



# (g)ROOT Final Report

Samuel Russo   Amy Bui   Eliza Encherian  
Zachary Goldstein   Nickolas Gravel

May 7, 2022

# Contents

<b>1</b>	<b>Acknowledgments</b>	<b>4</b>
<b>2</b>	<b>Resources</b>	<b>4</b>
<b>3</b>	<b>Introduction</b>	<b>4</b>
<b>4</b>	<b>Language Tutorial</b>	<b>5</b>
4.1	Environment Setup . . . . .	5
4.2	Compile and Run . . . . .	5
4.3	Compiler Flags and other Intermediates . . . . .	6
4.4	Using the Language . . . . .	6
<b>5</b>	<b>Language Reference Manual</b>	<b>8</b>
5.1	How to read manual . . . . .	8
5.2	Lexical Convention . . . . .	8
5.2.1	Blanks . . . . .	8
5.2.2	Comments . . . . .	8
5.2.3	Identifiers . . . . .	8
5.2.4	Integer Literals . . . . .	9
5.2.5	Boolean Literals . . . . .	9
5.2.6	Character Literals . . . . .	9
5.3	Reserved Keywords . . . . .	9
5.3.1	Syntax . . . . .	9
5.4	Values . . . . .	10
5.4.1	Integer numbers . . . . .	10
5.4.2	Boolean values . . . . .	10
5.4.3	Characters . . . . .	10
5.4.4	Functions . . . . .	10
5.5	Types . . . . .	11
5.5.1	Type Inferencing . . . . .	11
5.6	Definitions and Expressions . . . . .	12
5.6.1	Values . . . . .	12
5.6.2	Parenthetical expressions . . . . .	12
5.6.3	Function application . . . . .	12
5.6.4	Lambda Expression . . . . .	13
5.6.5	Global definitions . . . . .	13
5.6.6	Local definitions . . . . .	14
5.6.7	If-expression . . . . .	14
5.7	Functions . . . . .	15
5.7.1	Standard Basis . . . . .	15
5.7.2	Higher-Order Functions . . . . .	17
<b>6</b>	<b>Project Plan</b>	<b>18</b>
6.1	Planning, Specification, and Development . . . . .	18
6.2	Project Timeline . . . . .	19

6.3	Roles and Responsibilities	20
6.4	Development Tools	20
6.5	Project Logs	21
<b>7</b>	<b>Architectural Design</b>	<b>22</b>
7.1	Scanner/Parser	22
7.2	Scope Checker	22
7.3	Type Inferencer	23
7.4	Monomorphization	23
7.5	Closure Conversion I (Hof)	24
7.6	Closure Conversion II (Conversion)	24
7.7	Code Generation	25
<b>8</b>	<b>Test Plan</b>	<b>26</b>
8.1	Test Method	26
8.2	Three Test Samples	28
8.2.1	Sample 1: Nested If	28
8.2.2	Sample 2: Closure	30
8.2.3	Sample 3: Higher-Order Function	32
<b>9</b>	<b>Lessons Learned</b>	<b>38</b>
9.1	Amy Bui	38
9.2	Nickolas Gravel	38
9.3	Sam Russo	39
9.4	Eliza Encherman	39
9.5	Zachary Goldstein	40
<b>10</b>	<b>Reach Goals &amp; Future Changes</b>	<b>41</b>
<b>11</b>	<b>Appendix</b>	<b>42</b>
11.1	toplevel.ml	42
11.2	ast.ml	44
11.3	scanner.mll	46
11.4	parser.mly	49
11.5	scope.ml	52
11.6	tast.ml	54
11.7	infer.ml	58
11.8	mast.ml	65
11.9	mono.ml	68
11.10	hast.ml	76
11.11	hof.ml	80
11.12	cast.ml	87
11.13	conversion.ml	92
11.14	llgtype.ml	99
11.15	codegen.ml	100
11.16	diagnostic.ml	112
11.17	git etiquette	115

# 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 Programming: Correct + Efficient + Beautiful, Michael R. Clarkson et al. 2022

[OCaml Manual](#)

## 3 Introduction

g(ROOT) is a general-purpose functional programming language, with a abbreviated and minimalist 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 top-level variable redefinitions.

g(ROOT) utilizes the Hindley-Milner type system 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-0.14.0`
- `gcc-11.3.1`

If there are compilation issues that may be due to versioning, we have a [docker image](#) that may help.

To maximize the portability of our language development environment, we developed a docker image. You may opt to install the dependencies and resolve issues on your own machine if you choose, but working within the docker image will guarantee success. To initialize the development environment docker container, simply enter:

```
make dev
```

### 4.2 Compile and Run

*Currently does not support linking, so all source code must be in the same file. The  $g(ROOT)$  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 Compiler Flags and other Intermediates

*The following compiler flag options are available for other intermediate representations:*

```
$ ./toplevel --help
$ usage: ./toplevel.native [-a|-n|-t|-m|-h|-v|-l|-c] [file.gt]
$ -a Print the AST (default)
$ -n Print the AST (name-checking)
$ -t Print the TAST
$ -m Print the MAST
$ -h Print the HAST
$ -v Print the CAST
$ -l Print the generated LLVM IR
$ -c Check and print the generated LLVM IR
$ -help Display this list of options
$ --help Display this list of options
```

## 4.4 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:

```
1 (val x 42)
2 (printi x)
```

The above compiled code outputs:

```
$ 42
```

Complex example code:

```
1 (val letterGrade (lambda (test)
2   (if (> 90 test)
3     'A'
4     (if (> 80 test)
5       'B'
6       (if (> 70 test)
7         'C'
8         'D')))))
9
10 (printc (letterGrade 89))    (; this prints B ;)
11
```

```

12 (val computeGrade (lambda (test test2 test3)
13   (let ([sum (+ (+ test test2) test3)])
14     (let ([avg (/ sum 3)])
15       (letterGrade avg))))
16
17 (printc (computeGrade 88 90 91)) (; this prints B ;)
18
19 (val letterGrade (lambda (test)
20   (if (> 85 test)
21     'A'
22     (if (> 75 test)
23       'B'
24       (if (> 65 test)
25         'C'
26         'D')))))
27
28 (printc (letterGrade 89))          (; this prints A ;)
29 (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 adjacent characters (`;` and terminated by two adjacent characters `;`). Multiline comments are allowed with this. Single line comments using `;` are also allowed for ease. Nested comments are not supported.

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

;; This is another comment

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

(; This is a multi-lined comment
;; still the same comment
(; still the same comment
  comment ends ;)
This is not in a comment, will produce an error ;)
```

#### 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](#).

$$\langle \textit{ident} \rangle ::= \textit{letter} \{ \textit{letter} \mid \_ \}$$
$$\langle \textit{letter} \rangle ::= ! \dots \& \mid * \dots : \mid < \dots Z \mid ` \dots z \mid \sim \mid |$$



### 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.

$$\langle \textit{integer-literal} \rangle ::= [-] \textit{digit} \{ \textit{digit} \}$$
$$\langle \textit{digit} \rangle ::= 0 \dots 9$$

### 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.

$$\langle \textit{boolean-literal} \rangle ::= \# (\textbf{t} \mid \textbf{f})$$

### 5.2.6 Character Literals

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

## 5.3 Reserved Keywords

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

```
if      val      #t      '  
let     lambda   #f      (      )
```

The following tree-related keywords are still recognized and reserved, but their uses are unimplemented. Please be aware:

```
leaf  
  
tree
```

#### 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

g(ROOT) has an **inferred typing** system with Int, Char, and Bool primitive data types. The language will try its best to type expressions based on their use in the program utilizing the known types of primitives and literals. Currently, there are no predefined data structure types.

The primitive types recognized are as follows:

- **integers**
- **booleans**
- **characters**

Other types that can be inferred are **function** type and **polymorphic** type.

Most convention represent polymorphic type variable using alphabet letters, ie. 'a. You will see these represented in g(ROOT)'s TAST and MAST with numbers, ie. '1.

### 5.5.1 Type Inferencing

g(ROOT) is a statically typed language meaning every expression is given a type at compile time. Furthermore, g(ROOT) is an implicitly typed language, meaning the types of expressions are inferred based on known primitive types and how the expressions are used. This is known as constraint-based type inference, or the Hadley-Milner type system method. At a high level, the algorithm works by assigning "type variables" to expressions whose type is still unknown. Then we proceed to gather more information about the expression by collecting more constraints and solving them through a series of substitutions. Polymorphism is supported.

## 5.6 Definitions and Expressions

$\langle def \rangle$	$::=$ ( val <i>ident</i> <i>expr</i> )   <i>expr</i>
$\langle expr \rangle$	$::=$ <i>literal</i>   <i>ident</i>   ( <i>binary-operator</i> <i>expr</i> <i>expr</i> )   ( <i>ident</i> { <i>expr</i> } )   ( let ( [ <i>ident</i> <i>expr</i> ] { [ <i>ident</i> <i>expr</i> ] } ) <i>expr</i> )   ( if <i>expr</i> <i>expr</i> <i>expr</i> )   ( lambda ( { <i>arguments</i> } ) <i>expr</i> )
$\langle literal \rangle$	$::=$ <i>integer-literal</i>   <i>boolean-literal</i>   <i>character</i>   <i>leaf</i>
$\langle arguments \rangle$	$::=$ $\epsilon$   <i>ident</i> :: <i>arguments</i>

g(ROOT) Concrete Syntax

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 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 - only allowed at the top level, and new definition must always be 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))
```

## 5.7 Functions

### 5.7.1 Standard Basis

All of the following primitive functions are defined in a standard basis, but can be redefined by the user at their own risk. All integer algebraic operations are signed.

- **printi**

Type:  $int \rightarrow int$

Purpose: sends the string of an integer to standard out.

- **putc**

Type:  $char \rightarrow int$

Purpose: sends the string of an character to standard out.

- **printb**

Type:  $bool \rightarrow int$

Purpose: sends the string of a boolean to standard out using the `#t` or `#f` representation of the boolean.

- **+**

Type:  $int\ int \rightarrow int$

Purpose: integer addition

- **-**

Type:  $int\ int \rightarrow int$

Purpose: integer subtraction

- **\***

Type:  $int\ int \rightarrow int$

Purpose: integer multiplication

- **/**

Type:  $int\ int \rightarrow int$

Purpose: integer division (NOTE: divide-by-zero produces undefined behavior)

- **mod**

Type:  $int\ int \rightarrow int$

Purpose: integer modular division

- **<**

Type:  $int\ int \rightarrow bool$

Purpose: integer comparison

- $\leq$   
Type:  $int\ int \rightarrow bool$   
Purpose: integer comparison
- $>$   
Type:  $int\ int \rightarrow bool$   
Purpose: integer comparison
- $\geq$   
Type:  $int\ int \rightarrow bool$   
Purpose: integer comparison
- $=i$   
Type:  $int\ int \rightarrow bool$   
Purpose: integer equality
- $\neq i$   
Type:  $int\ int \rightarrow bool$   
Purpose: integer inequality
- $\sim$   
Type:  $int \rightarrow int$   
Purpose: integer negation
- `not`  
Type:  $bool \rightarrow bool$   
Purpose: boolean negation
- $\&\&$   
Type:  $bool\ bool \rightarrow bool$   
Purpose: boolean and
- $||$   
Type:  $bool\ bool \rightarrow bool$   
Purpose: boolean or



### 5.7.2 Higher-Order Functions

User-defined functions are just like values in g(ROOT). In other words, functions may be passed as arguments to other functions, and may be returned from functions. This allows programmers to combine smaller functions into larger functions increasing the code's re-usability. Note: Passing around built-in functions as such is not supported. Here is a contrived example showing the use of higher-order functions in the g(ROOT) programming language:

```
(val foo (lambda (n) (+ n 2)))  
(val bar (lambda (x) x))  
(printi ((bar foo) 6))
```

Passing polymorphic functions to other polymorphic functions that apply them leads to undefined behavior and may raise an error.

Not allowed:

```
(val foo (lambda (f a) (f a)))  
(val bar (lambda (x) x))  
(foo bar 6)
```

## 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. We developed a [protocol](#) for git that is included in the appendix . Generally, every feature was implemented 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 amongst group members depending on interest, with frequent collaboration with project mentor and other members as the need arose 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 past 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

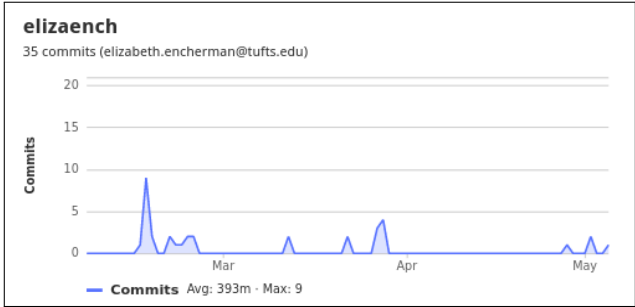
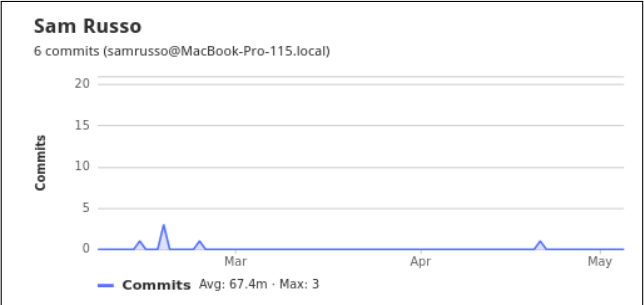
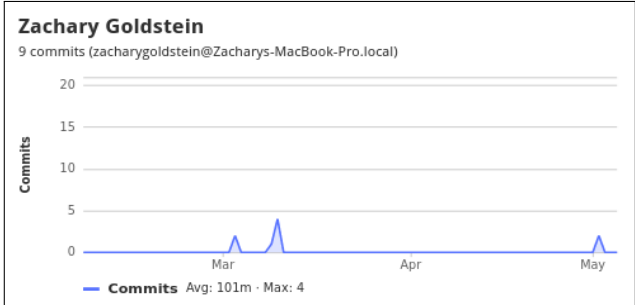
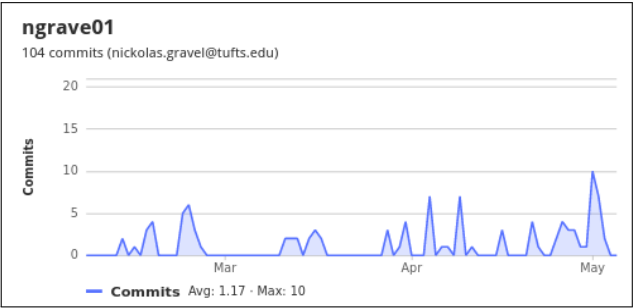
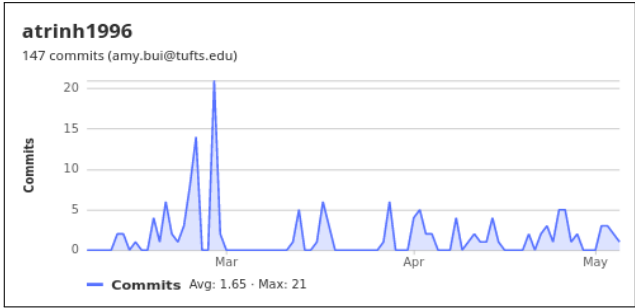
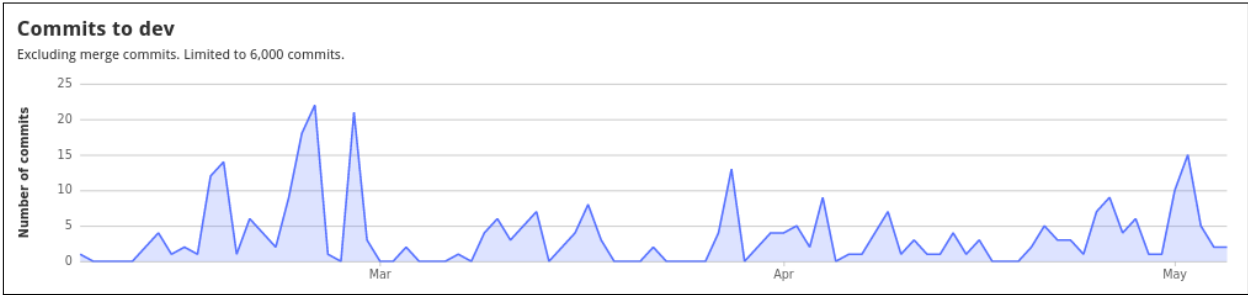
As the project progressed and implementing features of a functional language became more difficult, we abandoned these roles and each subgroup or member focused on progressively implementing different language features. Everyone was responsible initially testing their own feature and subsequent debugging, 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 function implementation across stages testing and debugging, deliverables
Sam Russo	Type Inferencer, testing
Zachary Goldstein	Errors & Warnings, test suite, Docker environment
Nickolas Gravel	Type Inferencer, test script
Amy Bui	Monomorphizer, Conversion I & II, Codegen, testing

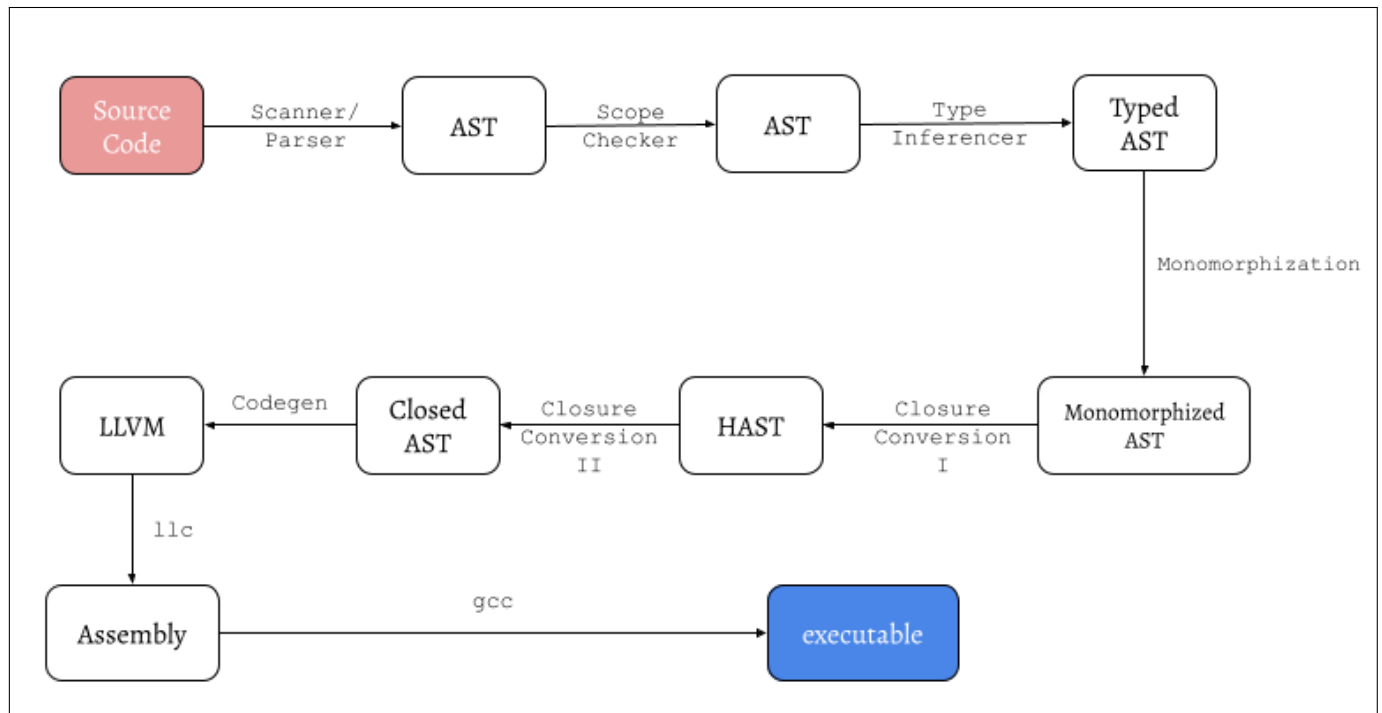
## 6.4 Development Tools

As mentioned previously, we used GitLab to manage version control and parallel development of features. Most work was done in Sublime, VSCode, or vim, and some group members used a [docker image](#) to avoid possible environment conflicts.

# 6.5 Project Logs



## 7 Architectural Design



### 7.1 Scanner/Parser

- Given the groot source code, the Scanner/Parser scans and tokenizes sequences of characters to build the basic abstract syntax tree of the program.
- *Authored by the whole group. Final syntax incorporated into Scanner/Parser done by Zach.*

### 7.2 Scope Checker

- This module performs a partial semantic check on a given abstract syntax tree, in which any variable name used has been bound and is used in the scope that it is allowed. The module reports an unbound variable, if any; otherwise, the same abstract syntax tree it receives is outputted.
- *Authored by Amy, with help incorporating primitives from Eliza.*

### 7.3 Type Inferencer

- The type inference takes an abstract syntax tree, and assumes all named variables are bound to some value. (g)ROOT is implicitly typed, so this module detects the types of all expressions and definitions. We follow the Hindley-Milner type system method in order to infer types. Using the types already given by primitives and literals, the inferencer generates constraints and then solves those constraints in order to deduce the type of the current definition it is inferring; those constraints are then used to infer the type of the rest of the program. The type inferencer will throw an error if it catches any type errors, such as type mismatches in function application of arguments. The module allows for polymorphism, so a definition may be typed to a type variable; however, to make resolving polymorphic types easier, this module also disallows nested lambdas. The inferencer outputs the final typed abstract syntax tree, or TAST.
- *Authored by Sam and Nik, with help from Mert. Primitives were incorporated by Eliza.*

### 7.4 Monomorphization

- This module takes a typed abstract syntax tree, which may or may not have any type variables, and monomorphizes it, getting rid of any polymorphism such that the tree has concrete types for everything. This is accomplished by tagging expressions that are polymorphic, finding where they are used (application), and using the types of the arguments it is used with to match type variables to concrete types; a mono-typed definition of the original polymorphic variable is then inserted into the program for that specific use case. The module outputs a monomorphic-typed abstract syntax tree.
- Bug: The algorithm we implemented for this resolves polymorphic types for non-nested lambda definitions, but cannot resolve them for nested lambdas which involves partial function application in order to get concrete types for each type variables. We concluded additional passes to resolve polymorphic types was required, but ran out of time for this. So, to ensure no type variables leak through to subsequent phases, Mono does one additional pass through the program, re-typing any leftover type variables to our int type. It is a recognized bug, so nested lambdas may result in undefined behavior or llc will raise an error regarding any type mismatches.
- *Authored by Amy.*

## 7.5 Closure Conversion I (Hof)

- Conversion I (or Hof) takes a monomorphic-typed abstract syntax tree, and creates partial closures for every lambda expression by re-typing every lambda expression from a function type to a new abstract type we called a closure type. Conversion I's closure types will carry a unique name to associate with the lambda expression (generated in the module), the original lambda's return type, the list of the formal types, AND a list of the types of the free variables. This is considered "partial" because no closure thus far is closed with the values or a reference to the values of the free variables.

Re-typing in this module helps subsequent modules deal with higher-order functions more easily, because Conversion I finds uses of a Hof, and re-types the function type to the appropriate closure type in function application

This module returns a h-abstract syntax tree. As indicated by the name, this intermediate phase makes handling higher-order functions easier in the next module.

- *Authored by Amy.*

## 7.6 Closure Conversion II (Conversion)

- Conversion II takes a h-abstract syntax tree, and finishes creating closures for every lambda expression. The new closure type in this module now carries the unique name previously described in Conversion I appended with a string to indicate it is referring to the type of the lambda, the lambda's function type augmented to include the types of the free variables appended to the list of the formal types, and a separate list of just the types of the free variables, if any. A closed lambda expression itself carries the original unique name, which will link it to its function definition later, and a list of the free variables converted into typed variable expressions. The closed lambda no longer carries the expression representing the body of the lambda, because this module creates a record type that represents a function definition for the lambda, which does carry the lambda's body expression and has a function name that matches the name of the lambda. These function definitions are set aside in a separate list to be passed to the next phase. Also being passed to the next phase is a list of all closure types created.

The other significant task this module has is to track the number of times a named variable is redefined; this allows closures to use the original value of the free variable it was closed with even after that variable gets redefined elsewhere.

The final output of this module is a struct containing the closed abstract syntax tree, list of function definitions, list of closure types generated during this phase, and a map of names to a list of their occurrences and types.

- *Authored by Amy, with help from Mert, and with help incorporating primitives from Eliza.*



## 7.7 Code Generation

- Code generation produces the llvm instruction for each definition and expression in the program. Each kind of definition and expression produce a particular pattern of llvm instructions. Given the information described in Closure Conversion II, code generation has four major steps to accomplish this:

1. Every “closure type” mentioned previously gets declared a named struct type in the llvm (using the available function type and free variable types); they all have the following common structure:

```
{ fptr* ; { typ ; } }
```

Where the first struct member is always a pointer to the function created as a lambda’s function definition, and all subsequent member(s), if any, are where the values of free variables will get stored.

2. Every named variable (including versions of it, should it have been defined multiple times) is globally defined and initialized to zero or nullptr depending on whether or not it is a function type, struct type, or some constant. When global variables are referred to in the program, it is either to assign them a value or use them. Because all definitions are evaluated in order in a main function (discussed next), no variable is used before a value is stored in them in the llvm.
  3. Every definition/expression in the closed ast gets turned into llvm instructions and put in a “main” function body. This allows us to mimic sequential execution.
  4. Every “function definition” mentioned previously is declared and defined by generating llvm instruction for the body’s expression.
- *Authored by Amy and Eliza.*

## 8 Test Plan

### 8.1 Test Method

Professor Richard Townsend’s testing script used in his MicroC compiler, `testall.sh`, was adapted to perform testing for the `g(ROOT)` compiler. Changes were made to add support for `.gt` files, and support the testing file directory tree we had decided upon, however, the general control flow and output remains the same.

As our compiler matured, the testing script also needed to mature. Based on the compiler’s current phase, different components needed to be tested to ensure they all were in working order. In addition to the testing script, many printing functions were written to assist in the debugging process. The most critical change to the testing script was the inclusion of test sets. Tests are grouped according to language aspect, and each test set is run individually. The testing script no longer executes a set of tests; now it processes a series of test sets, each containing a set of tests pertaining to that testset. This allows us to organize our tests logically and helps us localize bugs quickly. In our submission, the folder containing the test sets is simply called “testsets”. The test sets therein are listed below:

```
testsets
├── apply
├── arithmetic_binops
├── boolean_binops
├── comparison_binops
├── conditionals
├── inference
├── lambda
├── let
├── sanity_checks
├── semantics
├── types
├── unops
└── val
```

Each test set has several subdirectories. The “tests” subdirectory contains the `g(ROOT)` files comprising each test. The “ref\_ast”, “ref\_llvm”, “ref\_stderr”, and “ref\_stdout” subdirectories contain the reference output for each aspect of the compilation (ASTs, LLVM, errors, and output). When the testing script is run, additional subdirectories are generated, each containing some of the output from the tests. The “asm”, “ast”, “diff”, “exe”, “llvm”, “stderr”, and “stdout” subdirectories contain the assembly, ASTs, diff output, LLVM, errors, and output respectively. An example testset structure is given below.

We established various roles for each member of the group. Each role was made to divide key responsibilities that person was responsible for. though as we progressed through the development process these roles tended to be passed amongst other group members depending on various factors (i.e. availability) and simply what made the most practical sense. For instance, testing was a role that was shared across group members. The testing script was largely modified and updated by Nickolas Gravel and Zach Goldstein because they had the most experience working in bash than other members of the group. Furthermore, all members spent time creating tests and test references for modules.

```
example_testset
├── asm
├── ast
├── diff
├── exe
├── llvm
├── ref_ast
├── ref_llvm
├── ref_stderr
├── ref_stdout
├── stderr
├── stdout
└── tests
```

Note that there are three tests that still fail; these all pertain to the propagation of free variables in nested lambda expressions and nested let expressions. The current type inferencer implementation is not able to consistently resolve the type of free variables in nested expressions containing several levels of lets or lambdas.

let/test-let

lambda/test-lambda

inference/test-nlambda4

## 8.2 Three Test Samples

### 8.2.1 Sample 1: Nested If

```
1 (val a 'a')
2 (val b 'b')
3
4 (if (if (=i 0 0)
5     #t
6     #f)
7   (putc a)
8   (putc b))
```

Sample Test 1 Nested If-expression

LLVM IR output:

```
1 ; ModuleID = 'gROOT'
2 source_filename = "gROOT"
3
4 @fmt = private unnamed_addr constant [4 x i8] c"%d\0A\00", align 1
5 @boolT = private unnamed_addr constant [3 x i8] c"#t\00", align 1
6 @boolF = private unnamed_addr constant [3 x i8] c"#f\00", align 1
7 @_a_1 = global i8* null
8 @_b_1 = global i8* null
9 @globalChar = private unnamed_addr constant [2 x i8] c"a\00", align 1
10 @globalChar.1 = private unnamed_addr constant [2 x i8] c"b\00", align 1
11
12 declare i32 @printf(i8*, ...)
13
14 declare i32 @puts(i8*)
15
16 define i32 @main() {
17 entry:
18   %spc = alloca i8*, align 8
19   %loc = getelementptr i8*, i8** %spc, i32 0
20   store i8* getelementptr inbounds ([2 x i8], [2 x i8]* @globalChar, i32 0, i32 0), i8** %loc, align 8
21   %character_ptr = load i8*, i8** %spc, align 8
22   store i8* %character_ptr, i8** @_a_1, align 8
23   %spc1 = alloca i8*, align 8
24   %loc2 = getelementptr i8*, i8** %spc1, i32 0
25   store i8* getelementptr inbounds ([2 x i8], [2 x i8]* @globalChar.1, i32 0, i32 0), i8** %loc2, align 8
26   %character_ptr3 = load i8*, i8** %spc1, align 8
27   store i8* %character_ptr3, i8** @_b_1, align 8
28   %if-res-ptr = alloca i32, align 4
29   %if-res-ptr4 = alloca i1, align 1
30   br i1 true, label %then, label %else
```

```

31
32 merge:                                     ; preds = %else, %then
33     %if-res-val = load i1, i1* %if-res-ptr4, align 1
34     br i1 %if-res-val, label %then6, label %else7
35
36 then:                                     ; preds = %entry
37     store i1 true, i1* %if-res-ptr4, align 1
38     br label %merge
39
40 else:                                     ; preds = %entry
41     store i1 false, i1* %if-res-ptr4, align 1
42     br label %merge
43
44 merge5:                                   ; preds = %else7, %then6
45     %if-res-val9 = load i32, i32* %if-res-ptr, align 4
46     ret i32 0
47
48 then6:                                   ; preds = %merge
49     %_a_1 = load i8*, i8** @_a_1, align 8
50     %printc = call i32 @puts(i8* %_a_1)
51     store i32 %printc, i32* %if-res-ptr, align 4
52     br label %merge5
53
54 else7:                                   ; preds = %merge
55     %_b_1 = load i8*, i8** @_b_1, align 8
56     %printc8 = call i32 @puts(i8* %_b_1)
57     store i32 %printc8, i32* %if-res-ptr, align 4
58     br label %merge5
59 }

```

## 8.2.2 Sample 2: Closure

```
1 (val x 42)
2 (val retx (lambda (n) x))
3 (printi (retx 6))
```

Sample Test 2 Closure

LLVM IR output:

```
1 ; ModuleID = 'gROOT'
2 source_filename = "gROOT"
3
4 %_anon1_struct = type { i32 (i32, i32)*, i32 }
5 %_anon0_struct = type { i32 (i32, i32)*, i32 }
6
7 @fmt = private unnamed_addr constant [4 x i8] c"%d\0A\00", align 1
8 @boolT = private unnamed_addr constant [3 x i8] c"#t\00", align 1
9 @boolF = private unnamed_addr constant [3 x i8] c"#f\00", align 1
10 @_anon0_1 = global i32 (i32, i32)* null
11 @_anon1_1 = global i32 (i32, i32)* null
12 @retx_2 = global %_anon1_struct* null
13 @retx_1 = global %_anon0_struct* null
14 @_x_1 = global i32 0
15
16 declare i32 @printf(i8*, ...)
17
18 declare i32 @puts(i8*)
19
20 define i32 @main() {
21 entry:
22     store i32 42, i32* @_x_1, align 4
23     %gstruct = alloca %_anon0_struct, align 8
24     %funcField = getelementptr inbounds %_anon0_struct, %_anon0_struct* %gstruct, i32 0, i32 0
25     store i32 (i32, i32)* @_anon0, i32 (i32, i32)** %funcField, align 8
26     @_x_1 = load i32, i32* @_x_1, align 4
27     %freeField = getelementptr inbounds %_anon0_struct, %_anon0_struct* %gstruct, i32 0, i32 1
28     store i32 @_x_1, i32* %freeField, align 4
29     store %_anon0_struct* %gstruct, %_anon0_struct** @retx_1, align 8
30     %gstruct1 = alloca %_anon1_struct, align 8
31     %funcField2 = getelementptr inbounds %_anon1_struct, %_anon1_struct* %gstruct1, i32 0, i32 0
32     store i32 (i32, i32)* @_anon1, i32 (i32, i32)** %funcField2, align 8
33     @_x_13 = load i32, i32* @_x_1, align 4
34     %freeField4 = getelementptr inbounds %_anon1_struct, %_anon1_struct* %gstruct1, i32 0, i32 1
35     store i32 @_x_13, i32* %freeField4, align 4
36     store %_anon1_struct* %gstruct1, %_anon1_struct** @retx_2, align 8
37     @retx_2 = load %_anon1_struct*, %_anon1_struct** @retx_2, align 8
```

```

38 %freePtr = getelementptr inbounds @_anon1_struct, @_anon1_struct* %_retx_2, i32 0, i32 1
39 %freeVal = load i32, i32* %freePtr, align 4
40 %function_access = getelementptr inbounds @_anon1_struct, @_anon1_struct* %_retx_2, i32 0, i32 0
41 %function_call = load i32 (i32, i32)*, i32 (i32, i32)** %function_access, align 8
42 %function_result = call i32 @function_call(i32 6, i32 %freeVal)
43 %printi = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8], [4 x i8]* @fmt, i32 0, i32
44 ret i32 0
45 }
46
47 define i32 @anon1(i32 %n, i32 %_x_1) {
48 entry:
49 %n1 = alloca i32, align 4
50 store i32 %n, i32* %n1, align 4
51 %_x_12 = alloca i32, align 4
52 store i32 %_x_1, i32* %_x_12, align 4
53 %_x_13 = load i32, i32* %_x_12, align 4
54 ret i32 %_x_13
55 }
56
57 define i32 @anon0(i32 %n, i32 %_x_1) {
58 entry:
59 %n1 = alloca i32, align 4
60 store i32 %n, i32* %n1, align 4
61 %_x_12 = alloca i32, align 4
62 store i32 %_x_1, i32* %_x_12, align 4
63 %_x_13 = load i32, i32* %_x_12, align 4
64 ret i32 %_x_13
65 }

```

### 8.2.3 Sample 3: Higher-Order Function

```

1  (val add1 (lambda (x) (+ x 1)))
2
3  (val sub1 (lambda (x) (- x 1)))
4
5  (val retx (lambda (x) x))
6
7  (val callFunc (lambda (func arg) (func arg)))
8
9  (val retDoubleLambda (lambda () (lambda (x) (* x 2))))
10
11 (printi (callFunc add1 41)) → → → → → ( ; prints 42 ; )
12 (printi (callFunc sub1 43)) → → → → → ( ; prints 42 ; )
13 (printc (callFunc retx 'c')) → → → → → ( ; prints c ; )
14 (printi (callFunc (lambda (x) (* x x)) 2)) → → → → → ( ; prints 4 ; )
15 (printi (callFunc retx 42)) → → → → → ( ; prints 42 ; )
16 (printb (callFunc retx #t)) → → → → → ( ; prints #t ; )
17 (printi ((retDoubleLambda) 500)) → → → → → ( ; prints 1000 ; )
18 (printi ((retx add1) 5)) → → → → → ( ; prints 6 ; )

```

### Sample Test 3 demonstrating higher-order functions

LLVM IR output:

```

1 ; ModuleID = 'gROOT'
2 source_filename = "gROOT"
3
4 %_anon11_struct = type { i1 (i1)* }
5 %_anon0_struct = type { i32 (i32)* }
6 %_anon1_struct = type { i32 (i32)* }
7 %_anon7_struct = type { i8* (i8)* }
8 %_anon9_struct = type { i32 (i32)* }
9 %_anon10_struct = type { i1 (%_anon11_struct*, i1)* }
10 %_anon8_struct = type { i32 (%_anon9_struct*, i32)* }
11 %_anon6_struct = type { i8* (%_anon7_struct*, i8)* }
12 %_anon5_struct = type { i32 (%_anon1_struct*, i32)* }
13 %_anon4_struct = type { i32 (%_anon0_struct*, i32)* }
14 %_anon2_struct = type { i32 (i32)* }
15
16 @fmt = private unnamed_addr constant [4 x i8] c"%d\0A\00", align 1
17 @boolT = private unnamed_addr constant [3 x i8] c"#t\00", align 1
18 @boolF = private unnamed_addr constant [3 x i8] c"#f\00", align 1
19 @__anon0_1 = global i32 (i32)* null
20 @__anon1_1 = global i32 (i32)* null
21 @__anon10_1 = global i1 (%_anon11_struct*, i1)* null
22 @__anon11_1 = global i1 (i1)* null
23 @__anon2_1 = global i32 (i32)* null
24 @__anon4_1 = global i32 (%_anon0_struct*, i32)* null
25 @__anon5_1 = global i32 (%_anon1_struct*, i32)* null
26 @__anon6_1 = global i8* (%_anon7_struct*, i8)* null

```



```

27  __anon7_1 = global i8* (i8*)* null
28  __anon8_1 = global i32 (%_anon9_struct*, i32)* null
29  __anon9_1 = global i32 (i32)* null
30  _add1_1 = global %_anon0_struct* null
31  _callFunc_5 = global %_anon10_struct* null
32  _callFunc_4 = global %_anon8_struct* null
33  _callFunc_3 = global %_anon6_struct* null
34  _callFunc_2 = global %_anon5_struct* null
35  _callFunc_1 = global %_anon4_struct* null
36  _retx_4 = global %_anon11_struct* null
37  _retx_3 = global %_anon9_struct* null
38  _retx_2 = global %_anon7_struct* null
39  _retx_1 = global %_anon2_struct* null
40  _sub1_1 = global %_anon1_struct* null
41  @globalChar = private unnamed_addr constant [2 x i8] c"c\00", align 1
42
43  declare i32 @printf(i8*, ...)
44
45  declare i32 @puts(i8*)
46
47  define i32 @main() {
48  entry:
49      %gstruct = alloca %_anon0_struct, align 8
50      %funcField = getelementptr inbounds %_anon0_struct, %_anon0_struct* %gstruct, i32 0, i32 0
51      store i32 (i32)* @_anon0, i32 (i32)** %funcField, align 8
52      store %_anon0_struct* %gstruct, %_anon0_struct** @_add1_1, align 8
53      %gstruct1 = alloca %_anon1_struct, align 8
54      %funcField2 = getelementptr inbounds %_anon1_struct, %_anon1_struct* %gstruct1, i32 0, i32 0
55      store i32 (i32)* @_anon1, i32 (i32)** %funcField2, align 8
56      store %_anon1_struct* %gstruct1, %_anon1_struct** @_sub1_1, align 8
57      %gstruct3 = alloca %_anon2_struct, align 8
58      %funcField4 = getelementptr inbounds %_anon2_struct, %_anon2_struct* %gstruct3, i32 0, i32 0
59      store i32 (i32)* @_anon2, i32 (i32)** %funcField4, align 8
60      store %_anon2_struct* %gstruct3, %_anon2_struct** @_retx_1, align 8
61      %gstruct5 = alloca %_anon4_struct, align 8
62      %funcField6 = getelementptr inbounds %_anon4_struct, %_anon4_struct* %gstruct5, i32 0, i32 0
63      store i32 (%_anon0_struct*, i32)* @_anon4, i32 (%_anon0_struct*, i32)** %funcField6, align 8
64      store %_anon4_struct* %gstruct5, %_anon4_struct** @_callFunc_1, align 8
65      _callFunc_1 = load %_anon4_struct*, %_anon4_struct** @_callFunc_1, align 8
66      _add1_1 = load %_anon0_struct*, %_anon0_struct** @_add1_1, align 8
67      %function_access = getelementptr inbounds %_anon4_struct, %_anon4_struct* %_callFunc_1, i32 0, i32 0
68      %function_call = load i32 (%_anon0_struct*, i32)*, i32 (%_anon0_struct*, i32)** %function_access, align 8
69      %function_result = call i32 @function_call(%_anon0_struct* %_add1_1, i32 41)
70      %printi = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8], [4 x i8]* @fmt, i32 0, i32 0)
71      %gstruct7 = alloca %_anon5_struct, align 8
72      %funcField8 = getelementptr inbounds %_anon5_struct, %_anon5_struct* %gstruct7, i32 0, i32 0

```

