ident*⟩ ::= *letter* { *letter* | _ }

⟨*letter*⟩ ::= `!...&` | `*...:` | `<...Z` | `` ` ...z `` | `~` | |

5.2.4 Integer Literals

An integer literal is a decimal, represented by a sequence of one or more digits, optionally preceded by a minus sign.

⟨*integer-literal*⟩ ::= [-] *digit* {*digit*}

⟨*digit*⟩ ::= 0...9

5.2.5 Boolean Literals

Boolean literals are represented by two adjacent characters; the first is the octothorp character (#), and it is immediately followed by either the t or the f character.

⟨*boolean-literal*⟩ ::= # (t | f)

5.2.6 Character Literals

Character literals are a single character enclosed by two ' (single-quote) characters.

5.2.7 Operators

All of the following operators are prefix characters or prefixed characters read as single token. Binary operators are expected to be followed by two expressions, unary operators are expected to be followed by one expression.

⟨*operator*⟩ ::= (*unary-operator* | *binary-operator*)

⟨*unary-operator*⟩ ::= ! | -

⟨*binary-operator*⟩ ::= + | - | * | / | mod
| == | < | > | ≤ | ≥ | !=
| && | ||

5.3 Keywords

The below identifiers are reserved keywords and cannot be used except in their capacity as reserve keywords:

```
if      val
let     lambda
```

The following character sequence are also keywords:

```
==      +       &&      >       '
!=      -       ||      mod     #t
<=      *       !       (       #f
>=      /       <       )
```

The following tree-related keywords are still recognized, but their uses are unimplemented. Please be aware: `leaf` `elm` `tree` `cld` `sib`

5.3.1 Syntax

See Definitions and Expression for concrete syntax for each definition and expressions, with detailed examples.

5.4 Values

Base Values

5.4.1 Integer numbers

Integer values are integer numbers in range from -2^{32} to $2^{32}-1$, similar to LLVM's integers, and may support a wider range of integer values on other machines, such as -2^{64} to $2^{64}-1$ on a 64-bit machine.

5.4.2 Boolean values

Booleans have two values. `#t` evaluates to the boolean value `true`, and `#f` evaluates to the boolean value `false`.

5.4.3 Characters

Character values are 8-bit integers between 0 and 255, and follow ASCII standard.

5.4.4 Functions

Functional values are mappings from values to value.

5.5 Types

5.6 Definitions and Expressions

⟨*def*⟩ ::= `(` `val` *ident* *expr* `)`
| *expr*

⟨*expr*⟩ ::= *literal*
| *ident*
| *unary-operator* *expr*
| `(` *binary-operator* *expr* *expr* `)`
| `(` *ident* {*expr*} `)`
| `(` `let` `(` [*ident* *expr*] {[*ident* *expr*]} `)` *expr* `)`
| `(` `if` *expr* *expr* *expr* `)`
| `(` `lambda` `(` {*arguments*} `)` *expr* `)`

⟨*literal*⟩ ::= *integer-literal* | *boolean-literal* | *character* | leaf

⟨*arguments*⟩ ::= ϵ
| *ident* :: *arguments*

Expressions are values or parenthetical expressions.

5.6.1 Values

see Values.

5.6.2 Parenthetical expressions

Parenthetical expressions are always within parentheses and include function application, lambda expressions, global and local definitions, binary and unary operations, and if-statements. In the above concrete syntax, the parentheses in this font, `( ... )`, are syntax requirements, rather than denoting a grouping which is given by (...)

5.6.3 Function application

Function application in (g)ROOT always returns a value, and is written as expression to apply, followed by a list of zero or more expressions, which are its arguments. The arguments are not separated from the applied expression by parentheses. (g)ROOT has first-class functions, therefore functions can be passed as arguments. While partial application is allowed, this feature is use-at-your-own-risk as it may have undefined behavior due to a the recognized bug in type monomorphization.

Example:

```
(foo)
(bar a b)
((baz x) y)
```

5.6.4 Lambda Expression

Lambda expressions are accomplished with the `lambda` keyword, a parentheses-enclosed list of 0 or more identifiers as formal arguments, followed by the expression that may use those arguments and/or any free variables. Nesting of lambda expressions is allowed, but not recommended for the same reason stated above for why partial function application is not recommended.

Example:

```
(lambda () #t)
(lambda (x) x)
(lambda (x y) (+ x y))

(lambda (a) (add2 a b))
```

5.6.5 Global definitions

Global definitions are accomplished using the `val` keyword, followed by an identifier, followed by the expression which is to be bound to that value.

Example:

```
(val x 4)
(val y (+ x 5))
(val foo (lambda (arg) ( * arg arg)))
```

Calling a global definition with a preexisting identifier will re-bind that identifier to the new value - onlys allowed at the top level, and new definition must always of the same type as the previous definition.

5.6.6 Local definitions

Local definitions are found with the `let` expressions, which is the `let` keyword followed by the identifier(s) and the expression(s) to be bound to it, followed by the expression that local variable may be used. Let expressions must have at least one local binding.

Example:

```
(let ([x 4]) (+ 2 x))  (; return 6 ;)
(let ([x 4]) x)        (; return 4 ;)
(let () y)             (; not allowed! ;)
```

Variables defined within the let binding are not defined outside of it, while variables globally relative to the let can be accessed within it. Since let bindings are a type of expression, this allows for chained let bindings.

Example:

```
(let x 4
   (let y 5
```

```
      (let z 9
         (+ x (- y z)))))
```

5.6.7 If-expression

If-expressions are the only form of control flow in (g)ROOT, and are always formed with the `if` keyword followed by three expressions (the *condition*, the *true case* and the *false case*). Omission of the false case is a syntax error, and the expressions are not separated by parentheses, brackets, or keywords.

Example:

```
(if #t 1 2)
(if (< 3 4)
   (+ x y)
   (- x y))
```

5.6.8 Unary operators

Unary operations (used for boolean or signed negation) must not be enclosed in parentheses. They are accomplished with a unary operator in front of the expression they negate.

Example:

```
-3
-(+ 3 4)
-(if #t 2 3)
-x

!#t
!x
!(expr)
```

5.6.9 Binary operators

The general use of binary operators is as follows: (*binary-operator* $expr_1$ $expr_2$)

The **arithmetic operators** (`+` , `-` , `*` , `/` , `mod`) take two expressions that evaluate to integers.

The **comparator operators** (`==` , $<$, $>$, $\leq$, $\geq$, `!=`) take two expressions that both evaluate to either integers or booleans.

The **boolean operators** (`&&` , `||`) take two expressions that evaluate to booleans.

5.7 Functions

5.7.1 Built-In Functions

Note: Throughout this section, the character pair `tr` is used as shorthand to represent a tree value, an identifier bound to a tree value.

There are four built-in functions which are integral to the tree data type. It is a checked-runtime error to call `elm, sib, cld` on a `leaf`:

1. `elm`

   Usage: `(elm tr)`

   Purpose: returns the element of the provided tree value `(tr)`

2. `sib`

   Usage: `(sib tr)`

   Purpose: returns the sibling tree of the provided tree value `(tr)`

3. `cld`

   Usage: `(cld tr)`

   Purpose: returns the child tree of the provided tree value `(tr)`

4. `leaf?`

   Usage: `(leaf?  tr)`

   Purpose: Returns boolean val `#t` iff `tr` is a leaf.

5. `tree`

   Usage: `(tree ELEMENT SIBLING CHILD)`

   Purpose: constructs a tree value containing the value ELEMENT, with sibling SIBLING and child CHILD

5.7.2 Standard Library Functions

Note: Throughout this section, the character pair `tr` is used as shorthand to represent a tree value, an identifier bound to a tree value.