```

73 store i32 (%_anon1_struct*, i32)* @_anon5, i32 (%_anon1_struct*, i32)** %funcField8, align 8
74 store %_anon5_struct* %gstruct7, %_anon5_struct** @_callFunc_2, align 8
75 %_callFunc_2 = load %_anon5_struct*, %_anon5_struct** @_callFunc_2, align 8
76 %_sub1_1 = load %_anon1_struct*, %_anon1_struct** @_sub1_1, align 8
77 %function_access9 = getelementptr inbounds %_anon5_struct, %_anon5_struct* %_callFunc_2, i32 0, i32 0
78 %function_call10 = load i32 (%_anon1_struct*, i32)*, i32 (%_anon1_struct*, i32)** %function_access9, align 8
79 %function_result11 = call i32 @function_call10(%_anon1_struct* %_sub1_1, i32 43)
80 %printi12 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8], [4 x i8]* @fmt, i32 0, i32 0), i8* %function_result11)
81 %gstruct13 = alloca %_anon7_struct, align 8
82 %funcField14 = getelementptr inbounds %_anon7_struct, %_anon7_struct* %gstruct13, i32 0, i32 0
83 store i8* (i8*)* @_anon7, i8* (i8*)** %funcField14, align 8
84 store %_anon7_struct* %gstruct13, %_anon7_struct** @_retx_2, align 8
85 %gstruct15 = alloca %_anon6_struct, align 8
86 %funcField16 = getelementptr inbounds %_anon6_struct, %_anon6_struct* %gstruct15, i32 0, i32 0
87 store i8* (%_anon7_struct*, i8*)* @_anon6, i8* (%_anon7_struct*, i8*)** %funcField16, align 8
88 store %_anon6_struct* %gstruct15, %_anon6_struct** @_callFunc_3, align 8
89 %_callFunc_3 = load %_anon6_struct*, %_anon6_struct** @_callFunc_3, align 8
90 %_retx_2 = load %_anon7_struct*, %_anon7_struct** @_retx_2, align 8
91 %spc = alloca i8*, align 8
92 %loc = getelementptr i8*, i8** %spc, i32 0
93 store i8* getelementptr inbounds ([2 x i8], [2 x i8]* @globalChar, i32 0, i32 0), i8** %loc, align 8
94 %character_ptr = load i8*, i8** %spc, align 8
95 %function_access17 = getelementptr inbounds %_anon6_struct, %_anon6_struct* %_callFunc_3, i32 0, i32 0
96 %function_call18 = load i8* (%_anon7_struct*, i8*)*, i8* (%_anon7_struct*, i8*)** %function_access17, align 8
97 %function_result19 = call i8* @function_call18(%_anon7_struct* %_retx_2, i8* %character_ptr)
98 %putc = call i32 @puts(i8* %function_result19)
99 %gstruct20 = alloca %_anon9_struct, align 8
100 %funcField21 = getelementptr inbounds %_anon9_struct, %_anon9_struct* %gstruct20, i32 0, i32 0
101 store i32 (i32)* @_anon9, i32 (i32)** %funcField21, align 8
102 store %_anon9_struct* %gstruct20, %_anon9_struct** @_retx_3, align 8
103 %gstruct22 = alloca %_anon8_struct, align 8
104 %funcField23 = getelementptr inbounds %_anon8_struct, %_anon8_struct* %gstruct22, i32 0, i32 0
105 store i32 (%_anon9_struct*, i32)* @_anon8, i32 (%_anon9_struct*, i32)** %funcField23, align 8
106 store %_anon8_struct* %gstruct22, %_anon8_struct** @_callFunc_4, align 8
107 %_callFunc_4 = load %_anon8_struct*, %_anon8_struct** @_callFunc_4, align 8
108 %_retx_3 = load %_anon9_struct*, %_anon9_struct** @_retx_3, align 8
109 %function_access24 = getelementptr inbounds %_anon8_struct, %_anon8_struct* %_callFunc_4, i32 0, i32 0
110 %function_call25 = load i32 (%_anon9_struct*, i32)*, i32 (%_anon9_struct*, i32)** %function_access24, align 8
111 %function_result26 = call i32 @function_call25(%_anon9_struct* %_retx_3, i32 42)
112 %printi27 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8], [4 x i8]* @fmt, i32 0, i32 0), i8* %function_result26)
113 %gstruct28 = alloca %_anon11_struct, align 8
114 %funcField29 = getelementptr inbounds %_anon11_struct, %_anon11_struct* %gstruct28, i32 0, i32 0
115 store i1 (i1)* @_anon11, i1 (i1)** %funcField29, align 8
116 store %_anon11_struct* %gstruct28, %_anon11_struct** @_retx_4, align 8
117 %gstruct30 = alloca %_anon10_struct, align 8
118 %funcField31 = getelementptr inbounds %_anon10_struct, %_anon10_struct* %gstruct30, i32 0, i32 0

```

```

119     store i1 (%_anon11_struct*, i1)* @_anon10, i1 (%_anon11_struct*, i1)** %funcField31, align 8
120     store %_anon10_struct* %gstruct30, %_anon10_struct** @_callFunc_5, align 8
121     %_callFunc_5 = load %_anon10_struct*, %_anon10_struct** @_callFunc_5, align 8
122     %retx_4 = load %_anon11_struct*, %_anon11_struct** @_retx_4, align 8
123     %function_access32 = getelementptr inbounds %_anon10_struct, %_anon10_struct* %_callFunc_5, i32 0, i32
124     %function_call33 = load i1 (%_anon11_struct*, i1)*, i1 (%_anon11_struct*, i1)** %function_access32, al
125     %function_result34 = call i1 %function_call33(%_anon11_struct* %retx_4, i1 true)
126     %printb = call i32 @puts(i8* getelementptr inbounds ([3 x i8], [3 x i8]* @boolF, i32 0, i32 0))
127     ret i32 0
128 }
129
130 define i1 @_anon10(%_anon11_struct* %func, i1 %arg) {
131 entry:
132     %func1 = alloca %_anon11_struct*, align 8
133     store %_anon11_struct* %func, %_anon11_struct** %func1, align 8
134     %arg2 = alloca i1, align 1
135     store i1 %arg, i1* %arg2, align 1
136     %func3 = load %_anon11_struct*, %_anon11_struct** %func1, align 8
137     %arg4 = load i1, i1* %arg2, align 1
138     %function_access = getelementptr inbounds %_anon11_struct, %_anon11_struct* %func3, i32 0, i32 0
139     %function_call = load i1 (i1)*, i1 (i1)** %function_access, align 8
140     %function_result = call i1 %function_call(i1 %arg4)
141     ret i1 %function_result
142 }
143
144 define i1 @_anon11(i1 %x) {
145 entry:
146     %x1 = alloca i1, align 1
147     store i1 %x, i1* %x1, align 1
148     %x2 = load i1, i1* %x1, align 1
149     ret i1 %x2
150 }
151
152 define i32 @_anon8(%_anon9_struct* %func, i32 %arg) {
153 entry:
154     %func1 = alloca %_anon9_struct*, align 8
155     store %_anon9_struct* %func, %_anon9_struct** %func1, align 8
156     %arg2 = alloca i32, align 4
157     store i32 %arg, i32* %arg2, align 4
158     %func3 = load %_anon9_struct*, %_anon9_struct** %func1, align 8
159     %arg4 = load i32, i32* %arg2, align 4
160     %function_access = getelementptr inbounds %_anon9_struct, %_anon9_struct* %func3, i32 0, i32 0
161     %function_call = load i32 (i32)*, i32 (i32)** %function_access, align 8
162     %function_result = call i32 %function_call(i32 %arg4)
163     ret i32 %function_result
164 }

```

```

165
166 define i32 @_anon9(i32 %x) {
167 entry:
168   %x1 = alloca i32, align 4
169   store i32 %x, i32* %x1, align 4
170   %x2 = load i32, i32* %x1, align 4
171   ret i32 %x2
172 }
173
174 define i8* @_anon6(%_anon7_struct* %func, i8* %arg) {
175 entry:
176   %func1 = alloca %_anon7_struct*, align 8
177   store %_anon7_struct* %func, %_anon7_struct** %func1, align 8
178   %arg2 = alloca i8*, align 8
179   store i8* %arg, i8** %arg2, align 8
180   %func3 = load %_anon7_struct*, %_anon7_struct** %func1, align 8
181   %arg4 = load i8*, i8** %arg2, align 8
182   %function_access = getelementptr inbounds %_anon7_struct, %_anon7_struct* %func3, i32 0, i32 0
183   %function_call = load i8* (i8*)*, i8* (i8**) %function_access, align 8
184   %function_result = call i8* %function_call(i8* %arg4)
185   ret i8* %function_result
186 }
187
188 define i8* @_anon7(i8* %x) {
189 entry:
190   %x1 = alloca i8*, align 8
191   store i8* %x, i8** %x1, align 8
192   %x2 = load i8*, i8** %x1, align 8
193   ret i8* %x2
194 }
195
196 define i32 @_anon5(%_anon1_struct* %func, i32 %arg) {
197 entry:
198   %func1 = alloca %_anon1_struct*, align 8
199   store %_anon1_struct* %func, %_anon1_struct** %func1, align 8
200   %arg2 = alloca i32, align 4
201   store i32 %arg, i32* %arg2, align 4
202   %func3 = load %_anon1_struct*, %_anon1_struct** %func1, align 8
203   %arg4 = load i32, i32* %arg2, align 4
204   %function_access = getelementptr inbounds %_anon1_struct, %_anon1_struct* %func3, i32 0, i32 0
205   %function_call = load i32 (i32)*, i32 (i32)** %function_access, align 8
206   %function_result = call i32 %function_call(i32 %arg4)
207   ret i32 %function_result
208 }
209
210 define i32 @_anon4(%_anon0_struct* %func, i32 %arg) {

```

```

211 entry:
212     %func1 = alloca %_anon0_struct*, align 8
213     store %_anon0_struct* %func, %_anon0_struct** %func1, align 8
214     %arg2 = alloca i32, align 4
215     store i32 %arg, i32* %arg2, align 4
216     %func3 = load %_anon0_struct*, %_anon0_struct** %func1, align 8
217     %arg4 = load i32, i32* %arg2, align 4
218     %function_access = getelementptr inbounds %_anon0_struct, %_anon0_struct* %func3, i32 0, i32 0
219     %function_call = load i32 (i32)*, i32 (i32)** %function_access, align 8
220     %function_result = call i32 @function_call(i32 %arg4)
221     ret i32 %function_result
222 }
223
224 define i32 @_anon2(i32 %x) {
225 entry:
226     %x1 = alloca i32, align 4
227     store i32 %x, i32* %x1, align 4
228     %x2 = load i32, i32* %x1, align 4
229     ret i32 %x2
230 }
231
232 define i32 @_anon1(i32 %x) {
233 entry:
234     %x1 = alloca i32, align 4
235     store i32 %x, i32* %x1, align 4
236     %x2 = load i32, i32* %x1, align 4
237     %subtraction = sub i32 %x2, 1
238     ret i32 %subtraction
239 }
240
241 define i32 @_anon0(i32 %x) {
242 entry:
243     %x1 = alloca i32, align 4
244     store i32 %x, i32* %x1, align 4
245     %x2 = load i32, i32* %x1, align 4
246     %addition = add i32 %x2, 1
247     ret i32 %addition
248 }

```

## 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 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

I learned a great deal about languages and compilers from this project - I definitely agree with Amy that everyone should write a compiler before graduating, and I'd recommend the language creation step as well, as I feel like I really understand the code I write and see better, as well as understand some of the thought processes behind some the peculiarities of different languages. I also whole-heartedly agree with my groupmates about the difficulties involved with implementing functional languages and type inferencing - this project very much reinforced how what is simpler for the user is often far more complicated to implement than something like static typing, that's a little more work on the user's side but much easier to check.

I also would emphasize the importance of good practices like documentation, throwing specific error statements, and making functions to print structures as you introduce structures - I frequently would be spending hours in debugging just finding where a specific error was coming from, which would then often be followed by hours more because it was hard to understand what a function did or what a variable was at a certain moment in time.

## 9.5 Zachary Goldstein

Upon reflection I have the following takeaways:

- It is critically important to ensure that all group members understand every aspect of the language design and how each module interfaces with other modules. Throughout the development of the compiler, different group members working on different modules made different assumptions about the structure of each augmented AST; this led to some frustrating refactoring when we brought all the pieces together.
- I benefit a great deal from synchronous, in-person work sessions. When working alone on the project, I found I used my time less effectively than when working in a shared space with other group members. Unfortunately, the group was not able to meet for more than an hour or two a week (which is to be expected for a group of 5 busy collegians), and nearly all of our meetings were virtual.

To future groups I give the following advice:

- When designing the language and deciding what features you will implement, prioritize feasibility. While type-inferencing is useful and interesting, our choice to implement an inferencer came at the expense of several other features that were ultimately more integral to our design goals.
- While functional languages are beautifully simple for programmers, they are not so simple to compile. Most of the functional features that we implemented/attempted (like higher-order functions, lambda closures, and inferred polymorphism) are not native to the imperative LLVM IR. It may be easier to design an imperative side-effecting language given the time constraints of the course. This is especially true if group members are unfamiliar with functional programming.
- Decide upon a consistent interface for each module.
- Work in person whenever possible. Spending time in a shared space increases the accountability of yourself and others, and allows for quick and easy idea sharing.



## 10 Reach Goals & Future Changes

As was stated before, the group did not get to accomplish some of the other interesting features mentioned in earlier iterations of the compiler. Here are some reach goals and future changes we'd might like to implement for the future for g(ROOT):

- Add character comparison (reference and structural), boolean comparison
- Add printing without a newline character
- Implement tree primitive data types (which would also allow a string representation as a unary char tree)
- Recursion recursion recursion recursion
- Introduce references/pointers

# 11 Appendix

## 11.1 toplevel.ml

Author(s): Sam, Eliza, Amy, Nik, Zach

```
1  (* Top-level of the groot compiler: scan & parse input,
2     build the AST, generate LLVM IR *)
3
4  type action =
5    | Ast
6    | Name_Check
7    | Tast
8    | Mast
9    | Hast
10   | Cast
11   | LLVM_IR
12   | Compile
13
14
15  let () =
16    let action = ref Ast in
17    let set_action a () = action := a in
18    let speclist = [
19      ("-a", Arg.Unit (set_action Ast),          "Print the AST (default)");
20      ("-n", Arg.Unit (set_action Name_Check),    "Print the AST (name-checking)");
21      ("-t", Arg.Unit (set_action Tast),          "Print the TAST");
22      ("-m", Arg.Unit (set_action Mast),          "Print the MAST");
23      ("-h", Arg.Unit (set_action Hast),          "Print the HAST");
24      ("-v", Arg.Unit (set_action Cast),          "Print the CAST");
25      ("-l", Arg.Unit (set_action LLVM_IR),       "Print the generated LLVM IR");
26      ("-c", Arg.Unit (set_action Compile),
27        "Check and print the generated LLVM IR");
28    ] in
29
30
31  let usage_msg =
32    "usage: ./toplevel.native [-a|-n|-t|-m|-h|-v|-l|-c] [file.gt]" in
33  let channel = ref stdin in
34  Arg.parse speclist (fun filename -> channel := open_in filename) usage_msg;
35  let lexbuf = Lexing.from_channel !channel in
36  let ast = Parser.prog Scanner.tokenize lexbuf in
37  match !action with
38  (* Default action - print the AST using ast *)
39  | Ast -> print_string (Ast.string_of_prog ast)
40  (* All other action needs to generate an SAST, store in variable sast *)
41  | _ ->
```

```

42 let ast' = Scope.check ast in
43 match !action with
44 | Ast -> ()
45 | Name_Check -> print_string (Ast.string_of_prog ast')
46 | _ ->
47   let tast = Infer.type_infer ast' in
48   match !action with
49   | Tast -> print_string (Tast.string_of_tprog tast)
50   | _ ->
51     let mast = Mono.monomorphize tast in
52     match !action with
53     | Mast -> print_string (Mast.string_of_mprog mast)
54     | _ ->
55       let hast = Hof.clean mast in
56       match !action with
57       | Hast -> print_string (Hast.string_of_hprog hast)
58       | _ ->
59         let cast = Conversion.conversion hast in
60         match !action with
61         | Cast -> print_string (Cast.string_of_cprog cast)
62         | LLVM_IR ->
63           print_string (Llvm.string_of_llmodule (Codegen.translate cast))
64         | Compile ->
65           let the_module = Codegen.translate cast in
66           Llvm_analysis.assert_valid_module the_module;
67           print_string (Llvm.string_of_llmodule the_module)
68         | _ -> print_string usage_msg

```

## 11.2 ast.ml

Author(s): Sam, Eliza, Amy, Nik, Zach

```
1  (* Abstract Syntax Tree (AST) for Groot *)
2
3  (* Type of Variable Names *)
4  type ident = string
5
6  type expr =
7    | Literal of value
8    | Var      of ident
9    | If       of expr * expr * expr
10   | Apply    of expr * expr list
11   | Let      of (ident * expr) list * expr
12   | Lambda   of ident list * expr
13 and value =
14   | Char     of char
15   | Int      of int
16   | Bool     of bool
17   | Root     of tree
18 and tree =
19   | Leaf
20   | Branch of expr * tree * tree
21
22 type defn =
23   | Val of ident * expr
24   | Expr of expr
25
26 type prog = defn list
27
28
29 (* Pretty printing functions *)
30
31 (* toString for Ast.expr *)
32 let rec string_of_expr = function
33   | Literal(lit) -> string_of_value lit
34   | Var(v) -> v
35   | If(condition, true_branch, false_branch) ->
36     "(if " ^ string_of_expr condition ^ " "
37     ^ string_of_expr true_branch ^ " "
38     ^ string_of_expr false_branch ^ ")"
39   | Apply(f, args) ->
40     "(" ^ string_of_expr f ^ " "
41     ^ String.concat " " (List.map string_of_expr args) ^ ")"
42   | Let(binds, body) ->
43     let string_of_binding = function
```

```

44     (id, e) -> "[" ^ id ^ " " ^ (string_of_expr e) ^ "]"
45   in
46     "(let (" ^ String.concat " " (List.map string_of_binding binds) ^ ") "
47     ^ string_of_expr body ^ ")"
48 | Lambda(formals, body) ->
49     "(lambda (" ^ String.concat " " formals ^ ") "
50     ^ string_of_expr body ^ ")"
51
52 (* toString for Ast.value *)
53 and string_of_value = function
54 | Char(c)      -> "'" ^ String.make 1 c ^ "'"
55 | Int(i)       -> string_of_int i
56 | Bool(b)      -> if b then "#t" else "#f"
57 | Root(tr)     -> string_of_tree tr
58 (* Closure(a,b,c) -> "CLOSURE: string_of_closure unimplemented" *)
59 (* Primitive(p, vals) -> "PRIMITIVE: string_of_primitive unimplemented" *)
60
61 (* toString for Ast.tree *)
62 and string_of_tree = function
63 | Leaf -> "leaf"
64 | Branch(ex, sib, child) ->
65     (* Branch type is given by "tree" string *)
66     "(tree " ^ string_of_expr ex ^ " "
67     ^ string_of_tree sib ^ " "
68     ^ string_of_tree child ^ ")"
69
70 (* toString for Ast.defn *)
71 let string_of_defn = function
72 | Val(id, e) -> "(val " ^ id ^ " " ^ string_of_expr e ^ ")"
73 | Expr(e)    -> string_of_expr e
74
75 (* toString for Ast.prog *)
76 let string_of_prog defs =
77     String.concat "\n" (List.map string_of_defn defs) ^ "\n"

```

## 11.3 scanner.mll

Author(s): Sam, Eliza, Amy, Nik, Zach

```
1  (*
2      scanner.mll
3
4      Lexer file to create a lexical analyzer from a set of regular expressions.
5
6      Compile with command to produce scanner.ml with the ocaml code:
7          ocamllex scanner.mll
8  *)
9
10 (* Header *)
11 {
12     open Parser
13 }
14
15 (* Regular Expressions *)
16 let digit = ['0'-'9']
17 let integer = ['-']?['0'-'9']+
18 let alpha = ['a'-'z']
19 let leaf = ("leaf"|"()")
20 let chrcode = digit+
21
22 (* all visible characters, excluding ()'[]\;{}| *)
23 let ident = ['!'-'&' '*'-';' '<'-'Z' '^'-'z' '~' ' ' '|']
24             ['!'-'&' '*'-';' '<'-'Z' '^'-'z' '~' ' ' '|']*
25
26
27 (* ToKeNiZe *)
28 rule tokenize = parse
29   | [' ' '\n' '\t' '\r'] { tokenize lexbuf }
30   | ";;" { single_comment lexbuf }
31   | "(;" { multi_comment lexbuf }
32   | '(' { LPAREN }
33   | ')' { RPAREN }
34   | '[' { LSQUARE }
35   | ']' { RSQUARE }
36   | "tree" { BRANCH }
37   | "leaf" { LEAF }
38   | "if" { IF }
39   | "'" { apos_handler lexbuf }
40   | integer as ival { INT(int_of_string ival) }
41   | "#t" { BOOL(true) }
42   | "#f" { BOOL(false) }
43   | "lambda" { LAMBDA }
```

```

44 | "let"           { LET }
45 | "val"           { VAL }
46 | ident as id     { ID(id) }
47 | eof             { EOF }
48 | _              { Diagnostic.error(Diagnostic.lex_error "unrecognized character" lexbuf) }
49 and single_comment = parse
50 | '\n'           { tokenize lexbuf }
51 | eof            { EOF }
52 | _             { single_comment lexbuf }
53 and multi_comment = parse
54 | ";)"           { tokenize lexbuf }
55 | eof            { EOF }
56 | _             { multi_comment lexbuf }
57
58 (* apostrophe handler *)
59 and apos_handler = parse
60 | '(' [^''']      { tree_builder lexbuf }
61 | '''            { Diagnostic.error (Diagnostic.lex_error "empty character literal" lexbuf) }
62 | '\\\          { escaped_char_handler lexbuf }
63 | _ as c         { char_builder c lexbuf }
64
65 and tree_builder = parse
66 | _ { Diagnostic.error (Diagnostic.Unimplemented "in-place tree syntax") }
67
68 and char_builder c = parse
69 | '''           { CHAR(c) }
70 | _ { Diagnostic.error
71     (Diagnostic.lex_error ("character literal contains more "
72     ^ "than one token") lexbuf) }
73
74 and escaped_char_handler = parse
75 | '\\\ { char_builder '\\\ lexbuf }
76 | '\"' { char_builder '\"' lexbuf }
77 | '\'' { char_builder '\'' lexbuf }
78 | '\n' { char_builder '\n' lexbuf }
79 | '\r' { char_builder '\r' lexbuf }
80 | '\t' { char_builder '\t' lexbuf }
81 | '\b' { char_builder '\b' lexbuf }
82 | '\ ' { char_builder '\ ' lexbuf }
83 | chrcode as ord
84     { print_string ord; if int_of_string ord > 255
85       then Diagnostic.error
86         (Diagnostic.lex_error "invalid escape sequence ASCII value"
87         lexbuf)
88       else char_builder (Char.chr (int_of_string ord)) lexbuf }
89 | _ { Diagnostic.error

```

```
(Diagnostic.lex_error "unrecognized escape sequence" lexbuf) }
```

---



## 11.4 parser.mly

Author(s): Sam, Eliza, Amy, Nik, Zach