The first set of functions introduced in the standard library allow for programmers to use some of their beloved list functions present in other languages:

1. `(cons a b)`

   Creates a 1-ary tree approximating the functionality of single-dimensional list in other functional languages. `a` is any value not a tree, `b` may be `leaf` or `nil`.

2. `(car tr)`

   Extracts the first value in the pair.

3. `(cdr tr)`

   Extracts the second value in the pair.

4. `(null? tr)`

   A predicate function that evaluates to true iff the pseudo-list is empty.

5. `(val nil leaf)`

   A value `nil` that mirrors the built-in keyword leaf.

6. `(append xs ys)`

   Append the list of elements in `ys` to the end of `xs`.

7. `(revapp xs ys)`

   Append the list of elements in `ys` to the reverse of `xs`.

Now some useful tree functions!

1. `(graft oak fir)`

   Appends a sibling tree `fir` to some other tree `oak`. The result of calling this function is a new tree in which `fir` is the rightmost sibling of `oak`. This function is extremely useful when re-shaping trees.

2. `(level-flatten tr)`

   Flattens the provided tree to a 1-ary tree, arranging elements in level-order.

3. `(pre-flatten tr)`

   Flattens the provided tree to a 1-ary tree, arranging elements in pre-order.

4. `(post-flatten tr)`

   Flattens the provided tree to a 1-ary tree, arranging elements in post-order.

5. `(map f tr)` Maps a function `f` over every element of the provided tree `tr`.

6. `(filter p? tr)`

   Constructs a new tree containing only elements that satisfy the predicate function `p?`, which must return a boolean value.

7. `(level-fold fn base tr)`

   Folds a tree, visiting each element in a level-order traversal. Note that the accumulation function `fn` must take TWO arguments, the first of which is the current element, and second is the rest of the tree.

8. `(pre-fold fn base tr)`

   Folds a tree, visiting each element in a pre-order traversal. Note that the accumulation function `fn` must take TWO arguments, the first of which is the current element, and second is the rest of the tree.

9. `(post-fold fn base tr)`

   Folds a tree, visiting each element in a post-order traversal. Note that the accumulation function `fn` must take TWO arguments, the first of which is the current element, and second is the rest of the tree.

10. `(fold fn base tr)`

    Folds a tree, in a more intuitively tree-like manner. Note that the accumulation function `fn` must take THREE arguments: the value of the current node, the sibling accumulator, and the child.accumulator.

11. `(node-count tr)`

    Evaluates to the number of nodes in the provided tree.

12. `(height tr)`

    Evaluates to the height of the provided tree.

Higher-Order Functions:

1. `(curry f)`

   Allows for the partial application of a two-argument function.

2. `(uncurry f)`

   Uncurries a previously curried function such that both of its arguments must be provided at the same time.

3. `(o f g)`

   Given two single-argument functions, returns a single function which is the composition of `f` and `g`.

4. `(flip f)`

   Given a two argument function, reverses the order in which the arguments are evaluated and passed to the function.

5. `(flurry f)`

   Given a two argument function, reverses the order in which the arguments are evaluated and passed to the function, and allows for the partial application of that function. This function both flips and curries `f` (hence the fun name).

5.8 LRM Appendix

```
(; We can define standard list functions with our tree data type! Fun! ;)

(define (cons a b) (tree a leaf (tree b leaf leaf)))
(define (car  tr) (elm tr))
(define (cdr  tr) (cld tr))
(define (nil   ) leaf)
(define (null? tr) (leaf? tree))

(; a list function! ;)
(define append (xs ys)
    (if (null? xs)
        ys
        (cons (car xs) (append (cdr xs) ys))
    )
)

(; another list function! ;)
(define revapp (xs ys) ; (reverse xs) followed by ys
    (if (null? xs)
        ys
        (revapp (cdr xs) (cons (car xs) ys))
    )
)

(; appends a sibling tree (fir) to tree (oak) ;)
(define (graft oak fir)
    (if (leaf? oak)
        fir
        (tree (elm oak) (graft (sib oak) fir) (cld oak))
    )
)

(; attempt 4
   this function flattens a tree, level-order ;)
(define (level-flatten tr)
    (if (leaf? tr)
        leaf
        (tree
            (elm tr)
            leaf
            (level-flatten (graft (sib tr) (cld tr)))
        )
    )
)
```

flattening functions!

```
(; this function flattens a tree, pre-order ;)
```

```
(define (pre-flatten tr)
    (if (leaf? tr)
        leaf
        (tree
            (elm tr)
            leaf
            (pre-flatten (graft (cld tr) (sib tr)))
        )
    )
)


(; this function flattens a tree, post-order ;)
(define (post-flatten tr)
    (if (leaf? tr)
        leaf
        (if (leaf? (cld tr))
            (tree
                (elm tr)
                leaf
                (post-flatten (sib tr)))
            (post-flatten (graft
                                (graft
                                    (cld tr)
                                    (tree (elm tr) leaf leaf))
                                (sib tr))
            )
        )
    )
)
```

Let's create some higher order functions! Yay!

```
(define (curry f)   (lambda (x)  (lambda (y) (f x y))))
(define (uncurry f) (lambda (x y) ((f x) y)))
(define (o     f g) (lambda (x) (f (g x))))
(define (flip  f)   (lambda (x)  (lambda (y) (f y x))))
```

function: (flurry func)

- precondition:

  func is a function that takes exactly two args

- evalutation:

  evaluates to a curried form of func, where:

  1. the first value passed to the curried function is treated as the second argument to (func).
  2. the second value passed to the curried function is treated as the first argument

to (func).

- note: flurry is a contraction of "flip & curry", and we're very proud of the wordplay there.

```
(define flurry (func)
    (curry (lambda (a b) (func b a)))
)


(; mapping function! ;)
(define (map f tr)
    (if (leaf? tr)
        leaf
        (tree
            (f (elm tr))
            (map f (sib tr))
            (map f (cld tr))
        )
    )
)


(; filter function! ;)
(define filter (p? tr)
    (if (leaf? tr)
        leaf
        (if (p? (elm tr))
            (tree (elm tr) (filter p? (sib tr)) (filter p? (cld tr)))
            (filter p? (graft (cld tr) (sib tr)))
        )
    )
)




(; folding functions! ;)

(; attempt 4
   this function folds a tree, level-order ;)
(define (level-fold fn base tr)
    (if (leaf? tr)
        base
        (fn
            (elm tr)
            (level-fold (graft (sib tr) (cld tr)))
        )
    )
)
```

```
(; this function folds a tree, pre-order ;)
(define (pre-fold fn base tr)
    (if (leaf? tr)
        base
        (fn
            (elm tr)
            (pre-fold (graft (cld tr) (sib tr)))
        )
    )
)

(; this function folds a tree, post-order ;)
(define (post-fold fn base tr)
    (if (leaf? tr)
        base
        (if (leaf? (cld tr))
            (fn (elm tr) (post-fold (sib tr)))
            (post-fold (graft
                           (graft (cld tr) (tree (elm tr) leaf leaf))
                               (sib tr)
                       )
            )
        )
    )
)

(; once we implement first-class functions, we
   should find that the behaviors of these three
   functions are exactly equivalent to their
   lower-class cousins ;)
(define (level-fold-flatten tr) (level-fold cons nil tr))
(define (pre-fold-flatten tr) (pre-fold cons nil tr))
(define (post-fold-flatten tr) (post-fold cons nil tr))

(; this function folds a tree, in a more intuitively
   tree-like manner
   fn must take in three params:
   - the current value
   - the sibling accumulator
   - the child accumulator ;)
(define (fold fn base tr)
    (if (leaf? tr)
        base
        (fn
            (elm tr)
            (fold fn base (sib tr))
            (fold fn base (cld tr))
        )
```

```
    )
)



(; some more fun folding functions! ;)
(define (node-count tr)
    (pre-fold (lambda (elem count) (+ 1 count)) 0 tr)
)

(; this uses the tree fold to calculate the height! ;)
(define (height tr)
    (fold
        (lambda (elem clds sibs) (+ 1 (max clds sibs)))
        0 tr
    )
)

(define (height-no-fold tr)
    (if (leaf? tr)
        0
        (max (height (sib tr))
            (+ 1 (height (cld tr)))
        )
    )
)
```