```
1  /*
2   * parser.mly
3   *   produces a parser from a context-free grammar specification
4   *
5   *   Compile with command to produce parser.ml with the ocaml code:
6   *       ocaml yacc parser.mly
7   */
8
9  /* Header */
10 %{{
11     open Ast
12 %}}
13
14 /* Tokens */
15 %token LPAREN RPAREN
16 %token LSQUARE RSQUARE
17 %token PLUS MINUS TIMES DIVIDE MOD
18 %token EQ NEQ LT GT LEQ GEQ AND OR NOT
19 %token IF
20 %token <char> CHAR
21 %token <int> INT
22 %token <bool> BOOL
23 %token <string> ID
24 %token BRANCH LEAF
25 %token EOF
26 %token LAMBDA LET VAL
27
28 /* Precedence */
29 %nonassoc OR
30 %nonassoc AND
31 %nonassoc LT GT
32 %nonassoc EQ NEQ
33 %nonassoc LEQ GEQ
34 %nonassoc PLUS MINUS
35 %nonassoc TIMES DIVIDE
36 %nonassoc NEG
37 %nonassoc NOT
38 %nonassoc BRANCH LEAF
39
40
41 /* Declarations */
42 %start prog
43 %type <Ast.prog> prog
```

```

44
45 %%
46
47 prog:
48   | defn_list EOF                { $1 }
49
50 defn_list:
51   | /* nothing */                { [] }
52   | defn defn_list               { $1 :: $2 }
53
54 defn:
55   | expr                        { Expr($1) }
56   | LPAREN VAL ID expr RPAREN  { Val($3, $4) }
57
58 formals_opt:
59   | /* nothing */                { [] }
60   | formal_list                 { $1 }
61
62 formal_list:
63   | ID                          { [$1] }
64   | ID formal_list              { $1 :: $2 }
65
66
67 /* Rules */
68 value:
69   | CHAR                        { Char($1) }
70   | INT                         { Int($1) }
71   | BOOL                        { Bool($1) }
72   | tree                        { Root($1) }
73   /* ! Note: tree is not a token - no need for a ROOT token while scanning */
74
75
76 tree:
77   | LEAF                        { Leaf }
78   | LPAREN BRANCH expr tree tree RPAREN { Branch($3, $4, $5) }
79
80
81 let_binding_list:
82   | /* nothing */                { [] }
83   | LSQUARE RSQUARE let_binding_list
84     { Diagnostic.warning (Diagnostic.parse_warning "empty let binding" 1); $3 }
85   /* NON FATAL */
86   | LSQUARE expr RSQUARE let_binding_list
87     { Diagnostic.error (Diagnostic.parse_error ("let binding must contain"
88       ~ " id and value") 2) } /* FATAL */
89   | LSQUARE ID expr RSQUARE let_binding_list { ($2, $3) :: $5 }

```

```

90
91
92 expr_list:
93   | /* null */           { [] }
94   | expr expr_list       { $1 :: $2 }
95
96
97 expr:
98   | value                 { Literal($1) }
99   | ID                   { Var($1) }
100  | LPAREN expr expr_list RPAREN { Apply($2, $3) }
101  | LPAREN LET LPAREN let_binding_list RPAREN expr RPAREN { Let($4, $6)}
102  | LPAREN IF expr expr expr RPAREN { If($3, $4, $5) }
103  | LPAREN LAMBDA LPAREN formals_opt RPAREN expr RPAREN { Lambda($4, $6) }

```

---

## 11.5 scope.ml

Author(s): Amy, Eliza

```
1  (* Name (scope) checks variable names *)
2  open Ast
3
4  (* toplevel naming environment, preloaded with built-ins *)
5  let nameEnv = List.fold_right List.cons [ "printi"; "printb"; "printc";
6                                             "+"; "-"; "*"; "/"; "mod";
7                                             "<"; ">"; ">="; "<="; "~";
8                                             "!=i"; "=i"; "&&"; "||"; "not" ] []
9
10  (* Takes an AST and checks if variables are bound in scope.
11     Returns same AST if so, otherwise raises Unbound variable error
12     if a variable is unbound. *)
13  let check defns =
14
15    (* Recursively checks the scope of variables names used in an expression *)
16    let rec checkExpr expression rho =
17      let rec exp e = match e with
18        | Literal _ -> ()
19        | Var id ->
20          if List.mem id rho then ()
21          else Diagnostic.error (Diagnostic.Unbound id)
22        | If (e1, e2, e3) ->
23          let (_, _, _) = (exp e1, exp e2, exp e3) in ()
24        | Apply (f, args) ->
25          let _ = exp f in List.iter exp args
26        | Let (bs, body) ->
27          let (xs, es) = List.split bs in
28          let () = List.iter exp es in
29          checkExpr body (List.fold_right List.cons xs rho)
30        | Lambda (formals, body) ->
31          checkExpr body (List.fold_right List.cons formals rho)
32      in exp expression
33    in
34
35    let rec checkDef ds env =
36      match ds with
37      | [] -> env
38      | f :: rest ->
39        let env' =
40          (match f with
41           | Val (id, exp) ->
42             let () = checkExpr exp env in
43             id :: env
```

```
44 | Expr exp -> let () = checkExpr exp env in env)
45   in checkDef rest env'
46   in
47   let _ = checkDef defs nameEnv in
48
49   (* Returns the AST if no error raised *)
50   defs
```

---

## 11.6 tast.ml

Author(s): Sam, Nik

```
1  (*
2      TAST
3      Type inference.
4  *)
5
6  open Ast
7
8  exception Type_error of string
9
10 let type_error msg = raise (Type_error msg)
11
12
13 type gtype =
14   | TYCON of tycon
15   | TYVAR of tyvar
16   | CONAPP of conapp
17 and tycon =
18   | TyInt
19   | TyBool
20   | TyChar
21   | TArrow of gtype
22 and tyvar =
23   | TVariable of int
24 and conapp = (tycon * gtype list)
25
26 type tyscheme = tyvar list * gtype
27
28
29
30 let inttype = TYCON TyInt
31 let chartype = TYCON TyChar
32 let booltype = TYCON TyBool
33 let functiontype resultType formalsTypes =
34   CONAPP (TArrow resultType, formalsTypes)
35
36
37 (* TAST expression *)
38 type texpr = gtype * tx
39 and tx =
40   | TLiteral      of tvalue
41   | TypedVar      of ident
42   | TypedIf       of texpr * texpr * texpr
43   | TypedApply    of texpr * texpr list
```

```

44 | TypedLet      of (ident * texpr) list * texpr
45 | TypedLambda of (gtype * ident) list * texpr
46 and tvalue = TChar of char | TInt of int | TBool of bool | TRoot of ttree
47 and ttree = TLeaf | TBranch of tvalue * ttree * ttree
48
49 type tdefn = TVal of ident * texpr | TExpr of texpr
50 type tprog = tdefn list
51
52 (* Pretty printer *)
53
54 (* String of gtypes *)
55 let rec string_of_ttype = function
56 | TYCON ty -> string_of_tycon ty
57 | TYVAR tp -> string_of_tyvar tp
58 | CONAPP con -> string_of_conapp con
59 and string_of_tycon = function
60 | TyInt -> "int"
61 | TyBool -> "bool"
62 | TyChar -> "char"
63 | TArrow (retty) -> string_of_ttype retty
64
65 and string_of_tyvar = function
66 | TVariable n -> "'" ^ string_of_int n
67 and string_of_conapp (tyc, tys) =
68   string_of_tycon tyc ^ " (" ^ String.concat " " (List.map string_of_ttype tys) ^ ")"
69 and string_of_constraint (t1, t2) = "(" ^ string_of_ttype t1 ^ ", " ^ string_of_ttype t2 ^ ")"
70 and string_of_constraints cs =
71   "[" ^ String.concat " , " (List.map string_of_constraint cs) ^ "]"
72
73 and string_of_subs = function
74 | [] -> ""
75 | (t1, t2) :: cs ->
76   "(" ^ string_of_tyvar t1 ^ ", " ^ string_of_ttype t2 ^ ") "
77   ^ string_of_subs cs
78
79 and string_of_context = function
80 | [] -> ""
81 | (ident, (tvs, gt)) :: ctx ->
82   "\n=: " ^ ident ^ ", ("
83   ^ String.concat " , " (List.map string_of_tyvar tvs)
84   ^ "], " ^ string_of_ttype gt ^ ")" ^ string_of_context ctx
85
86 and string_of_tyformals (gt, ident) =
87   "(" ^ ident ^ " : " ^ string_of_ttype gt ^ ")"
88
89 (* String of a typed expression (texpr) == (type, t-expression) *)

```

```

90 let rec string_of_texpr (typ, exp) =
91   "[" ^ string_of_ttype typ ^ "]" ^ string_of_tx exp
92 and string_of_tx = function
93   TLiteral v -> string_of_tvalue v
94 | TypedVar id -> id
95 | TypedIf (te1, te2, te3) ->
96   "(if " ^ string_of_texpr te1 ^ " "
97   ^ string_of_texpr te2 ^ " "
98   ^ string_of_texpr te3 ^ ")"
99 | TypedApply (f, args) ->
100  "(" ^ string_of_texpr f ^ " "
101  ^ String.concat " " (List.map string_of_texpr args) ^ ")"
102 | TypedLet (binds, body) ->
103   let string_of_binding (id, e) =
104     "[" ^ id ^ " " ^ (string_of_texpr e) ^ "]"
105   in
106   "(let (" ^ String.concat " " (List.map string_of_binding binds) ^ ") "
107   ^ string_of_texpr body ^ ")"
108 | TypedLambda (formals, body) ->
109   let formalStringlist = List.map (fun (ty, x) ->
110     string_of_ttype ty ^ " " ^ x) formals in
111   "(lambda (" ^ String.concat ", " formalStringlist
112   ^ ") " ^ string_of_texpr body ^ ")"
113
114 (* toString for Sast.svalue *)
115 and string_of_tvalue = function
116   TChar c -> String.make 1 c
117 | TInt i -> string_of_int i
118 | TBool b -> if b then "#t" else "#f"
119 | TRoot tr -> string_of_ttree tr
120
121 (* toString for Sast.stree *)
122 and string_of_ttree = function
123   TLeaf -> "leaf"
124 | TBranch (v, sib, child) ->
125   "(tree " ^ string_of_tvalue v ^ " "
126   ^ string_of_ttree sib ^ " "
127   ^ string_of_ttree child ^ ")"
128
129 (* String of a typed defn (tdefn) *)
130 let string_of_tdefn = function
131   TVal (id, te) -> "(val " ^ id ^ " " ^ string_of_texpr te ^ ")"
132 | TExpr te -> string_of_texpr te
133
134
135

```



```
136  (* String of the tprog == tdefn list *)  
137  let string_of_tprog tdefs =  
138    String.concat "\n" (List.map string_of_tdefn tdefs) ^ "\n"
```

---

## 11.7 infer.ml

Author(s): Sam, Eliza, Nik

```
1 open Ast
2 open Tast
3 module StringMap = Map.Make (String)
4
5 (* prims - initializes context with built-in functions with their types *)
6 (* prims : (id * (tyvar list * gtype)) *)
7 let prims =
8   [
9     ("printb", ([ TVariable (-1) ], Tast.functiontype inttype [ booltype ]));
10    ("printi", ([ TVariable (-2) ], Tast.functiontype inttype [ inttype ]));
11    ("printc", ([ TVariable (-3) ], Tast.functiontype inttype [ chartype ]));
12    ("+", ([ TVariable (-4) ], Tast.functiontype inttype [ inttype; inttype ]));
13    ("-", ([ TVariable (-4) ], Tast.functiontype inttype [ inttype; inttype ]));
14    ("/", ([ TVariable (-4) ], Tast.functiontype inttype [ inttype; inttype ]));
15    ("*", ([ TVariable (-4) ], Tast.functiontype inttype [ inttype; inttype ]));
16    ("mod", ([ TVariable (-4) ], Tast.functiontype inttype [ inttype; inttype ]));
17   ("<", ([ TVariable (-5) ], Tast.functiontype booltype [ inttype; inttype ]));
18   (">", ([ TVariable (-5) ], Tast.functiontype booltype [ inttype; inttype ]));
19   ("<=", ([ TVariable (-5) ], Tast.functiontype booltype [ inttype; inttype ]));
20   (">=", ([ TVariable (-5) ], Tast.functiontype booltype [ inttype; inttype ]));
21    ("=i", ([ TVariable (-5) ], Tast.functiontype booltype [ inttype; inttype ]));
22    ("!=i", ([ TVariable (-5) ], Tast.functiontype booltype [ inttype; inttype ]));
23    ("&&", ([ TVariable (-6) ], Tast.functiontype booltype [ booltype; booltype ]));
24    ("||", ([ TVariable (-6) ], Tast.functiontype booltype [ booltype; booltype ]));
25    ("not", ([ TVariable (-7) ], Tast.functiontype booltype [ booltype ]));
26    ("~", ([ TVariable (-2) ], Tast.functiontype inttype [ inttype ]));
27   ]
28
29 (* is_ftv - returns true if 'gt' is equal to free type variable 'var'
30    (i.e. 'gt' is a type variable and 'var' is a free type variable). For the
31    conapp case, we recurse over the conapp's gtype list searching for any free
32    type variables. When this function returns true it means the type variable
33    is matching *)
34 let rec is_ftv (var : tyvar) (gt : gtype) =
35   match gt with
36   | TYCON _ -> false
37   | TYVAR v -> v = var
38   | CONAPP (_, gtlst) ->
39     (* if any x in gtlst is ftv this returns true, else returns false *)
40     List.fold_left (fun acc x -> is_ftv var x || acc) false gtlst
41
42 (* ftvs - returns a list of free type variables amongst a collection of gtypes *)
43 (* retty : tyvar list *)
```

```

44 let rec ftvs (ty : gtype) =
45   match ty with
46   | TYVAR t -> [ t ]
47   | TYCON _ -> []
48   | CONAPP (_, gtlst) -> List.fold_left (fun acc x -> acc @ ftvs x) [] gtlst
49
50   (* fresh - returns a fresh gtype variable (integer) *)
51 let fresh =
52   let k = ref 0 in
53   (* fun () -> incr k; TVariable !k *)
54   fun () -> incr k; TYVAR (TVariable !k)
55
56
57   (* gimme_tycon_gtype - sort of a hack function that we made to solve the bug we
58   came across in applying substitutions, called in tysubst *)
59   let gimme_tycon_gtype _ = function
60   | TYCON x -> x
61   | TYVAR x ->
62     Diagnostic.error (Diagnostic.TypeError ("the variable " ^ string_of_tyvar x
63       ^ " has type tyvar but an expression was expected of type tycon"))
64   | CONAPP x ->
65     Diagnostic.error (Diagnostic.TypeError ("the constructor " ^ string_of_conapp x
66       ^ " has type conapp but an expression was expected of type tycon"))
67
68
69   (* tysubst - subs in the type in place of type variable *)
70 let rec tysubst (one_sub : tyvar * gtype) (t : gtype) =
71   match (one_sub, t) with
72   | (x, y), TYVAR z -> if x = z then y else TYVAR z
73   | _, TYCON (TArrow retty) -> TYCON (TArrow (tysubst one_sub retty))
74   | _, TYCON c -> TYCON c
75   | (x, _), CONAPP (a, bs) ->
76     let tycon = gimme_tycon_gtype x in
77     CONAPP (tycon (tysubst one_sub (TYCON a)), (List.map (tysubst one_sub) bs))
78
79
80   (* sub - updates a list of constraints with substitutions in theta *)
81 let sub (theta : (tyvar * gtype) list) (cns : (gtype * gtype) list) =
82   (* sub_one - takes in single constraint and updates it with substitution in theta *)
83   let sub_one (cn : gtype * gtype) =
84     List.fold_left
85       (fun ((c1, c2) : gtype * gtype) (sub : tyvar * gtype) ->
86         (tysubst sub c1, tysubst sub c2))
87       cn theta
88   in
89   List.map sub_one cns

```

```

90
91 (* compose - applies the substitutions in theta1 to theta2 *)
92 let compose theta1 theta2 =
93   (* sub_one - takes a single substitution in theta1 and applies it to theta2 *)
94   let sub_one cn =
95     List.fold_left
96       (fun (acc : tyvar * gtype) (one_sub : tyvar * gtype) ->
97         match (acc, one_sub) with
98         | (a1, TYVAR a2), (s1, TYVAR s2) ->
99           if s1 = a1 then (s1, TYVAR a2)
100           else if s1 = a2 then (a1, TYVAR s2)
101           else acc
102         | (a1, a2), (s1, TYVAR _) -> if a1 = s1 then (s1, a2) else acc
103         | (a1, _), (s1, s2) -> if a1 = s1 then (s1, s2) else acc)
104     cn theta1
105   in
106   List.map sub_one theta2
107
108 (* solve': solves a single constraint 'c' *)
109 let rec solve' (c : gtype * gtype) =
110   match c with
111   | TYVAR t1, TYVAR t2 -> [ (t1, TYVAR t2) ]
112   | TYVAR t1, TYCON t2 -> [ (t1, TYCON t2) ]
113   | TYVAR t1, CONAPP t2 ->
114     if is_ftv t1 (CONAPP t2) then
115       Diagnostic.error (Diagnostic.TypeError "type variable is not free type in type constructor")
116     else [ (t1, CONAPP t2) ]
117   | TYCON t1, TYVAR t2 -> solve' (TYVAR t2, TYCON t1)
118   | TYCON (TArrow (TYVAR t1)), TYCON t2 -> [ (t1, TYCON t2) ]
119   | TYCON t1, TYCON (TArrow (TYVAR t2)) -> [ (t2, TYCON t1) ]
120   | TYCON t1, TYCON t2 ->
121     if t1 = t2 then []
122     else
123       Diagnostic.error (Diagnostic.TypeError
124         ("type constructor mismatch " ^ string_of_tycon t1
125           ^ " != " ^ string_of_tycon t2))
126   | TYCON t1, CONAPP t2 ->
127     Diagnostic.error (Diagnostic.TypeError
128       ("type constructor mismatch " ^ string_of_tycon t1
129         ^ " != " ^ string_of_conapp t2))
130   | CONAPP t1, TYVAR t2 -> solve' (TYVAR t2, CONAPP t1)
131   | CONAPP t1, TYCON t2 ->
132     Diagnostic.error (Diagnostic.TypeError
133       ("type constructor mismatch " ^ string_of_conapp t1
134         ^ " != " ^ string_of_tycon t2))
135   | CONAPP t1, CONAPP t2 -> (

```

```

136     match (t1, t2) with
137     | (TArrow t1, tys1), (TArrow t2, tys2) ->
138         solve ((t1, t2) :: List.combine tys1 tys2)
139     | _ ->
140         Diagnostic.error (Diagnostic.TypeError
141             ("type constructor mismatch " ^ string_of_conapp t1
142              ^ " != " ^ string_of_conapp t2)))
143
144
145     (* solve - solves a list of constraints, calls 'solver' to iterate through the
146        constraint list, once constraint list has been iterated 'compose' is
147        called to tie 'theta1' and 'theta2' together, returns theta *)
148     and solve (constraints : (gtype * gtype) list) =
149         let solver cns =
150             match cns with
151             | [] -> []
152             | cn :: cns ->
153                 let theta1 = solve' cn in
154                 let theta2 = solve (sub theta1 cns) in
155                 (compose theta2 theta1) @ theta2
156         in solver constraints
157
158
159     (* generate_constraints gctx e:
160        infers the type of expression 'e' and a generates a set of constraints,
161        'gctx' refers to the global context 'e' can refer to.
162
163        Type References:
164        ctx : (ident * tyscheme) list == (ident * (tyvar list * gtype)) list
165        tyscheme : (tyvar list * gtype)
166        retty : gtype * (gtype * gtype) list * (gtype * tx) *)
167     let rec generate_constraints gctx e =
168         let rec constrain ctx e =
169             match e with
170             | Literal e -> value e
171             | Var name ->
172                 let (_, (_, tau)) = List.find (fun x -> fst x = name) ctx in
173                 (tau, [], (tau, TypedVar name))
174             | If (e1, e2, e3) ->
175                 let t1, c1, tex1 = generate_constraints ctx e1 in
176                 let t2, c2, tex2 = generate_constraints ctx e2 in
177                 let t3, c3, tex3 = generate_constraints ctx e3 in
178                 let c = [ (booltype, t1); (t3, t2) ] @ c1 @ c2 @ c3 in
179                 let tex = TypedIf (tex1, tex2, tex3) in
180                 (t3, c, (t3, tex))
181             | Apply (f, args) ->

```

```

182   let t1, c1, tex1 = generate_constraints ctx f in
183   let ts2, c2, teks2 =
184     List.fold_left
185       (fun acc e ->
186         let t, c, x = generate_constraints ctx e in
187         let ts, cs, xs = acc in
188         (t :: ts, c @ cs, x :: xs))
189       ([], c1, []) (List.rev args)
190   in
191   (* reverse args to maintain arg order *)
192   let retType = fresh () in
193   ( retType,
194     (t1, Tfst.functiontype retType ts2) :: c2,
195     (retType, TypedApply (tex1, teks2)) )
196 | Let (bindings, expr) ->
197   let l = List.map (fun (_, e) -> generate_constraints ctx e) bindings in
198   let cns = List.concat (List.map (fun (_, c, _) -> c) l) in
199   let taus = List.map (fun (t, _, _) -> t) l in
200   let asts = List.map (fun (_, _, a) -> a) l in
201   let names = List.map fst bindings in
202   let ctx_addition =
203     List.map (fun (n, t) -> (n, ([], t))) (List.combine names taus)
204   in
205   let new_ctx = ctx_addition @ ctx in
206   let b_tau, b_cns, b_tast = generate_constraints new_ctx expr in
207   (b_tau, b_cns @ cns, (b_tau, TypedLet (List.combine names asts, b_tast)))
208 | Lambda (formals, body) ->
209   let binding = List.map (fun x -> (x, ([], fresh ()))) formals in
210   let new_context = binding @ ctx in
211   let t, c, tex = generate_constraints new_context body in
212   let _, tyschms = List.split binding in
213   let _, formaltys = List.split tyschms in
214   let typedFormals = List.combine formaltys formals in
215   ( Tfst.functiontype t formaltys,
216     c,
217     (Tfst.functiontype t formaltys, TypedLambda (typedFormals, tex)) )
218 and value v =
219   match v with
220   | Int e -> (inttype, [], (inttype, TLiteral (TInt e)))
221   | Char e -> (chartype, [], (chartype, TLiteral (TChar e)))
222   | Bool e -> (booltype, [], (booltype, TLiteral (TBool e)))
223   | Root t -> tree t
224 and tree t =
225   match t with
226   | Leaf -> Diagnostic.error (Diagnostic.Unimplemented "constraint generation for Leaf")
227   | Branch _ -> Diagnostic.error (Diagnostic.Unimplemented "constraint generation for Branch")

```

```

228   in
229   constrain gctx e
230
231
232   (* get_constraints - returns a list of Tasts
233      Tasts : [ (ident * (gtype * tx)) ] = [ (ident * texpr) | texpr ] = [ tdefs ]
234      tyscheme : (tyvar list * gtype) *)
235   let get_constraints (ctx : (ident * tyscheme) list) (d : defn) =
236     match d with
237     | Val (name, e) ->
238       let t, c, tex = generate_constraints ctx e in
239       (t, c, TVal (name, tex))
240     | Expr e ->
241       let (t, c, tex) = generate_constraints ctx e in
242       (t, c, TExpr tex)
243
244
245   (* input: (tyvar * gtype) list *)
246   (* retty: tdefn -> tdefn *)
247   let apply_subs (sub : (tyvar * gtype) list) =
248     match sub with
249     | [] -> (fun x -> x)
250     | xs ->
251       let final_ans =
252         (fun tdef ->
253           (* xs - the list of substitutions we want to apply *)
254           (* tdef - the tdefn we want to apply the substitutions to *)
255           let rec expr_only_case (x : texpr) =
256             List.fold_left
257               (* anon fun - takes one texpr and takes one substitution and subs substitution into the texpr *)
258               (fun (tast_gt, tast_tx) (tv, gt) ->
259                 (* updated_tast_tx - matches texpr with tx and recurses on expressions *)
260                 let updated_tast_tx = match tast_tx with
261                   | TypedIf (x, y, z) ->
262                     TypedIf (expr_only_case x, expr_only_case y, expr_only_case z)
263                   | TypedApply (x, xs) ->
264                     let txs = List.map expr_only_case xs in
265                     TypedApply (expr_only_case x, txs)
266                   | TypedLet ((its), x) -> TypedLet (List.map (fun (x, y) ->
267                     (x, expr_only_case y)) its, expr_only_case x)
268                   | TypedLambda (tyformals, body) ->
269                     TypedLambda ((List.map (fun (x, y) -> (tysubst (tv, gt) x, y))
270                       tyformals), expr_only_case body)
271                   | TLiteral x -> TLiteral x
272                   | TypedVar x -> TypedVar x
273               in

```

```

274         let temp = (tysubst (tv, gt) tast_gt, updated_tast_tx) in temp) x xs in
275     match tdef with
276     | TVal (name, x) -> TVal (name, (expr_only_case x))
277       (* Do we need to do anything with updating context here? *)
278     | TExpr x -> TExpr (expr_only_case x)
279     )
280   in final_ans
281
282   (* update_ctx - if the typed definition is a TVal this function will make sure
283     there are no unbound type variables and tha *)
284   let update_ctx ctx tydefn =
285     match tydefn with
286     | TVal (name, (gt, _)) ->
287       (name, (List.filter (fun x -> List.exists (fun y -> y = x) (ftvs gt)) (ftvs gt), gt))::ctx
288     | TExpr _ -> ctx
289
290
291   (* type_infer
292     input : ( ident / ident * expr ) list
293     returns : ( ident * (gtype * tx) ) list *)
294
295   let type_infer (ds : defn list) =
296     let rec infer_defns ctx defn =
297       match defn with
298       | [] -> []
299       | d :: ds ->
300         (* get the constraints for the defn *)
301         let _, cs, tex = get_constraints ctx d in
302         (* subs -> (Infer.tyvar * Infer.gtype) list *)
303         let subs = solve cs in
304         (* apply subs to tdefs *)
305         let tdefn = (apply_subs subs) tex in
306         (* update ctx *)
307         let ctx' = update_ctx ctx tdefn in
308         (* recurse *)
309         tdefn :: infer_defns ctx' ds
310     in
311     infer_defns prims ds
312
313   (* type_infer
314     input : ( ident / ident * expr ) list
315     returns : ( ident * (gtype * tx) ) list *)

```



## 11.8 mast.ml

Author(s): Amy

```
1  (* MAST -- monomorphized AST where polymorphism is removed *)
2  module StringMap = Map.Make(String)
3
4  type mname = string
5
6  type mtype =
7      Mtycon of mtycon
8      | Mtyvar of int
9      | Mconapp of mconapp
10 and mtycon =
11     MIntty
12     | MCharty
13     | MBoolty
14     | MTarrow of mtype
15 and mconapp = (mtycon * mtype list)
16
17 let integerTy = Mtycon MIntty
18 let characterTy = Mtycon MCharty
19 let booleanTy = Mtycon MBoolty
20 let functionTy (ret, args) = Mconapp (MTarrow ret, args)
21
22
23
24 type mexpr = mtype * mx
25 and mx =
26     | MLiteral   of mvalue
27     | MVar       of mname
28     | MIf        of mexpr * mexpr * mexpr
29     | MApply     of mexpr * mexpr list
30     | MLet       of (mname * mexpr) list * mexpr
31     | MLambda    of (mtype * mname) list * mexpr
32 and mvalue =
33     | MChar      of char
34     | MInt       of int
35     | MBool      of bool
36     | MRoot      of mtree
37 and mtree =
38     | MLeaf
39     | MBranch    of mvalue * mtree * mtree
40
41 type mdefn =
42     | MVal       of mname * mexpr
43     | MExpr      of mexpr
```

```

44
45
46 type polyty_env = mexpr StringMap.t
47
48 type mprog = mdefn list
49
50
51
52 (* Pretty printer *)
53
54 (* String of mtypes *)
55 let rec string_of_mtype = function
56   | Mtycon ty -> string_of_mtycon ty
57   | Mconapp con -> string_of_mconapp con
58   | Mtyvar i -> "" ^ string_of_int i
59 and string_of_mtycon = function
60   | MIntty -> "int"
61   | MBoolty -> "bool"
62   | MCharty -> "char"
63   | MTarrow (retty) -> string_of_mtype retty
64 and string_of_mconapp (tyc, tys) =
65   string_of_mtycon tyc ^ " ("
66   ^ String.concat " " (List.map string_of_mtype tys) ^ ")"
67
68
69 (* String of a typed expression (mexpr) == (type, m-expression) *)
70 let rec string_of_mexpr (typ, exp) =
71   "[" ^ string_of_mtype typ ^ "]" ^ string_of_mx exp
72 and string_of_mx = function
73   | MLiteral v -> string_of_mvalue v
74   | MVar id -> id
75   | MIf (e1, e2, e3) ->
76     "(if " ^ string_of_mexpr e1 ^ " "
77     ^ string_of_mexpr e2 ^ " "
78     ^ string_of_mexpr e3 ^ ")"
79   | MApply (f, args) ->
80     "(" ^ string_of_mexpr f ^ " "
81     ^ String.concat " " (List.map string_of_mexpr args) ^ ")"
82   | MLet (binds, body) ->
83     let string_of_binding (id, e) =
84       "[" ^ id ^ " " ^ (string_of_mexpr e) ^ "]"
85     in
86     "(let (" ^ String.concat " " (List.map string_of_binding binds) ^ ") "
87     ^ string_of_mexpr body ^ ")"
88   | MLambda (formals, body) ->
89     let formalStringlist = List.map (fun (ty, x) -> string_of_mtype ty ^ " " ^ x) formals in

```

```

90     "(lambda (" ^ String.concat ", " formalStringlist
91     ^ ") " ^ string_of_mexpr body ^ ")"
92   (* toString for Mast.mvalue *)
93   and string_of_mvalue = function
94   | MChar c -> String.make 1 c
95   | MInt i -> string_of_int i
96   | MBool b -> if b then "#t" else "#f"
97   | MRoot tr -> string_of_mtree tr
98   (* toString for Mast.mtree *)
99   and string_of_mtree = function
100  | MLeaf -> "leaf"
101  | MBranch (v, sib, child) ->
102    "(tree " ^ string_of_mvalue v ^ " "
103    ^ string_of_mtree sib ^ " "
104    ^ string_of_mtree child ^ ")"
105
106
107
108   (* String of a mono typed defn (mdefn) *)
109   let string_of_mdefn = function
110   | MVal (id, me) -> "(val " ^ id ^ " " ^ string_of_mexpr me ^ ")"
111   | MExpr me -> string_of_mexpr me
112
113
114   (* String of the mprog == mdefn list *)
115   let string_of_mprog mdefs =
116     String.concat "\n" (List.map string_of_mdefn mdefs) ^ "\n"

```

## 11.9 mono.ml

Author(s): Amy

```
1  (* Monomorphizes a typed (incl poly) program *)
2
3  open Tast
4  open Mast
5
6
7  (* Function takes a tprog (list of typed definitions),
8     and monomorphizes it. to produce a mprog *)
9  let monomorphize (tdefsns : tprog) =
10
11     (* Takes a Tast.gtype and returns the equivalent Mast.mtype *)
12     let rec ofGtype = function
13         TYCON ty    -> Mtycon  (ofTycon ty)
14       | TYVAR tp    -> Mtyvar  (ofTyvar tp)
15       | CONAPP con  -> Mconapp (ofConapp con)
16     and ofTycon = function
17         TyInt       -> MIntty
18       | TyBool      -> MBoolty
19       | TyChar      -> MCharty
20       | TArrow rety -> MTarrow (ofGtype rety)
21     and ofTyvar = function
22         TVariable n -> n
23     and ofConapp (tyc, tys) = (ofTycon tyc, List.map ofGtype tys)
24     in
25
26
27
28
29     (* Takes an mtype, and returns true if it is polymorphic, false o.w. *)
30     let rec isPolymorphic (typ : mtype) = match typ with
31         | Mtycon t -> poly_tycon t
32         | Mtyvar _ -> true
33         | Mconapp c -> poly_conapp c
34     and poly_tycon = function
35         MIntty | MBoolty | MCharty -> false
36       | MTarrow t -> isPolymorphic t
37     and poly_conapp (tyc, mtys) =
38         (poly_tycon tyc)
39       || (List.fold_left (fun init mtyp -> init || (isPolymorphic mtyp))
40           false mtys)
41     in
42
43
```

```

44
45  (* Takes a type environment and a string key "id". Returns the
46     value (mtype) that the key maps to. *)
47  let lookup (id : mname) (gamma : polyty_env) =
48      StringMap.find id gamma
49  in
50
51
52  (* Takes a name and an polyty_env, and inserts it into the map *)
53  let set_aside (id : mname) ((ty, exp) : mexpr) (gamma : polyty_env) =
54      StringMap.add id (ty, exp) gamma
55  in
56
57
58  (* Returns true if ty is a type variable *)
59  let isTyvar (ty : mtype) = match ty with
60      Mtycon _   -> false
61      | Mtyvar _ -> true
62      | Mconapp _ -> false
63  in
64
65  (* Returns true if ty is a function type *)
66  let isFunctionType (ty : mtype) = match ty with
67      Mconapp (MTarrow _, _) -> true
68      | _                     -> false
69  in
70
71
72  (* (fty, exp) == poly lambda expression
73     (ty) == mono function type *)
74  let resolve (prog : mprog) (id : mname) (ty : mtype) ((fty, exp) : mexpr) =
75      (* Given a function type, returns the list of the types of the arguments *)
76      let get_type_of_args = function
77          Mconapp (MTarrow _, formaltys) -> formaltys
78          | _ -> Diagnostic.error
79              (Diagnostic.MonoError "cannot monomorphize non-function type")
80      in
81
82      let formaltys      = get_type_of_args ty in (* mono *)
83      let polyargtys     = get_type_of_args fty in (* poly *)
84      let substitutions = List.combine polyargtys formaltys in
85
86  (* Given a (polymorphic) mtype, returns the monomorphic version *)
87  let resolve_mty (mty : mtype) =
88      let apply_subs typ (arg, sub) =
89          if isTyvar arg

```

```

90     then
91         let tyvarID =
92             (match arg with
93                 Mtyvar i -> i
94                 | _ -> Diagnostic.error
95                     (Diagnostic.MonoError "non-tyvar substitution"))
96     in
97     let rec search_mtype = function
98         Mtycon tyc -> Mtycon (search_tycon tyc)
99         | Mtyvar i -> if i = tyvarID then sub else Mtyvar i
100        | Mconapp con -> Mconapp (search_con con)
101    and search_tycon = function
102        MIntty -> MIntty
103        | MCharty -> MCharty
104        | MBoolty -> MBoolty
105        | MTarrow retty -> MTarrow (search_mtype retty)
106    and search_con (tyc, mtys) =
107        (search_tycon tyc, List.map search_mtype mtys)
108    in search_mtype typ
109 else typ
110 in List.fold_left apply_subs mty substitutions
111 in
112
113     (* Given an (polymorphic) mx, returns the monomorphic version, with an
114        updated program, if any. *)
115     let rec resolve_mx pro = function
116         MLiteral l -> (MLiteral l, pro)
117         | MVar v -> (MVar v, pro)
118         | MIf ((t1, e1), (t2, e2), (t3, e3)) ->
119             let t1' = resolve_mty t1 in
120             let t2' = resolve_mty t2 in
121             let t3' = resolve_mty t3 in
122             let (e1', pro1) = resolve_mx pro e1 in
123             let (e2', pro2) = resolve_mx pro1 e2 in
124             let (e3', pro3) = resolve_mx pro2 e3 in
125             (MIf ((t1', e1'), (t2', e2'), (t3', e3')), pro3)
126         | MApply ((appty, app), args) ->
127             (* resolve the expression thats applied *)
128             let appty' = resolve_mty appty in
129             let (app', pro') = resolve_mx pro app in
130             (* resolve the arguments of the application *)
131             let (argtys, argexps) = List.split args in
132             let argtys' = List.map resolve_mty argtys in
133             let (argexps', pro'') = resolve_listOf_mx pro' argexps in
134             let args' = List.combine argtys' argexps' in
135             (MApply ((appty', app'), args'), pro'')

```

```

136 | MLet (bs, body) ->
137   let (names, bexprs) = List.split bs in
138   let (btys, bmxs) = List.split bexprs in
139   let btys' = List.map resolve_mty btys in
140   let (bmxs', pro') = resolve_listOf_mx pro bmxs in
141   let bs' = List.combine names (List.combine btys' bmxs') in
142   let (body', pro'') = resolve_mexpr pro' body in
143   (MLet (bs', body'), pro'')
144 | MLambda (formals, body) ->
145   let (formaltys, names) = List.split formals in
146   let formaltys' = List.map resolve_mty formaltys in
147   let formals' = List.combine formaltys' names in
148   let (body', pro') = resolve_mexpr pro body in
149   let lambdaExp = MLambda (formals', body') in
150   let pro'' = (MVal (id, (ty, lambdaExp))) :: pro' in
151   (lambdaExp, pro'')
152 and resolve_mexpr pro ((ty, mexp) : mexpr) =
153   let ty' = resolve_mty ty in
154   let (mexp', pro') = resolve_mx pro mexp in
155   let monoexp' = (ty', mexp') in
156   (monoexp', pro')
157 and resolve_listOf_mx pro (mxs : mx list) =
158   let (mx', pro') =
159     List.fold_left
160       (fun (mexlist, prog) mex ->
161         let (mex', prog') = resolve_mx prog mex in
162         (mex' :: mexlist, prog'))
163       ([], pro) mxs
164   in
165   let mx' = List.rev mx' in
166   (mx', pro')
167 in
168
169 let (exp', prog') = resolve_mx prog exp in
170 ((ty, exp'), prog')
171
172 in
173
174
175
176
177 (* Takes a texpr and returns the equivalent mexpr
178    and the prog (list of mdefsns) *)
179 let rec expr (gamma : polyty_env) (prog : mprog) ((ty, ex) : texpr) =
180   match ex with
181   | TLiteral l -> ((ofGtype ty, MLiteral (value l)), prog)

```

```

182 | TypedVar v ->
183   (* let () = print_endline ("looking for: " ^ v) in *)
184   let vartyp = (try fst (lookup v gamma)
185                 with Not_found ->
186                   (* let () = print_endline "didn't find it" in *)
187                     ofGtype ty) in
188   let actualtyp = ofGtype ty in
189   if (isPolymorphic vartyp) && (isFunctionType vartyp)
190   then
191     let polyexp = lookup v gamma in
192     let (_, prog') = resolve prog v actualtyp polyexp in
193     ((actualtyp, MVar v), prog')
194   else
195     ((actualtyp, MVar v), prog)
196 | TypedIf (t1, t2, t3) ->
197   let (mexp1, prog1) = expr gamma prog t1 in
198   let (mexp2, prog2) = expr gamma prog1 t2 in
199   let (mexp3, prog3) = expr gamma prog2 t3 in
200   ((fst mexp3, MIf (mexp1, mexp2, mexp3)), prog3)
201 | TypedApply (f, args) ->
202   let (f', prog') = expr gamma prog f in
203   let (args', prog'') =
204     List.fold_left
205       (fun (arglst, pro) arg ->
206         let (arg', pro') = expr gamma pro arg in
207         (arg' :: arglst, pro'))
208       ([], prog') args
209   in
210   let args' = List.rev args' in
211   ((ofGtype ty, MApply (f', args')), prog'')
212 | TypedLet (bs, body) ->
213   let binding (x, e) = let (e', _) = expr gamma prog e in (x, e') in
214   let bs' = List.map binding bs in
215   let (body', prog') = expr gamma prog body in
216   ((ofGtype ty, MLet (bs', body')), prog')
217 | TypedLambda (formals, body) ->
218   let (formaltys, names) = List.split formals in
219   let formaltys' = List.map ofGtype formaltys in
220   let formals' = List.combine formaltys' names in
221   let gamma' = List.fold_left
222     (fun env (ty, name) ->
223       if isPolymorphic ty
224       then set_aside name (ty, MVar name) env
225       else env)
226     gamma
227     formals' in

```



```

228     let (body', prog') = expr gamma' prog body in
229     ((ofGtype ty, MLambda (formals', body')), prog')
230 and value = function
231   | TChar c -> MChar c
232   | TInt i -> MInt i
233   | TBool b -> MBool b
234   | TRoot t -> MRoot (tree t)
235 and tree = function
236   | TLeaf -> MLeaf
237   | TBranch (v, t1, t2) -> MBranch (value v, tree t1, tree t2)
238 in
239
240
241 (* Takes the current mprog built so far, and one tdefn, and adds
242    the monomorphized version to the mprog. Returns a new mprog
243    with the new definition added in. *)
244 let mono ((gamma, prog) : polyty_env * mprog) = function
245   TVal (id, (ty, texp)) ->
246     let ((mty, mexp), prog') = expr gamma prog (ty, texp) in
247     if isPolymorphic mty
248     then
249       let gamma' = set_aside id (mty, mexp) gamma in
250       (gamma', MVal (id, (mty, mexp)) :: prog')
251     else (gamma, MVal (id, (mty, mexp)) :: prog')
252   | TExpr (ty, texp) ->
253     let ((mty, mexp), prog') = expr gamma prog (ty, texp) in
254     let () = if isPolymorphic mty then
255       Diagnostic.warning
256         (Diagnostic.MonoWarning ("polymorphic type leftover;"
257                                ^ " resolving to integers"))
258     else ()
259     in (gamma, MExpr (mty, mexp) :: prog')
260
261     (* if isPolymorphic mty
262        then (gamma, prog')
263        else (gamma, MExpr (mty, mexp) :: prog') *)
264 in
265
266 let (_, program) = List.fold_left mono (StringMap.empty, []) tdefs in
267
268
269
270
271 (* Bug/Bandaaid - unable to resolve polymorphism.
272    Iterate through the current "mono" typed program.
273    Insert integer type wherever leftover type variables remain *)

```

```

274 let buggy_resolve (prog : mprog) (def : mdefn) =
275
276   (* resolves any remaining polymorphism in an mexpr to integer type *)
277   let rec resolve_expr ((ty, exp) : mexpr) =
278     (* turns tyvar into int type *)
279     let rec resolve_mty = function
280       | Mtycon t -> Mtycon (resolve_tycon t)
281       | Mtyvar _ -> Mtycon MIntty
282       | Mconapp c -> Mconapp (resolve_conapp c)
283     and resolve_tycon = function
284       | MIntty -> MIntty
285       | MBoolty -> MBoolty
286       | MCharty -> MCharty
287       | MTarrow t -> MTarrow (resolve_mty t)
288     and resolve_conapp (tyc, mtys) =
289       (resolve_tycon tyc, List.map resolve_mty mtys)
290     in
291
292     (* finds and resolves any nested tyvars to int type *)
293     let resolve_mx = function
294       | MLiteral l -> MLiteral l
295       | MVar v -> MVar v
296       | MIf (e1, e2, e3) ->
297         let r1 = resolve_expr e1 in
298         let r2 = resolve_expr e2 in
299         let r3 = resolve_expr e3 in
300         MIf (r1, r2, r3)
301       | MApply (f, args) ->
302         let f' = resolve_expr f in
303         let args' = List.map resolve_expr args in
304         MApply (f', args')
305       | MLet (bs, body) ->
306         let bs' = List.map (fun (name, mex) ->
307           (name, resolve_expr mex))
308           bs in
309         let body' = resolve_expr body in
310         MLet (bs', body')
311       | MLambda (formals, body) ->
312         let formals' =
313           List.map (fun (mty, name) -> (resolve_mty mty, name)) formals in
314         let body' = resolve_expr body in
315         MLambda (formals', body')
316     in
317
318     (resolve_mty ty, resolve_mx exp)
319   in

```