6 Project Plan

6.1 Planning, Specification, and Development

Initial planning involved going through several different ideas for a language before we settled on a functional language with an achievable feature that stands out. Most members of the group primarily had experience with imperative languages, so this topic provided an interesting challenge for everyone.

We came up with a preliminary syntax for which we could parse and described in our LRM. The syntax was finalized (to include definitions) once parsing of expressions worked and as we moved on to more complex phases of the compiler.

We used GitLab for organization and version control, and relied on clear communication over text, Discord, and in-person to keep each other up-to-date with responsibilities, new tasks, scheduling, road blocks, and goals reached. Every feature was worked on a separate branch, and merges were usually done as a group to resolve merge conflicts between branches together, if any.

Phase 1 (Scanner/Parser) was done as a group. In subsequent phases, different tasks were divided amongsts group members depending on interest, with frequent collaboration with project mentor and other members as the need arises and to double-check each others work. Different responsibilities are described below.

We did not have an official style-guide, and relied on everyone to keep their work readable and well-documented. A few final passes through each file was done to ensure good style.

6.2 Project Timeline

For the most part, we tried to finish as much of a deliverable as possible before a given deadline. For subtasks without any hard due dates, finish-dates became more ambiguous, because a member's work tested by another member of the group, often times, revealed a new bug or a new feature that needs to be implemented before the original task is complete. Due to some of these challenges, we often found ourselves working up to a deadline or working pass an expected finish date.

Goal	**Finish or Submit Date**
Final Language Idea	Feb. 2
Project Proposal	Feb. 4
Phase 1 test script	Feb. 23
Phase 1: Scanner & Parser	Feb. 24
Language Reference Manual	Feb. 28
Final Language Syntax	Mar. 5
Start planning and implementing type-inferencer	Mar. 5
Start planning code generation	Mar. 5
Start planning closure conversion	Mar. 11
Semant for purposes of testing Conversion and Codegen	Mar. 13
Partial Semant and Codegen Module that forces a printing	Mar. 11
New phase 2 test script and reference outputs for all phases	Mar. 24
Phase 2: Hello World	Mar. 28
Conversion	Apr. 15
Codegen	Apr. 19
Extended Test Suite	Apr. 20
Type Inferencer	Apr. 23
Start Monomorphization to deal with infer's polymorphic types	Apr. 23
Debug Conversion & Codegen	Apr. 29
thread primitives through each module	Apr. 29
Debug Infer	May 2
Mono	May 2
Create HAST & Hof (conversion I) phase and fit with Conversion (conversion II) to allow HOFs	May 5
Extend testing script	May 5
Presentation slides	May 5
Presentation	May 6
Final Paper & Submission	May 7

6.3 Roles and Responsibilities

These were the originally assigned recommended roles:

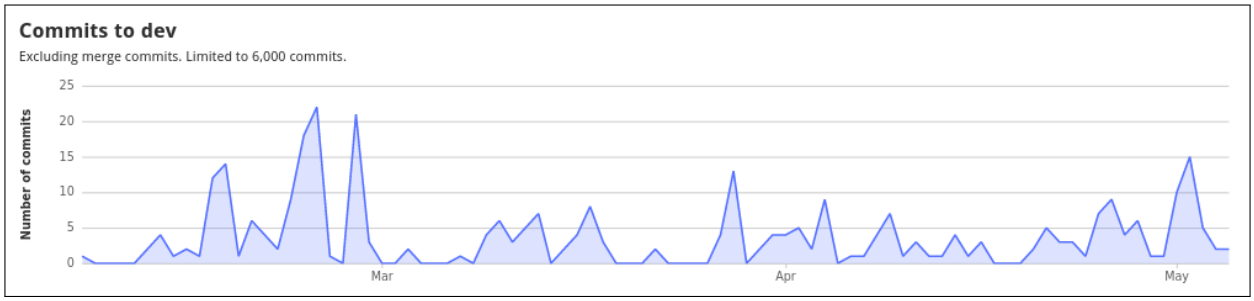
Name	Role
Eliza Encherman	Manager
Sam Russo	Tester
Zachary Goldstein	Language Guru
Nickolas Gravel	System Architect
Amy Bui	Facilitator

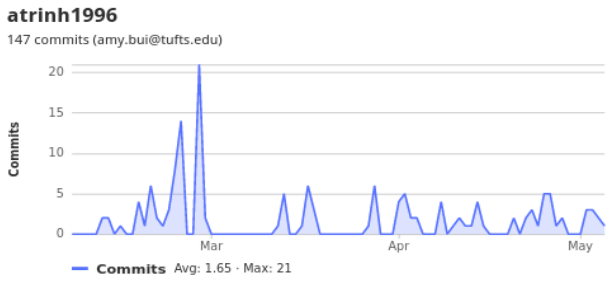
As the project progressed and implementing features of a functional language became more difficult, we abandoned these roles and each subgroup or member focused on progressively implementing different language features. Everyone was responsible intially testing their own feature and subsequent debugging, but we were also responsible for checking each others work and providing "fresh eyes" when testing someone else's implemented feature. Here is a list of each person's primary responsibility:

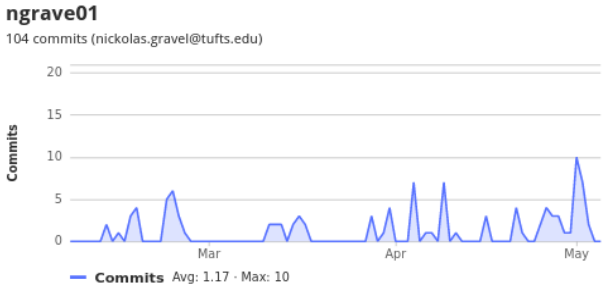
Name	Responsibilities
Eliza Encherman	Primitive recognition and testing in Scanner/Parser, Scope, Infer, Conversion, and primitive code generation
Sam Russo	Type Inferencer
Zachary Goldstein	Errors & Warnings, test suite, Docker environment
Nickolas Gravel	Type Inferencer
Amy Bui	monomorphizer, conversion I & II, codegen

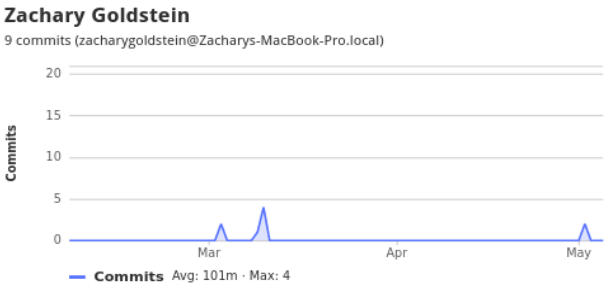
6.4 Development Tools

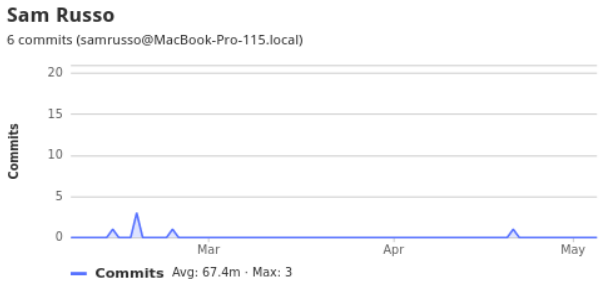
6.5 Project Logs

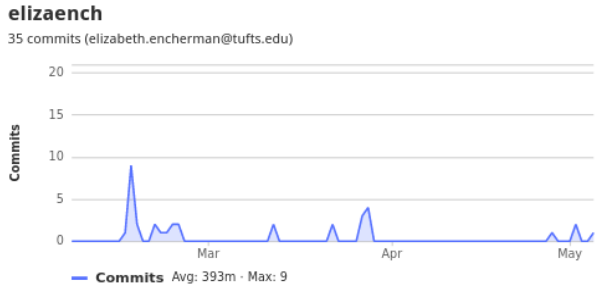




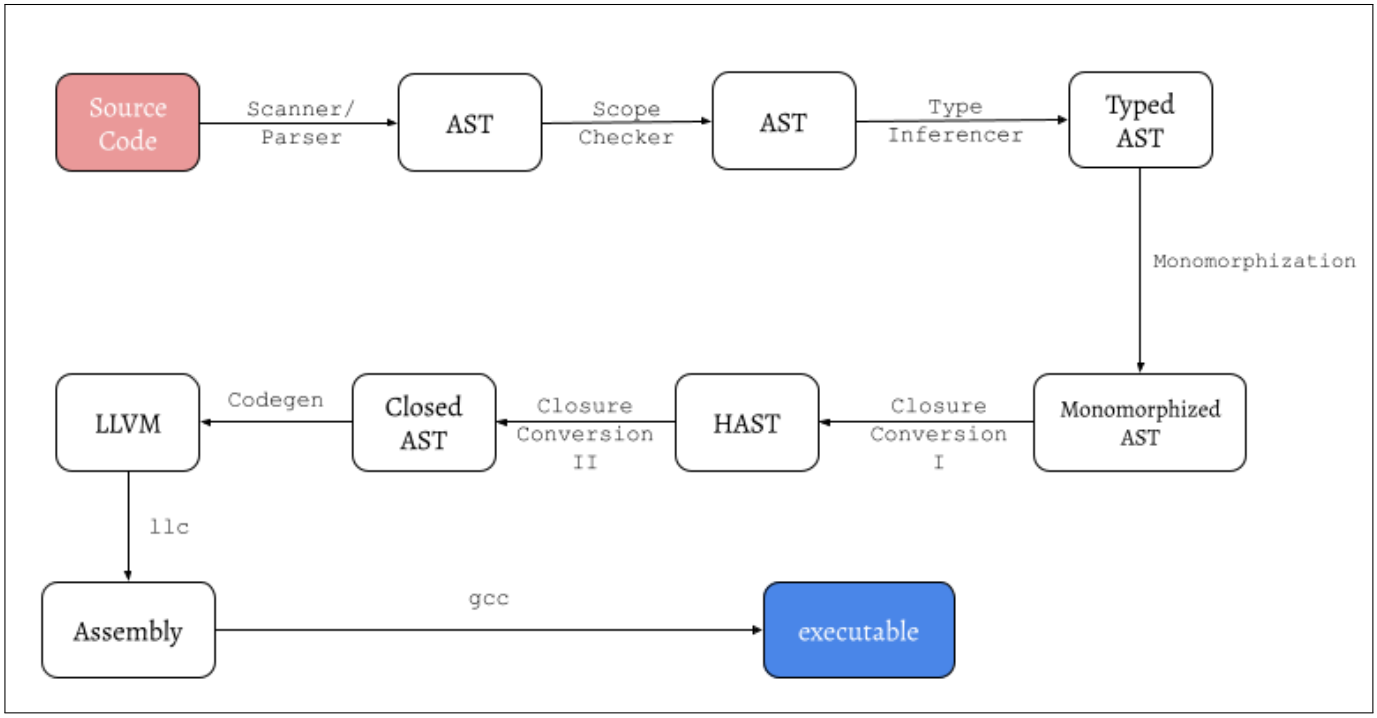








7 Architectural Design



8 Test Plan

9 Lessons Learned

9.1 Amy Bui

My main takeaway is that compiler writing, or even writing just a single optimization for a compiler, is a very complicated endeavor. We did not end up with a lot of our original goals and features we intended for our functional language, but the detours we took while exploring this whole other paradigm was very enriching, and I was never bored and always challenged. Everyone should write a compiler at least once before graduating. This was a great project in demonstrating the practical application of the concepts and theory we learnt in 170 and 105, and I'd recommend taking both classes before 107 or at the same time for a more comfortable time. I have a lot of appreciation for people who do research and contribute to the field of functional languages, and the complexity of lambda calculus. My advice to future students is that you should have a burning desire to implement type inferencing and lambda calculus before you settle on a a functional language as your project topic; I never knew how much we took static typing and statment blocks for granted. Or you can get a thrill from just taking on this challenge.

9.2 Nickolas Gravel

Compiler writing is a long, long journey filled with various pitfalls, unexpected challenges, dragons, and possibly avocados. For myself, I found the learning curve to be slightly steeper than my peers. Before this class, I had not taken CS105 Programming Languages or had any experience coding in a functional language. So, in the beginning, I encountered some tribulation understanding these concepts. Effectively, I not only needed to learn the basics of compiler writing, but needed to learn the basic components of a programming language, and the basics of coding in a functional language as well.

Then came type inferencing, a module that we had greatly underestimated the amount of time and work it would take to implement. I was one of the main programmers that worked on implementing this module. Again, I found that having some previous experience with type inferencing would have been a huge help here...but we persevered! I dove into the Hadley-Milner type system algorithm, and with the guidance of Mert and Michael Ryan Clarkson, after about a month of coding the and a week of debugging we had a working type inferencer!