```
320
321   (* Resolve the given mdefn *)
322   match def with
323   | MVal (id, ex) -> MVal (id, resolve_expr ex) :: prog
324   | MExpr ex      -> MExpr (resolve_expr ex) :: prog
325 in
326
327 List.fold_left buggy_resolve [] program
```

---

## 11.10 hast.ml

Author(s): Amy

```
1  (* HAST - resolves function types involved in the uses of HOFs *)
2
3  module StringMap = Map.Make(String)
4
5  type hname = string
6
7  type htype =
8      HTycon of htycon
9      | HConapp of hconapp
10 and htycon =
11     HIntty
12     | HCharty
13     | HBoolty
14     | HTarrow of htype
15     (* H-closures come with return type, formal types, free types *)
16     | HCLs of hname * htype * htype list * htype list
17 and hconapp = (htycon * htype list)
18
19 let intTy = HTycon HIntty
20 let charTy = HTycon HCharty
21 let boolTy = HTycon HBoolty
22
23 let partialClosurety (id, retty, formaltys, freetys) =
24     HTycon (HCLs (id, retty, formaltys, freetys))
25
26
27 type hexpr = htype * hx
28 and hx =
29     | HLiteral of hvalue
30     | HVar of hname
31     | HIf of hexpr * hexpr * hexpr
32     | HApply of hexpr * hexpr list
33     | HLet of (hname * hexpr) list * hexpr
34     | HLambda of (htype * hname) list * hexpr
35 and hvalue =
36     | HChar of char
37     | HInt of int
38     | HBool of bool
39     | HRoot of htree
40 and htree =
41     | HLeaf
42     | HBranch of hvalue * htree * htree
43
```

```

44 type hdefn =
45   | HVal      of hname * hexpr
46   | HExpr     of hexpr
47
48
49 type ty_env = ((int * htype) list) StringMap.t
50 type hof_env = (hname * hexpr) StringMap.t
51 let emptyEnvironment = StringMap.empty
52 let emptyListEnv = []
53
54
55 type hrec =
56   {
57     mutable program : hdefn list;
58     mutable gamma    : ty_env;
59     mutable hofs     : hof_env;
60   }
61
62 type hprog = hdefn list
63
64
65
66 (* Pretty printer *)
67 let rec string_of_htype = function
68   HTycon tyc -> string_of_hthycon tyc
69   | HConapp con -> string_of_hconapp con
70 and string_of_hthycon = function
71   HIntty -> "int"
72   | HCharty -> "char"
73   | HBoolty -> "bool"
74   | HTarrow retty -> string_of_htype retty
75   | HClis (id, retty, formaltys, freetys) ->
76     let string_of_list tys =
77       String.concat ", " (List.map string_of_htype tys) in
78     id ^ "{ " ^
79     string_of_htype retty
80     ^ " (" ^
81     string_of_list formaltys ^ " :: " ^
82     string_of_list freetys
83     ^ ") }"
84 and string_of_hconapp (tyc, tys) =
85   string_of_hthycon tyc ^ " ("
86   ^ String.concat " " (List.map string_of_htype tys) ^ ")"
87
88
89 (* String of a h-expression *)

```

```

90 let rec string_of_hexpr (typ,exp) =
91   "[" ^ string_of_htype typ ^ "]" ^ string_of_hx exp
92 and string_of_hx = function
93   | HLiteral v -> string_of_hvalue v
94   | HVar id -> id
95   | HIf (e1, e2, e3) ->
96     "(if " ^ string_of_hexpr e1 ^ " "
97     ^ string_of_hexpr e2 ^ " "
98     ^ string_of_hexpr e3 ^ ")"
99   | HApply (f, args) ->
100     "(" ^ string_of_hexpr f ^ " "
101     ^ String.concat " " (List.map string_of_hexpr args) ^ ")"
102   | HLet (binds, body) ->
103     let string_of_binding (id, e) =
104       "[" ^ id ^ " " ^ (string_of_hexpr e) ^ "]"
105     in
106     "(let (" ^ String.concat " " (List.map string_of_binding binds) ^ ") "
107     ^ string_of_hexpr body ^ ")"
108   | HLambda (formals, body) ->
109     let formalStringlist = List.map (fun (ty, x) ->
110       string_of_htype ty ^ " " ^ x)
111       formals in
112     "(lambda (" ^ String.concat ", " formalStringlist
113     ^ ") " ^ string_of_hexpr body ^ ")"
114 (* toString for Hast.hvalue *)
115 and string_of_hvalue = function
116   | HChar c -> String.make 1 c
117   | HInt i -> string_of_int i
118   | HBool b -> if b then "#t" else "#f"
119   | HRoot tr -> string_of_htree tr
120 (* toString for Hast.htree *)
121 and string_of_htree = function
122   | HLeaf -> "leaf"
123   | HBranch (v, sib, child) ->
124     "(tree " ^ string_of_hvalue v ^ " "
125     ^ string_of_htree sib ^ " "
126     ^ string_of_htree child ^ ")"
127
128
129
130 (* String of a hdefn *)
131 let string_of_hdefn = function
132   | HVal (id, e) -> "(val " ^ id ^ " " ^ string_of_hexpr e ^ ")"
133   | HExpr e -> string_of_hexpr e
134
135 (* Returns the string of an hprog *)

```

```
136 let string_of_hprog hdefns =  
137   String.concat "\n" (List.map string_of_hdefn hdefns) ^ "\n"
```

---

## 11.11 hof.ml

Author(s): Amy

```
1  (* Resolve higher order function uses (takes or returns functions)
2     to take or return closures. *)
3  open Mast
4  open Hast
5  module StringMap = Map.Make(String)
6
7  (* name used for anonymizing lambda functions *)
8  let anon = "_anon"
9  let count = ref 0
10
11  (* Pre-load rho with prints built in *)
12  let prerho env =
13    let add_prints map (k, v) =
14      StringMap.add k [v] map
15    in List.fold_left add_prints env
16      [("printi", (0, intTy)); ("printb", (0, boolTy));
17       ("printc", (0, charTy)); ("+", (0, intTy));
18       ("-", (0, intTy)); ("*", (0, intTy));
19       ("/", (0, intTy)); ("mod", (0, intTy));
20      ("<", (0, boolTy)); (">", (0, boolTy));
21      ("<=", (0, boolTy)); (">=", (0, boolTy));
22       ("!=i", (0, boolTy)); ("=i", (0, boolTy));
23       ("&&", (0, boolTy)); ("||", (0, boolTy));
24       ("not", (0, boolTy)); ("~", (0, intTy)) ]
25
26  (* list of variable names that get ignored/are not to be considered frees *)
27  let ignores = [ "printi"; "printb"; "printc";
28                  "+"; "-"; "*"; "/"; "mod";
29                  "<"; ">"; ">="; "<=";
30                  "!=i"; "=i"; "~";
31                  "&&"; "||"; "not" ]
32
33
34  (* partial hprog to return from this module *)
35  let res =
36    {
37      program = emptyListEnv;
38      gamma   = prerho emptyEnvironment;
39      hofs     = emptyEnvironment;
40    }
41
42  (* Inserts a hdefn into main hdefn list *)
43  let addMain d = res.program <- d :: res.program
```



```

44
45 (* Returns true if the given name id is already bound in the given
46 StringMap env. False otherwise *)
47 let isBound id env = StringMap.mem id env
48
49 (* Returns the first element of th value that "id" is bound to in the given
50 StringMap env. If the binding doesn't exist, Not_Found exception
51 is raised. *)
52 let find id env =
53   let occursList = StringMap.find id env in List.nth occursList 0
54
55 (* Adds a binding of k to v in the global StringMap env *)
56 let bindGamma k v = res.gamma <-
57   let currList = if isBound k res.gamma
58     then StringMap.find k res.gamma else [] in
59   let newList = v :: currList in
60   StringMap.add k newList res.gamma
61
62 let addHofs id lambda = res.hofs <- StringMap.add id lambda res.hofs
63
64 (* Adds a local binding of k to v in the given StringMap env *)
65 let bindLocal map k v =
66   let currList = if isBound k map then StringMap.find k map else [] in
67   let localList = (0, v) :: currList in
68   StringMap.add k localList map
69
70 (* Given a h-closure type, returns the closure's id *)
71 let getClosureId (cls : htype) = match cls with
72   HTycon (HCl (id, _, _, _)) -> id
73   | _ -> Diagnostic.error
74     (Diagnostic.TypeError ("Nonclosure-type accessed when"
75       ^ " trying to get closure ID"))
76
77
78 (* Converts a mtype to a htype *)
79 let rec ofMtype = function
80   Mtycon ty    -> HTycon (ofTycon ty)
81   | Mtyvar _    -> Diagnostic.error
82     (Diagnostic.TypeError ("unresolved polymorphic type"))
83   | Mconapp con -> HConapp (ofConapp con)
84 and ofTycon = function
85   MIntty      -> HIntty
86   | MBoolty   -> HBoolty
87   | MCharty   -> HCharty
88   | MTarrow retty -> HTarrow (ofMtype retty)
89 and ofConapp (tyc, tys) = (ofTycon tyc, List.map ofMtype tys)

```

```

90
91
92
93
94 (* Returns true if given htype is a function type *)
95 let rec isFunctionType = function
96   HTycon t  -> hof_tycon t
97   | HConapp t -> hof_conapp t
98 and hof_tycon = function
99   HIntty | HCharty | HBoolty -> false
100  | HTarrow _ -> true
101  | HClis (_, retty, argsty, _) ->
102    isFunctionType retty
103    || (List.fold_left
104        (fun init argty -> init || (isFunctionType argty))
105        false argsty)
106 and hof_conapp (tyc, tys) =
107   hof_tycon tyc
108   || (List.fold_left
109       (fun init typ -> init || (isFunctionType typ))
110       false tys)
111
112 (* Returns true if the given htype indicates a higher order function *)
113 let isHOF = function
114   HTycon (HClis (_, retty, argsty, _)) ->
115     isFunctionType retty
116     || (List.fold_left
117         (fun init argty -> init || (isFunctionType argty))
118         false argsty)
119   | _ -> false
120
121
122
123 (* Given expression an a string name n, returns true if n is
124 a free variable in the expression *)
125 let freeIn exp n =
126   let rec free (_, e) = match e with
127     | MLiteral _      -> false
128     | MVar s          -> s = n
129     | MIf (m1, m2, m3) -> free m1 || free m2 || free m3
130     | MApply (f, args) -> free f  || List.fold_left
131                           (fun a b -> a || free b)
132                           false args
133     | MLet (bs, body) -> List.fold_left (fun a (_, e) -> a || free e) false bs
134                           || (free body && not (List.fold_left
135                               (fun a (x, _) -> a || x = n)

```

```

136                                     false bs))
137 | MLambda (formals, body) ->
138   let (_, names) = List.split formals in
139   free body && not (List.fold_left (fun a x -> a || x = n) false names)
140 in free (integerTy, exp)
141
142 (* Given the formals list and body of a lambda (xs, e), and a
143 variable environment, the function returns a list of the types of the
144 free variables of the expression. The list of free types shall
145 not include those of built-in functions and primitives *)
146 let freeTypesOf (xs, e) gamma =
147   let freeGamma =
148     StringMap.filter
149       (fun n _ ->
150         if List.mem n ignores
151           then false
152           else freeIn (MLambda (xs, e)) n)
153     gamma
154   in
155   let getTy _ occursList res =
156     let (_, ty) = List.nth occursList 0 in
157     (* let id' = if num = 0 then id else "_" ^ id ^ string_of_int num in *)
158     (* (ty, id') :: res *)
159     ty :: res
160   in
161   StringMap.fold getTy freeGamma []
162
163
164 (* Resolves all lambda's function types, and uses of these expressions
165 to a new closure type. *)
166 let clean (mdefsns : mprog) =
167
168   (* Given a hexpr, returns a possibly updated hexpr using the given
169   environment. *)
170   let rec h_expr (env : ty_env) ((typ, exp) : hexpr) = match exp with
171   | HLiteral l -> (typ, HLiteral l)
172   | HVar v -> let (_, typ') = find v env in (typ', HVar v)
173   | HIf (e1, e2, e3) ->
174     let e1' = h_expr env e1 in
175     let e2' = h_expr env e2 in
176     let e3' = h_expr env e3 in
177     (fst e2', HIf (e1', e2', e3'))
178   | HApply (f, args) ->
179     let f' = h_expr env f in
180     let args' = List.map (h_expr env) args in
181     let retty =

```

```

182     (match fst f' with
183       HTycon (HClIs (_, ret, _, _)) -> ret
184     | _ -> intTy)
185   in (retty, HApply (f', args'))
186 | HLet (bs, body) ->
187   let bs' = List.map (fun (name, exp) -> (name, h_expr env exp)) bs in
188   let local_env = List.fold_left (fun map (name, (ty, _)) ->
189     bindLocal map name ty)
190     env bs' in
191   let body' = h_expr local_env body in
192   (fst body, HLet (bs', body'))
193 | HLambda (formals, body) ->
194   let local_env = List.fold_left (fun map (ty, name) ->
195     bindLocal map name ty)
196     env formals in
197   let body' = h_expr local_env body in
198   (fst body, HLambda (formals, body'))
199 in
200
201 (* Given a mexpr, returns the equivalent hexpr *)
202 let rec expr (env : ty_env) (mexp : mexpr) =
203   let rec expr' (ty, exp) = match exp with
204     MLiteral l -> (ofMtype ty, HLiteral (value l))
205   | MVar v -> let (_, typ) = find v env in (typ, HVar v)
206   | MIf (e1, e2, e3) ->
207     let e1' = expr' e1 in
208     let e2' = expr' e2 in
209     let e3' = expr' e3 in
210     (fst e2', HIf (e1', e2', e3'))
211   | MApply (f, args) ->
212     let (fty, f') = expr' f in
213     let args' = List.map expr' args in
214     if isHOF fty
215     then let (retty, fty') = resolveHOF env fty args' in
216        (retty, HApply ((fty', f'), args'))
217     else (ofMtype ty, HApply ((fty, f'), args'))
218   | MLet (bs, body) ->
219     let bs' = List.map (fun (name, e) -> (name, expr' e)) bs in
220     let local_env = List.fold_left (fun map (name, (typ, _)) ->
221       bindLocal map name typ)
222       env bs' in
223     let body' = expr local_env body in
224     (fst body', HLet (bs', body'))
225   | MLambda (formals, body) ->
226     (* name the closure *)
227     let id = anon ^ string_of_int !count in

```

```

228     let () = count := !count + 1 in
229
230     (* Put the formals in the environment *)
231     let (formaltys, formalnames) = List.split formals in
232     let formaltys' = List.map ofMtype formaltys in
233     let formals' = List.combine formaltys' formalnames in
234     let local_env = List.fold_left (fun map (typ, name) ->
235                                     bindLocal map name typ)
236                                     env formals' in
237     let body' = expr local_env body in
238     let retty = fst body' in
239     let freetys = freeTypesOf (formals, body) env in
240
241     (* Create new closure type *)
242     let closureType =
243         partialClosurety (id, retty, formaltys', freetys) in
244     (closureType, HLambda (formals', body'))
245 and value = function
246   | MChar c -> HChar c
247   | MInt i -> HInt i
248   | MBool b -> HBool b
249   | MRoot t -> HRoot (tree t)
250 and tree = function
251   | MLeaf -> HLeaf
252   | MBranch (v, t1, t2) -> HBranch (value v, tree t1, tree t2)
253 and resolveHOF (env : ty_env) (closuretype : htype)
254     (arguments : hexpr list) =
255     (* Extract argument actual types, and name of closure
256         being referenced *)
257     let (argtypes, _) = List.split arguments in
258     let cloName = getClosureId closuretype in
259     let (id, (_, lambdaexp)) = StringMap.find cloName res.hofs in
260     (* Function returns the subexpressions of a lambda expression *)
261     let get_lambda_subxs = function
262         HLambda (formals, body) -> (formals, body)
263     | _ -> Diagnostic.error
264         (Diagnostic.TypeError ("Nonclosure-type accessed when"
265                               ^ " tryong to get lambda subexpressions"))
266     in
267     (* Function replaces return types and argument types of a closure
268         type with the new given types passed in. *)
269     let resolve_cls_ty retty argtys = function
270         HTycon (HClS (id, _, _, freetys)) ->
271             HTycon (HClS (id, retty, argtys, freetys))
272     | _ -> Diagnostic.error
273         (Diagnostic.TypeError ("Nonclosure-type accessed when"

```

```

274         ^ " trying to resolve closure type"))
275     in
276     (* Lambda subexpressions *)
277     let (formals, body) = get_lambda_subxs lambdaexp in
278     let (_, formalnames) = List.split formals in
279     let formals' = List.combine argtypes formalnames in
280
281     let local_env = List.fold_left (fun map (typ, name) ->
282                                     bindLocal map name typ)
283                                     env formals' in
284     let body' = h_expr local_env body in
285     let retty = fst body' in
286     (* Resolve new closure types within closure types *)
287     let newClsty = resolve_cls_ty retty argtypes closuretype in
288     let newLambDef = HVal (id, (newClsty, HLambda (formals', body'))) in
289     let () = addMain newLambDef in
290     (retty, newClsty)
291
292     in expr' mexp
293 in
294
295
296     (* iterate through each defn and remove the hofs uses *)
297     let hofDef (def : mdefn) = match def with
298     | MVal (id, ex) ->
299         let (occurs, _) = if (isBound id res.gamma)
300                             then (find id res.gamma)
301                             else (0, intTy) in
302         let (ty, ex') = expr res.gamma ex in
303         let () = bindGamma id (occurs + 1, ty) in
304         if isHOF ty then addHofs (getClosureId ty) (id, (ty, ex'))
305         else addMain (HVal (id, (ty, ex')))
306     | MExpr ex ->
307         let hexp = expr res.gamma ex in addMain (HExpr hexp)
308
309     in
310
311     let _ = List.iter hofDef mdefns in
312     List.rev res.program

```

## 11.12 cast.ml

Author(s): Amy

```
1  (*
2    Closure converted Abstract Syntax tree
3    Assumes name-check and type-check have already happened
4  *)
5
6
7  module StringMap = Map.Make(String)
8
9  type cname = string
10
11  type ctype =
12    Tycon of tycon
13    | Conapp of conapp
14  and tycon =
15    Intty
16    | Charty
17    | Boolty
18    | Tarrow of ctype
19    | Clo of cname * ctype * ctype list
20  and conapp = (tycon * ctype list)
21
22
23  let intty = Tycon Intty
24  let charty = Tycon Charty
25  let boolty = Tycon Boolty
26
27  let funty (ret, args) =
28    Conapp (Tarrow ret, args)
29  let closuretype (id, functy, freetys) =
30    Tycon (Clo (id, functy, freetys))
31
32  (* int StringMap.t - for our rho/variable environment
33     (DOES NOT MAP TO VALUES) *)
34  type var_env = ((int * ctype) list) StringMap.t
35  let emptyEnv = StringMap.empty
36  let emptyList = []
37
38
39
40
41  type cexpr = ctype * cx
42  and cx =
43    | CLiteral of cvalue
```

```

44 | CVar      of cname
45 | CIf      of cexpr * cexpr * cexpr
46 | CApply  of cexpr * cexpr list * int
47 | CLet    of (cname * cexpr) list * cexpr
48 | CLambda  of cname * cexpr list
49 and cvalue =
50 | CChar    of char
51 | CInt     of int
52 | CBool    of bool
53 | CRoot    of ctree
54 and ctree =
55 | CLeaf
56 | CBranch  of cvalue * ctree * ctree
57
58 type cdefn =
59 | CVal      of cname * cexpr
60 | CExpr     of cexpr
61
62
63 (* function definiton record type (imperative style to record information) *)
64 type fdef =
65 {
66   body      : cexpr;
67   rettyp    : ctype;
68   fname     : cname;
69   formals   : (ctype * cname) list;
70   frees     : (ctype * cname) list;
71 }
72
73
74 (* closure is specifically a Tycon (Clo (struct name, function type field, frees field)) *)
75 type closure = ctype
76
77 (* a CAST *)
78 type cprog =
79 {
80   mutable main      : cdefn list; (* list for main instruction *)
81   mutable functions  : fdef list;  (* table of function definitions *)
82   mutable rho        : var_env;    (* variable declaration table *)
83   mutable structures : closure list;
84 }
85
86
87 (* Pretty Print *)
88 (* returns string of a ctype *)
89 let rec string_of_ctype = function

```



```

90     Tycon ty -> string_of_tycon ty
91   | Conapp con -> string_of_conapp con
92 and string_of_tycon = function
93   Intty  -> "int"
94   | Charty -> "char"
95   | Boolty -> "bool"
96   | Tarrow (retty) -> string_of_ctype retty
97   | Clo (sname, funty, freetys) ->
98     sname
99     (* ^ " {" *)
100    ^ " { "
101    ^ string_of_ctype funty ^ "; "
102    ^ String.concat ", " (List.map string_of_ctype freetys)
103    ^ "}"
104 and string_of_conapp (tyc, tys) =
105   string_of_tycon tyc ^ " ("
106   ^ String.concat ", " (List.map string_of_ctype tys) ^ ")"
107
108
109 (* stringifies cexpr *)
110 let rec string_of_cexpr (_, e) =
111   (* "[" *)
112   (* ^ string_of_ctype ty ^ " : " *)
113   (* ^ *)
114   string_of_cx e
115   (* ^ "]" *)
116 and string_of_cx = function
117   | CLiteral v -> string_of_cvalue v
118   | CVar n -> n
119   | CIf (e1, e2, e3) ->
120     "(if " ^ string_of_cexpr e1 ^ " "
121     ^ string_of_cexpr e2 ^ " "
122     ^ string_of_cexpr e3 ^ ")"
123   | CApply (f, args, _) ->
124     "(" ^ string_of_cexpr f ^ " "
125     ^ String.concat " " (List.map string_of_cexpr args) ^ ")"
126   | CLet (binds, body) ->
127     let string_of_binding (id, e) =
128       "[" ^ id ^ " " ^ (string_of_cexpr e) ^ "]"
129     in "(let (" ^ String.concat " " (List.map string_of_binding binds)
130       ^ ") " ^ string_of_cexpr body ^ ")"
131   | CLambda (id, frees) ->
132     "(" ^ id ^ " "
133     ^ String.concat " " (List.map string_of_cexpr frees)
134     ^ ")"
135 and string_of_cvalue = function

```

```

136 | CChar c -> String.make 1 c
137 | CInt i -> string_of_int i
138 | CBool b -> if b then "#t" else "#f"
139 | CRoot t -> string_of_ctree t
140 and string_of_ctree = function
141 | CLeaf -> "leaf"
142 | CBranch (v, sib, child) ->
143   "(tree " ^ string_of_cvalue v ^ " "
144   ^ string_of_ctree sib ^ " "
145   ^ string_of_ctree child ^ ")"
146
147 (* stringifies a cdefn *)
148 let string_of_cdefn = function
149 | CVal (id, e) -> "(val " ^ id ^ " " ^ string_of_cexpr e ^ ")"
150 | CExpr (cexp) -> string_of_cexpr cexp
151
152 let string_of_main main =
153   String.concat "\n" (List.map string_of_cdefn main) ^ "\n"
154
155 (* stringifies a list of fdefs *)
156 let string_of_functions (funcs : fdef list) =
157   let string_of_fdef ret_string {
158     rettyp = return;
159     fname = fname;
160     formals = formals;
161     frees = frees;
162     body = body;
163   } =
164     let string_of_formal (ty, para) = string_of_ctype ty ^ " " ^ para in
165     let args = formals @ frees in
166     let def = string_of_ctype return ^ " " ^ fname ^ " ("
167       ^ String.concat ", " (List.map string_of_formal args)
168       ^ ")\n{\n"
169       ^ string_of_cexpr body ^ "\n}\n"
170     in ret_string ^ def ^ "\n"
171   in List.fold_left string_of_fdef "" funcs
172
173 (* stringifies the variable lookup table *)
174 let string_of_rho rho =
175   StringMap.fold (fun id occursList s ->
176     let (num, ty) = List.nth occursList 0 in
177     s ^ id ^ ": " ^ string_of_ctype ty
178     ^ " " ^ id ^ string_of_int num
179     ^ "\n"
180   )
181   rho ""

```

```

182
183 (* Returns string of the list of closure/struct types *)
184 let string_of_structures structss =
185     let string_of_struct tyc = "struct " ^ string_of_ctype tyc
186     in String.concat "\n" (List.map string_of_struct structss)
187
188
189 (* Returns string of a cprog *)
190 let string_of_cprog { main = main; functions = functions;
191                     rho = rho;   structures = structures } =
192     "Main:\n" ^
193     string_of_main main ^ "\n\n" ^
194     "Functions:\n" ^
195     string_of_functions functions ^ "\n\n" ^
196     "Rho:\n" ^
197     string_of_rho rho ^ "\n\n" ^
198     "Structures:\n" ^
199     string_of_structures structures ^ "\n\n"

```

## 11.13 conversion.ml

Author(s): Amy, Eliza

```
1  (* Closure conversion for groot compiler *)
2  open Hast
3  open Cast
4
5  module StringMap = Map.Make(String)
6
7  (*****)
8
9  (* Pre-load rho with prints built in *)
10 let prerho env =
11   let add_prints map (k, v) =
12     StringMap.add k [v] map
13   in List.fold_left add_prints env
14     [("printi", (0, funty (intty, [intty])));
15      ("printb", (0, funty (boolty, [boolty])));
16      ("printc", (0, funty (charty, [charty])));
17      ("+", (0, funty (intty, [intty; intty])));
18      ("-", (0, funty (intty, [intty; intty])));
19      ("*", (0, funty (intty, [intty; intty])));
20      ("/", (0, funty (intty, [intty; intty])));
21      ("mod", (0, funty (intty, [intty; intty])));
22     ("<", (0, funty (boolty, [intty; intty])));
23     (">", (0, funty (boolty, [intty; intty])));
24     ("<=", (0, funty (boolty, [intty; intty])));
25     (">=", (0, funty (boolty, [intty; intty])));
26      ("!=i", (0, funty (boolty, [intty; intty])));
27      ("=i", (0, funty (boolty, [intty; intty])));
28      ("&&", (0, funty (boolty, [boolty; boolty])));
29      ("||", (0, funty (boolty, [boolty; boolty])));
30      ("not", (0, funty (boolty, [boolty])));
31      ("~", (0, funty (intty, [intty]))]
32
33  (* list of variable names that get ignored/are not to be considered frees *)
34  let ignores = [ "printi"; "printb"; "printc";
35                 "+"; "-"; "*"; "/"; "mod";
36                 "<"; ">"; ">="; "<="; "!=i"; "=i";
37                 "&&"; "||"; "not"; "~" ]
38
39
40  (* partial cprog to return from this module *)
41  let res =
42    {
43      main      = emptyList;
```

```

44     functions    = emptyList;
45     rho          = prerho emptyEnv;
46     structures   = emptyList
47   }
48
49
50
51
52   (* Converts a htype to a ctype *)
53   let rec ofHtype = function
54     HTycon ty    -> Tycon (ofTycon ty)
55   | HConapp con  -> Conapp (ofConapp con)
56   and ofTycon = function
57     HIntty       -> Intty
58   | HBoolty      -> Boolty
59   | HCharty      -> Charty
60   | HTarrow retty -> Tarrow (ofHtype retty)
61   | HCLs (id, retty, argsty, freetys) ->
62     let retty' = ofHtype retty in
63     let argsty' = List.map ofHtype argsty in
64     let freetys' = List.map ofHtype freetys in
65     Clo (id ^ "_struct", funty (retty', argsty' @ freetys'), freetys')
66   and ofConapp (tyc, tys) = (ofTycon tyc, List.map ofHtype tys)
67
68
69   (* puts the given cdefn into the main list *)
70   let addMain d = res.main <- d :: res.main
71
72   (* puts the given function name (id) mapping to its definition (f) in the
73     functions StringMap *)
74   let addFunction f = res.functions <- f :: res.functions
75
76   let getFunction id =
77     List.find (fun frecord -> id = frecord.fname) res.functions
78
79   let addClosure elem = res.structures <- elem :: res.structures
80
81   (* Returns true if the given name id is already bound in the given
82     StringMap env. False otherwise *)
83   let isBound id env = StringMap.mem id env
84
85   (* Adds a binding of k to v in the global StringMap env *)
86   let bind k v = res.rho <-
87     let currList = if isBound k res.rho then StringMap.find k res.rho else [] in
88     let newList = v :: currList in
89     StringMap.add k newList res.rho

```

```

90
91 (* Returns the value id is bound to in the given StringMap env. If the
92 binding doesn't exist, Not_Found exception is raised. *)
93 let find id env =
94   let occursList = StringMap.find id env in List.nth occursList 0
95
96 (* Adds a local binding of k to ctype in the given StringMap env *)
97 let bindLocal map k ty =
98   let currList = if isBound k map then StringMap.find k map else [] in
99   let locallist = (0, ty) :: currList in
100   StringMap.add k locallist map
101
102 (* Given expression an a string name n, returns true if n is
103 a free variable in the expression *)
104 let freeIn exp n =
105   let rec free (_, e) = match e with
106   | HLiteral _      -> false
107   | HVar s          -> s = n
108   | HIf (s1, s2, s3) -> free s1 || free s2 || free s3
109   | HApply (f, args) -> free f || List.fold_left
110                        (fun a b -> a || free b)
111                        false args
112   | HLet (bs, body) -> List.fold_left (fun a (_, e) -> a || free e) false bs
113                        || (free body && not (List.fold_left
114                                           (fun a (x, _) -> a || x = n)
115                                           false bs))
116   | HLambda (formals, body) ->
117     let (_, names) = List.split formals in
118     free body && not (List.fold_left (fun a x -> a || x = n) false names)
119   in free (intTy, exp)
120
121
122 (* Given the formals list and body of a lambda (xs, e), and a
123 variable environment, the function returns an environment with only
124 the free variables of this lambda. The environment of frees shall
125 not inculde the names of built-in functions and primitives *)
126 let improve (xs, e) rho =
127   StringMap.filter
128     (fun n _ ->
129      if List.mem n ignores
130      then false
131      else freeIn (HLambda (xs, e)) n)
132   rho
133
134
135 (* Given a var_env, returns a (ctype * name) list version *)

```

```

136 let toParamList (venv : var_env) =
137   StringMap.fold
138     (fun id occursList res ->
139       let (num, ty) = List.nth occursList 0 in
140       let id' = if num = 0 then id
141         else "_" ^ id ^ "_" ^ string_of_int num in
142       (ty, id') :: res)
143   venv []
144
145   (* turns a list of ty * name list to a Var list *)
146 let convertToVars (frees : (ctype * cname) list) =
147   List.map (fun (t, n) -> (t, CVar n)) frees
148
149
150
151   (* Generate a new function type for lambda expressions in order to account
152     for free variables, when given the original function type and an
153     association list of gtypes and var names to add to the new formals list
154     of the function type. *)
155 let newFunType (origTyp : htype) (newRet : ctype)
156   (toAdd : (ctype * cname) list) =
157   (match origTyp with
158     HTycon (HCLs (_, _, argsty, _)) ->
159     let newFormalTys = List.map ofHtype argsty in
160     let (newFreeTys, _) = List.split toAdd in
161     funty (newRet, newFormalTys @ newFreeTys)
162   | _ -> raise (Failure "Non-function function type"))
163
164
165
166   (* Converts given sexpr to cexpr, and returns the cexpr *)
167 let rec hexprToCexpr (env : var_env) (e : hexpr) =
168   let rec expr ((typ, ex) : hexpr) = match ex with
169   | HLiteral v -> (ofHtype typ, CLiteral (value v))
170   | HVar s      ->
171     (* In case s is a name of a define, get the closure type *)
172     let (occurs, ctyp) = find s env in
173     (* to match the renaming convention in svalToCval, and to ignore
174        built in prints *)
175     let vname = if occurs = 0
176       then s
177       else "_" ^ s ^ "_" ^ string_of_int occurs
178     in (ctyp, CVar (vname))
179   | HIf (e1, e2, e3) ->
180     let e1' = expr e1
181     and e2' = expr e2

```

```

182     and e3' = expr e3 in
183     (fst e2', CIf (e1', e2', e3'))
184 | HApply (f, args) ->
185     let (ctyp, f') = expr f in
186     let normalargs = List.map expr args in
187     (* actual type of the function application is the type of the return*)
188     let (retty, freesCount) =
189         (match ctyp with
190          Tycon (Clo (_, functy, freetys)) ->
191              (match functy with
192               Conapp (Tarrow ret, _) -> (ret, List.length freetys)
193               | _ -> raise (Failure "Non-function function type"))
194          | Conapp (Tarrow ret, _) -> (ret, 0)
195          | _ -> (intty, 0)) in
196     (retty, CApply ((ctyp, f'), normalargs, freesCount))
197 | HLet (bs, body) ->
198     let bs' = List.map (fun (name, hex) -> (name, expr hex)) bs in
199     let local_env =
200         List.fold_left (fun map (name, (ty, _)) ->
201                         bindLocal map name ty)
202                         env bs' in
203     let (ctyp, body') = hexprToCexpr local_env body in
204     (ctyp, CLet (bs', (ctyp, body')))
205     (* Suppose we hit a lambda expression, turn it into a closure *)
206 | HLambda (formals, body) -> create_anon_function formals body typ env
207 and value = function
208 | HChar c          -> CChar c
209 | HInt i           -> CInt i
210 | HBool b          -> CBool b
211 | HRoot t          -> CRoot (tree t)
212 and tree = function
213 | HLeaf            -> CLeaf
214 | HBranch (v, t1, t2) -> CBranch (value v, tree t1, tree t2)
215 in expr e
216 (* When given just a lambda expresion without a user defined identity/name
217    this function will generate a name and give the function a body --
218    Lambda lifting. *)
219 and create_anon_function (fformals : (hType * hname) list) (fbody : hexpr)
220                          (closurety : hType) (env : var_env) =
221
222
223 let fformals' = List.map (fun (hty, x) -> (ofHType hty, x)) fformals in
224 let local_env = List.fold_left
225                 (fun map (formtyp, name) ->
226                  bindLocal map name formtyp)
227                 env fformals' in

```



```

228 let func_body = hexprToCexpr local_env fbody in
229 let rettype = fst func_body in
230 let freeformals = toParamList (improve (fformals, fbody) env) in
231
232 (* All anonymous functions are named the same and numbered. *)
233 (* Given a h-closure type, returns the closure's id *)
234 let getClsID (cls : htype) = match cls with
235     HTycon (HCls (id, _, _, _)) -> id
236   | _ -> Diagnostic.error
237       (Diagnostic.TypeError ("Conversion: Nonclosure-type accessed"
238                             ^ " when trying to get closure ID"))
239 in
240 let id = getClsID closurety in
241
242 (* Create the record that represents the function body and definition *)
243 let func_def =
244     {
245         body      = func_body;
246         rettype   = rettype;
247         fname     = id;
248         formals   = fformals';
249         frees     = freeformals;
250     }
251 in
252 let () = addFunction func_def in
253
254 (* New function type will include the types of free arguments *)
255 let anonFunTy = newFunType closurety rettype freeformals in
256 (* Record the type of the anonymous function and its "rea" ftype *)
257 let () = bind id (1, anonFunTy) in
258 (* The value of a Lambda expression is a Closure -- new type construction
259   will help create the structs in codegen that represents the closure *)
260 let (freetys, _) = List.split freeformals in
261 let clo_ty = closuretype (id ^ "_struct", anonFunTy, freetys) in
262 let () = addClosure clo_ty in
263 let freeVars = convertToVars freeformals in
264 (clo_ty, CLambda (id, freeVars))
265
266
267
268 (* Converts given SVal to CVal, and returns the CVal *)
269 let hvalToCval (id, (ty, e)) =
270     (* check if id was already defined in rho, in order to get
271       the actual frequency the variable name was defined.
272       The (0, inttype is a placeholder) *)
273 let (occurs, _) = if (isBound id res.rho)

```

```

274         then (find id res.rho)
275         else (0, intty) in
276     (* Modify the name to account for the redefinitions, and so old closures
277       can access original variable values *)
278     let id' = "_" ^ id ^ "_" ^ string_of_int (occurs + 1) in
279     let (ty', cexp) = hexprToCexpr res.rho (ty, e) in
280     (* bind original name to the number of occurrences and the variable's type *)
281     let () = bind id (occurs + 1, ty') in
282     (* Return the possibly new CVal definition *)
283     CVal (id', (ty', cexp))
284     (*****)
285
286
287
288     (* Given an hprog (which is an hdefn list), convert returns a
289       cprog version. *)
290     let conversion hdefns =
291         (* With a given sdefn, function converts it to the appropriate CAST type
292           and sorts it to the appropriate list in a cprog type. *)
293         let convert = function
294             | HVal (id, (ty, hexp)) ->
295                 let cval = hvalToCval (id, (ty, hexp)) in addMain cval
296             | HExpr e ->
297                 let cexp = hexprToCexpr res.rho e in addMain (CExpr cexp)
298         in
299
300         let _ = List.iter convert hdefns in
301         {
302             main      = List.rev res.main;
303             functions = res.functions;
304             rho       = res.rho;
305             structures = res.structures
306         }

```

## 11.14 llgtype.ml

Author(s): Amy

```
1  (* llgtype.ml
2     Creates a context and puts types in it to use in the LLVM code. *)
3  module L = Llvm
4
5  (* creates the glocal context instance *)
6  let context = L.global_context ()
7
8  (* Add types to the context to use in the LLVM code *)
9  let int_ty      = L.i32_type  context
10 let char_ty      = L.i8_type  context
11 let char_ptr_ty = L.pointer_type char_ty
12 let bool_ty      = L.i1_type  context
13 let string_ty    = L.struct_type context [| L.pointer_type char_ty |]
14 let zero = L.const_int int_ty 0
15 (* REMOVE VOID later *)
16 (* let void_ty      = L.void_type context *)
17
18 let lltrue = L.const_int bool_ty 1
19
20 (* "tree_struct" will appear as the struct name in llvm code *)
21 let tree_struct_ty = L.named_struct_type context "tree_struct"
22 let tree_struct_ptr_ty = L.pointer_type tree_struct_ty
23 let () = L.struct_set_body
24     tree_struct_ty
25     [|
26         int_ty;
27         tree_struct_ptr_ty;
28         tree_struct_ptr_ty
29     |]
30     false
```

## 11.15 codegen.ml

Author(s): Amy, Eliza

```
1  (* Code Generation : Create a llmodule with llvm code from CAST *)
2  open Llgtype
3  open Cast
4
5
6  (* translate sdefns - Given an CAST (type cprog, a record type), the function
7     returns an LLVM module (llmodule type), which is the code generated from
8     the CAST. Throws exception if something is wrong. *)
9  let translate { main = main; functions = functions;
10                rho = rho; structures = structures } =
11
12
13  (* Convert Cast.ctype types to LLVM types *)
14  let ltype_of_type (struct_table : L.lltype StringMap.t) (ty : ctype) =
15    let rec ltype_of_ctype = function
16      Tycon ty -> ltype_of_tycon ty
17      | Conapp con -> ltype_of_conapp con
18    and ltype_of_tycon = function
19      Intty -> int_ty
20      | Boolty -> bool_ty
21      | Charty -> char_ptr_ty
22      | Tarrow ret -> ltype_of_ctype ret
23      | Clo (sname, _, _) ->
24        L.pointer_type (StringMap.find sname struct_table)
25    and ltype_of_conapp (tyc, ctys) =
26      let llretty = ltype_of_tycon tyc in
27      let llargtys = List.map ltype_of_ctype ctys in
28      L.pointer_type (L.function_type llretty (Array.of_list llargtys))
29    in ltype_of_ctype ty
30  in
31
32  (* Create an LLVM module (container into which we'll
33     generate actual code) *)
34  let the_module = L.create_module context "gROOT" in
35
36
37  (* DECLARE a print function (std::printf in C lib) *)
38  let printf_ty : L.lltype =
39    L.var_arg_function_type int_ty [| char_ptr_ty |] in
40  let printf_func : L.llvalue =
41    L.declare_function "printf" printf_ty the_module in
42
43  let puts_ty : L.lltype =
```

```

44     L.function_type int_ty [| char_ptr_ty |] in
45 let puts_func : L.llvalue =
46     L.declare_function "puts" puts_ty the_module in
47
48     (* Create a "main" function that takes nothing, and returns an int. *)
49 let main_ty = L.function_type int_ty [| |] in
50 let the_main = L.define_function "main" main_ty the_module in
51
52
53     (* create a builder for the whole program, start it in main block *)
54 let main_builder = L.builder_at_end context (L.entry_block the_main) in
55
56
57     (* Format strings to use with printf for our literals *)
58 let int_format_str = L.build_global_stringptr "%d\n" "fmt" main_builder
59
60     (* string constants for printing booleans *)
61 and boolT      = L.build_global_stringptr "#t" "boolT" main_builder
62 and boolF      = L.build_global_stringptr "#f" "boolF" main_builder
63 in
64
65 let zero = L.const_int int_ty 0 in
66 let print_true = L.build_in_bounds_gep boolT [| zero |] "" main_builder in
67 let print_false = L.build_in_bounds_gep boolF [| zero |] "" main_builder in
68
69     (* helper to construct named structs *)
70 let create_struct (name : cname) (membertys : ctype list)
71     (map : L.lltype StringMap.t) =
72     let llmembertys = List.map (ltype_of_type map) membertys in
73     let new_struct_ty = L.named_struct_type context name in
74     let () =
75         L.struct_set_body
76             new_struct_ty
77             (Array.of_list llmembertys)
78             false
79     in new_struct_ty
80 in
81
82
83     (* Declare the NAMED struct definitions *)
84 let struct_table : L.lltype StringMap.t =
85     let gen_struct_def map closure = match closure with
86         Tycon (Clo (name, anonFunTy, freetys)) ->
87             let v = create_struct name (anonFunTy :: freetys) map in
88             StringMap.add name v map
89     | _ -> Diagnostic.error

```

```

90         (Diagnostic.GenerationError "declaration of lambda that is non-closure type")
91     in
92     let structs = List.rev structures in
93     let rec declare_structs clsTyList result =
94         match clsTyList with
95         | [] -> result
96         | front :: rest ->
97             (try
98                 let result' = gen_struct_def result front in
99                 declare_structs rest result'
100             with Not_found ->
101                 let rest' = rest @ [front] in
102                 declare_structs rest' result)
103     in declare_structs structs StringMap.empty
104 in
105
106
107     (* Lookup table of function names (lambdas) to
108        (function block, function def) *)
109     let function_decls : (L.llvalue * fdef) StringMap.t =
110         let define_func map def =
111             let name = def.fname
112             and formal_types =
113                 Array.of_list (List.map (fun (t, _) -> ltype_of_type struct_table t)
114                                     (def.formals @ def.frees))
115             in
116             let ftype = L.function_type
117                 (ltype_of_type struct_table def.rettyp)
118                 formal_types
119             in StringMap.add name (L.define_function name ftype the_module, def) map
120         in List.fold_left define_func StringMap.empty functions
121     in
122
123
124     (* creates a global pointer to some variable's value *)
125     let create_global id ty =
126         let lltyp = ltype_of_type struct_table ty in
127         let rec const_ty = function
128             Tycon ty    -> const_tycon ty
129             (* / Tyvar tp    -> const_tyvar tp *)
130             | Conapp con -> const_conapp con
131         and const_tycon = function
132             Intty      -> L.const_int lltyp 0
133             | Boolty   -> L.const_int lltyp 0
134             | Charty   -> L.const_pointer_null lltyp
135             | Tarrow _  -> L.const_pointer_null lltyp

```

```

136 | Clo _ -> L.const_pointer_null lltyp
137 and const_conapp (_, _) = L.const_pointer_null lltyp
138 in
139 let init = const_ttyp ty in
140 L.define_global id init the_module
141 in
142
143 (* A lookup table of named globals to represent named values *)
144 let globals : L.llvalue StringMap.t =
145   let global_vars k occursList map =
146     let glo_var map' (num, typ) =
147       if num = 0
148       then map'
149       else let id = "_" ^ k ^ "_" ^ string_of_int num in
150             StringMap.add id (create_global id typ) map'
151     in List.fold_left glo_var map occursList
152   in
153   StringMap.fold global_vars rho StringMap.empty
154 in
155
156
157 (* Looks for variable named id in the local or global environment *)
158 let lookup id locals =
159   try StringMap.find id locals
160   with Not_found -> StringMap.find id globals
161 in
162
163 (* Add terminal instruction to a block *)
164 let add_terminal builder instr =
165   match L.block_terminator (L.insertion_block builder) with
166   | Some _ -> ()
167   | None -> ignore (instr builder) in
168
169
170 (* Construct constants code for literal values.
171   Function takes a builder and Sast.svalue, and returns the constructed
172   llvalue *)
173 let const_val builder = function
174   (* create the "string" constant in the code for the char *)
175   CChar c ->
176     let spc = L.build_alloca char_ptr_ty "spc" builder in
177     let globalChar =
178       L.build_global_string (String.make 1 c) "globalChar" builder in
179     let newStr = L.build_bitcast globalChar char_ptr_ty "newStr" builder in
180     let loc = L.build_gep spc [| zero |] "loc" builder in
181     let _ = L.build_store newStr loc builder in

```

```

182     L.build_load spc "character_ptr" builder
183   | CInt i -> L.const_int int_ty i
184   (* HAS to be an i1 lltype for the br instructions *)
185   | CBool b -> L.const_int bool_ty (if b then 1 else 0)
186   | CRoot _ -> Diagnostic.error
187       (Diagnostic.Unimplemented "codegen SRoot Literal")
188
189 in
190
191
192
193 (* Construct code for expressions
194    Arguments: llbuilder, lookup table of local variables,
195    current llblock, and Cast.cexpr.
196    Constructs the llvm where builder is located;
197    Returns an llbuilder and the llvalue representation of code. *)
198 let rec expr builder lenv block (etyp, e) =
199   match e with
200   | CLiteral v -> (builder, const_val builder v)
201   | CVar s ->
202       let varValue =
203         (try L.build_load (lookup s lenv) s builder
204          with Not_found -> Diagnostic.error
205              (Diagnostic.Unbound ("name \"" ^ s
206                                  ^ "\" not found in codegen"))))
207       in (builder, varValue)
208   | CIf (e1, (t2, e2), e3) ->
209       (* allocate space for result of the if statement *)
210       let result =
211         L.build_alloca (ltype_of_type struct_table t2) "if-res-ptr" builder
212       in
213
214       (* set aside the result of the condition expr *)
215       let (newbuilder, bool_val) = expr builder lenv block e1 in
216
217       (* Create the merge block for after exec of then or the else block *)
218       let merge_bb = L.append_block context "merge" block in
219       let branch_instr = L.build_br merge_bb in
220
221       (* Create the "then" block *)
222       let then_bb = L.append_block context "then" block in
223       let (then_builder, then_res) =
224         expr (L.builder_at_end context then_bb) lenv block (t2, e2) in
225       let _ = L.build_store then_res result then_builder in
226       let () = add_terminal then_builder branch_instr in
227

```



```

228      (* Create the "else" block *)
229      let else_bb = L.append_block context "else" block in
230      let (else_builder, else_res) =
231          expr (L.builder_at_end context else_bb) lenv block e3 in
232      let _ = L.build_store else_res result else_builder in
233      let () = add_terminal else_builder branch_instr in
234
235      (* Complete the if-then-else block, return should be llvalue *)
236      (* let _ = L.build_cond_br bool_val then_bb else_bb builder in *)
237      let _ = L.build_cond_br bool_val then_bb else_bb newbuilder in
238      (* Get the result of either the then or the else block *)
239      let merge_builder = L.builder_at_end context merge_bb in
240      let result_value = L.build_load result "if-res-val" merge_builder in
241      (merge_builder, result_value)
242  | CApply ((_, CVar "printfi"), [arg], _) ->
243      let (builder', argument) = expr builder lenv block arg in
244      let instruction = L.build_call printf_func
245          [| int_format_str ; argument |]
246          "printfi" builder'
247      in (builder', instruction)
248  | CApply ((_, CVar "printfc"), [arg], _) ->
249      let (builder', argument) = expr builder lenv block arg in
250      let instruction = L.build_call puts_func
251          [| argument |]
252          "printfc" builder'
253      in (builder', instruction)
254  | CApply ((_, CVar "printfb"), [arg], _) ->
255      let (builder', condition) = expr builder lenv block arg in
256      let bool_stringptr =
257          if condition = lltrue then print_true
258          else print_false in
259      let instruction = L.build_call puts_func
260          [| bool_stringptr |]
261          "printfb" builder'
262      in (builder', instruction)
263  | CApply ((_, CVar "~"), [arg], _) ->
264      let (builder', e) = expr builder lenv block arg in
265      let instruction = L.build_neg e "~" builder'
266      in (builder', instruction)
267  | CApply ((_, CVar "not"), [arg], _) ->
268      let (builder', e) = expr builder lenv block arg in
269      let instruction = L.build_not e "not" builder'
270      in (builder', instruction)
271  (* BINOP PRIMITIVES - Int and Boolean Algebra *)
272  | CApply ((_, CVar "+"), arg1::[arg2], _) ->
273      let (builder', e1) = expr builder lenv block arg1 in

```

```

274     let (builder'', e2) = expr builder' lenv block arg2 in
275     let instruction = L.build_add e1 e2 "addition" builder''
276     in (builder'', instruction)
277 | CApply ((_, CVar "-"), arg1::[arg2], _) ->
278     let (builder', e1) = expr builder' lenv block arg1 in
279     let (builder'', e2) = expr builder' lenv block arg2 in
280     let instruction = L.build_sub e1 e2 "subtraction" builder''
281     in (builder'', instruction)
282 | CApply ((_, CVar "/"), arg1::[arg2], _) ->
283     let (builder', e1) = expr builder' lenv block arg1 in
284     let (builder'', e2) = expr builder' lenv block arg2 in
285     let instruction = L.build_sdiv e1 e2 "division" builder''
286     in (builder'', instruction)
287 | CApply ((_, CVar "*"), arg1::[arg2], _) ->
288     let (builder', e1) = expr builder' lenv block arg1 in
289     let (builder'', e2) = expr builder' lenv block arg2 in
290     let instruction = L.build_mul e1 e2 "multiply" builder''
291     in (builder'', instruction)
292 | CApply ((_, CVar "mod"), arg1::[arg2], _) ->
293     let (builder', e1) = expr builder' lenv block arg1 in
294     let (builder'', e2) = expr builder' lenv block arg2 in
295     let instruction = L.build_srem e1 e2 "modulus" builder''
296     in (builder'', instruction)
297 | CApply ((_, CVar "&&"), arg1::[arg2], _) ->
298     let (builder', e1) = expr builder' lenv block arg1 in
299     let (builder'', e2) = expr builder' lenv block arg2 in
300     let instruction = L.build_and e1 e2 "logAND" builder''
301     in (builder'', instruction)
302 | CApply ((_, CVar "||"), arg1::[arg2], _) ->
303     let (builder', e1) = expr builder' lenv block arg1 in
304     let (builder'', e2) = expr builder' lenv block arg2 in
305     let instruction = L.build_or e1 e2 "logOR" builder''
306     in (builder'', instruction)
307 (* BINOP PRIMITIVES - Comparisons *)
308 | CApply ((_, CVar "<"), arg1::[arg2], _) ->
309     let (builder', e1) = expr builder' lenv block arg1 in
310     let (builder'', e2) = expr builder' lenv block arg2 in
311     let instruction = L.build_icmp L.Icmp.Slt e1 e2 "lt" builder''
312     in (builder'', instruction)
313 | CApply ((_, CVar ">"), arg1::[arg2], _) ->
314     let (builder', e1) = expr builder' lenv block arg1 in
315     let (builder'', e2) = expr builder' lenv block arg2 in
316     let instruction = L.build_icmp L.Icmp.Sgt e1 e2 "gt" builder''
317     in (builder'', instruction)
318 | CApply ((_, CVar "<="), arg1::[arg2], _) ->
319     let (builder', e1) = expr builder' lenv block arg1 in

```

```

320     let (builder'', e2) = expr builder' lenv block arg2 in
321     let instruction = L.build_icmp L.Icmp.Sle e1 e2 "leq" builder''
322     in (builder'', instruction)
323 | CApply ((_, CVar ">="), arg1::[arg2], _) ->
324     let (builder', e1) = expr builder' lenv block arg1 in
325     let (builder'', e2) = expr builder' lenv block arg2 in
326     let instruction = L.build_icmp L.Icmp.Sge e1 e2 "geq" builder''
327     in (builder'', instruction)
328 | CApply ((_, CVar "=i"), arg1::[arg2], _) ->
329     let (builder', e1) = expr builder' lenv block arg1 in
330     let (builder'', e2) = expr builder' lenv block arg2 in
331     let instruction = L.build_icmp L.Icmp.Eq e1 e2 "eqI" builder''
332     in (builder'', instruction)
333 | CApply ((_, CVar "!=i"), arg1::[arg2], _) ->
334     let (builder', e1) = expr builder' lenv block arg1 in
335     let (builder'', e2) = expr builder' lenv block arg2 in
336     let instruction = L.build_icmp L.Icmp.Ne e1 e2 "neqI" builder''
337     in (builder'', instruction)
338 | CApply (f, args, numFrees) ->
339     (* Since all normal function application is as struct value,
340      call to the function at member index 0 of the struct *)
341     let (builder', applyClosure) = expr builder' lenv block f in
342     (* List of llvalues representing the actual arguments *)
343     let (builder'', llargs) =
344         List.fold_left (fun (bld, arglist) arg ->
345             let (bld', llarg) = expr bld lenv block arg in
346             (bld', llarg :: arglist))
347         (builder', []) args
348     in let llargs = List.rev llargs in
349     (* List of llvalues representing the hidden frees to be
350      passed as arguments *)
351     let llfrees =
352         let rec get_free idx frees =
353             if idx > numFrees then List.rev frees
354             else
355                 let struct_freeMemPtr =
356                     L.build_struct_gep applyClosure idx "freePtr" builder'' in
357                 let struct_freeMem =
358                     L.build_load struct_freeMemPtr "freeVal" builder'' in
359                 let frees' = struct_freeMem :: frees
360                 in get_free (idx + 1) frees'
361         in get_free 1 []
362     in
363     (* Get the struct member at index 0, which is the function, call it *)
364     let ptrFuncPtr =
365         L.build_struct_gep applyClosure 0 "function_access" builder'' in

```

```

366     let funcPtr = L.build_load ptrFuncPtr "function_call" builder'' in
367     let instruction = L.build_call funcPtr
368       (Array.of_list (llargs @ llfrees))
369       "function_result" builder''
370     in (builder'', instruction)
371 | CLet (bs, body) ->
372   (* create each value in bs, and bind it to the name *)
373   let (builder', local_env) =
374     List.fold_left
375       (fun (bld, map) (name, (ty, cexp)) ->
376         let loc = L.build_alloca
377           (ltype_of_type struct_table ty) name bld in
378         let (bld', llcexp) = expr bld lenv block (ty, cexp) in
379         let _ = L.build_store llcexp loc bld' in
380         let map' = StringMap.add name loc map in
381         (bld', map'))
382       (builder, lenv) bs
383   (* Evaluate the body *)
384   in expr builder' local_env block body
385 | CLambda (id, freeargs)->
386   (match etyp with
387   | Tycon (Clo (clo_name, _, _)) ->
388     (* alloc and declare a new struct object *)
389     let struct_ty = StringMap.find clo_name struct_table in
390     let struct_obj = L.build_alloca struct_ty "gstruct" builder in
391     (* Set the function field of the closure *)
392     let struct_fmemb =
393       L.build_struct_gep struct_obj 0 "funcField" builder in
394     let (fblock, _) = StringMap.find id function_decls in
395     let _ = L.build_store fblock struct_fmemb builder in
396     (* Set each subsequent field of the frees *)
397     let (builder', llFreeArgs) =
398       List.fold_left
399         (fun (bld, arglist) arg ->
400           let (bld', llarg) = expr bld lenv block arg in
401           (bld', llarg :: arglist))
402         (builder, []) freeargs in
403     let llFreeArgs = List.rev llFreeArgs in
404     let numFrees = List.length freeargs in
405     let structFields =
406       (* Create pointers to struct members 1 to n *)
407       let rec generate_field_access idx fields =
408         if idx > numFrees then List.rev fields
409         else
410           let struct_freeMem =
411             L.build_struct_gep struct_obj idx "freeField" builder' in

```

```

412         let fields' = struct_freeMem :: fields in
413         generate_field_access (idx + 1) fields'
414     in generate_field_access 1 []
415 in
416 let _ =
417     let set_free arg field = L.build_store arg field builder'
418     in List.map2 set_free llFreeArgs structFields
419 in
420     (builder', struct_obj)
421 | _ -> Diagnostic.error
422     (Diagnostic.GenerationError "application of lambda that is non-closure type"))
423 in
424
425
426 (* generate code for a particular definition; returns an llbuilder. *)
427 let build_def builder = function
428     | CVal (id, (ty, e)) ->
429         (* assign a global define of a variable to a value *)
430         let (builder', e') = expr builder StringMap.empty the_main (ty, e) in
431         let _ = L.build_store e' (lookup id StringMap.empty) builder' in
432         builder'
433     | CExpr e ->
434         let (builder', _) = expr builder StringMap.empty the_main e in
435         builder'
436 in
437
438 (* procedure to generate code for each definition in main block.
439     Takes a current llbuilder where to put instructions, and a
440     list of definitions. Returns a builder of the last location
441     to put final instruction. *)
442 let rec build_main builder = function
443     [] -> builder
444     | front :: rest ->
445         let builder' = build_def builder front in
446         build_main builder' rest
447 in
448
449 let main_builder' = build_main main_builder main in
450 (* Every function definition needs to end in a ret.
451     Puts a return at end of main *)
452 let _ = L.build_ret (L.const_int int_ty 0) main_builder' in
453
454
455
456 (* Build function block bodies *)
457 let build_function_body fdef =

```

```

458 let (function_block, _) = StringMap.find fdef.fname function_decls in
459 let fbuilder = L.builder_at_end context (L.entry_block function_block) in
460
461 (* For each param, load them into the function body.
462   locals : llvalue StringMap.t*)
463 let locals =
464   let add_formal map (ty, nm) p =
465     let () = L.set_value_name nm p in
466     let local = L.build_alloca
467       (ltype_of_type struct_table ty) nm fbuilder in
468     let _ = L.build_store p local fbuilder in
469     StringMap.add nm local map
470   in
471   List.fold_left2
472     add_formal
473     StringMap.empty
474     (fdef.formals @ fdef.frees)
475     (Array.to_list (L.params function_block))
476   in
477
478 (* Build the return *)
479 let (fbuilder', result) = expr fbuilder locals function_block fdef.body in
480
481 (* Build instructions for returns based on the rettype *)
482 let build_ret t =
483   let rec ret_of_tyc = function
484     Tycon ty    -> ret_of_tycon ty
485     (* | Tyvar tp    -> ret_of_tyvar tp *)
486     | Conapp con -> ret_of_conapp con
487   and ret_of_tycon = function
488     Intty    -> L.build_ret result
489     | Boolty -> L.build_ret result
490     | Charty -> L.build_ret result
491     | Tarrow _ -> L.build_ret result
492     | Clo _    -> L.build_ret result
493   and ret_of_conapp (tyc, _) = ret_of_tycon tyc
494   in ret_of_tyc t
495   in
496   add_terminal fbuilder' (build_ret fdef.rettyp)
497   in
498
499 (* iterate through each function def we need to build and build it *)
500 let _ = List.iter build_function_body functions in
501
502 (* Return an llmodule *)
503 the_module

```

---

## 11.16 diagnostic.ml

Author(s): Zach

```
1  (* exception definition and pretty printing *)
2  (* maybe rename module to Diagnostic *)
3
4
5
6  (* character formatting functions *)
7  (* uses the \027 (ESC) ANSI escape codes *)
8
9  (*let red s = "\027[0m\027[31m" ^ s ^ "\027[0m"
10     let bold s = "\027[0m\027[1" ^ s ^ "\027[0m"*)
11  (*let red_bold s = "\027[0m\027[5;1;31m" ^ s ^ "\027[0m"
12
13     let purple_bold s = "\027[0m\027[1;35m" ^ s ^ "\027[0m"*)
14
15
16  (* Codes and explanations taken from:
17     https://stackoverflow.com/questions/4842424/list-of-ansi-color-escape-sequences
18  *)
19
20  (* character styling *)
21  let reset      = "0" (* all attributes off *)
22  let bold       = "1" (* bold or increased intensity *)
23  let faint      = "2" (* faint (decreased intensity) Not widely supported. *)
24  let italic     = "3" (* italic. Not widely supported. Sometimes treated as inverse. *)
25  let underline  = "4" (* underline text *)
26  let blink      = "5" (* slow Blink less than 150 per minute *)
27  let reverse    = "7" (* swap foreground and background colors *)
28
29  (* foreground colors *)
30  let fg_black   = "30"
31  let fg_red     = "31"
32  let fg_green   = "32"
33  let fg_yellow  = "33"
34  let fg_blue    = "34"
35  let fg_magenta = "35"
36  let fg_cyan    = "36"
37  let fg_white   = "37"
38
39  (* background colors *)
40  let bg_black   = "40"
41  let bg_red     = "41"
42  let bg_green   = "42"
43  let bg_yellow  = "43"
```



```

44 let bg_blue      = "44"
45 let bg_magenta   = "45"
46 let bg_cyan      = "46"
47 let bg_white     = "47"
48
49 (* preset effects lists for particular message types *)
50 let error_fx      = [fg_red;bold]
51 let warning_fx    = [fg_magenta;bold]
52 let note_fx       = [fg_cyan;bold;italic;underline]
53
54 let strfx fx str =
55   "\027[" ^ String.concat ";" fx ^ "m" ^ str ^ "\027[0m"
56
57 (* allowing for an exception to propagate will cause that pesky phrase
58  * "Fatal error: exception" to be printed; this avoids that
59  *)
60 let warning exn = try raise(exn) with _ -> prerr_endline (Printexc.to_string exn)
61 let error   exn = try raise(exn) with _ -> prerr_endline (Printexc.to_string exn); exit 1
62
63
64 exception LexingError    of string
65 exception ParsingError   of string
66 exception ParsingWarning of string
67
68 (* errors with positions, used while scanning and parsing *)
69 (* Courtesy of:
70    https://stackoverflow.com/questions/14046392/verbose-error-with-ocaml yacc
71    *)
72 let pos_fault msg (start : Lexing.position) (finish : Lexing.position) =
73   Printf.sprintf "(line %d, col %d-%d): %s" start.pos_lnum
74     (start.pos_cnum - start.pos_bol) (finish.pos_cnum - finish.pos_bol) msg
75
76 let lex_error msg lexbuf =
77   LexingError ((pos_fault "(" ^ (Lexing.lexeme lexbuf) ^ ")") (Lexing.lexeme_start_p lexbuf) (Lexing.lexeme_end_p lexbuf) msg)
78
79 let parse_error msg nterm =
80   ParsingError (pos_fault msg (Parsing.rhs_start_pos nterm) (Parsing.rhs_end_pos nterm))
81
82 let parse_warning msg nterm =
83   ParsingWarning (pos_fault msg (Parsing.rhs_start_pos nterm) (Parsing.rhs_end_pos nterm))
84 (* end courtesy of *)
85
86 exception Unimplemented of string
87 exception Unbound of string
88 exception EmptyLetBinding
89 exception TypeError of string

```

```

90 exception GenerationError of string
91 exception MonoError of string
92 exception MonoWarning of string
93 let () =
94   Printexc.register_printer (function
95     | Unimplemented   s -> Some ((strfx error_fx "Unimplemented Error: ") ^ s)
96     | Unbound         s -> Some ((strfx error_fx "Unbound Error: "      ) ^ s)
97     | TypeError       s -> Some ((strfx error_fx "Type Error: "          ) ^ s)
98     | GenerationError s -> Some ((strfx error_fx "Generation Error: "    ) ^ s)
99     | MonoError       s -> Some ((strfx error_fx "Mono Error: "           ) ^ s)
100    | MonoWarning     s -> Some ((strfx warning_fx "Polymorphic Warning: ") ^ s)
101    | ParsingWarning  s -> Some ((strfx warning_fx "Parsing Warning: "     ) ^ s)
102    | LexingError     s -> Some ((strfx error_fx "Lexing Error: "          ) ^ s)
103    | ParsingError    s -> Some ((strfx error_fx "Parsing Error: "        ) ^ s)
104    | _ -> None)
105
106
107
108 (*let () = (strfx) : string list * string -> string*)
109 (*let () = print_string ((strfx [bg_black;fg_white;blink] " hello ") ^ "\n")*)
110 (*let () =
111   Printexc.register_printer (function
112     