All in all, building a compiler is no joke. Even seemingly trivial steps in implementing a compiler were found to be not so trivial, requiring us to brain storm creative solutions to get to the next step. For anyone building a compiler, I recommend spending ample time writing up a thorough design. Having a written design indicating which data structures and data types were being passed and returned from each module would have helped in overall organization and may have prevented some of the various bugs that we encountered. Future students, if you are set on building a language with inferred types make sure you have a strong understanding how to implement type inferencing and design your inferencer thoroughly before implementing it.

Designing a programming language and building a compiler for it was riveting experience. I learned so much about functional languages, type inferencing, and what it really takes

to compile source code down to a final executable. Everyone with any interest in low-level programming should take this course, but I would recommend taking a course in programming languages more gradual learning curve.

9.3 Sam Russo

There are so many independent aspects to working on a compiler, from language design/syntax to memory to warnings to additional features. Coming up with solutions to the problems that we found in each of these domains and dealing with bugs was challenging, but it was also hard to ensure that the choices we made in each of the domains worked well with each other. If one person decided to do thing x while working in one area, sometimes we would realize later that that decision was incompatible or poorly compatible with a decision someone else made in a different domain at the same time.

Working with a group as big as ours for as long as we did (and I know in the scheme of things our team wasn't that big and we weren't working together for that long) is really hard. Challenges were mostly accountability (definitely including my own) and stemmed from everyone's having different schedules and different workloads. Different workloads meant that people 1) prioritized our project different amounts, 2) had differing amounts of time to dedicate to it, and 3) had different ideas of how to spend the time when we worked on it together. Specifically, the grad students in our group had much more time to work on the project, which led to pretty different levels of contribution. For future mixed grad-undergrad groups, I would think about this before committing to such a group, and if people decide to stick with it, then they should be very clear around group expectations.

9.4 Eliza Encherman

- Takeaway:
- Advice:

9.5 Zachary Goldstein

- Takeaway:
- Advice: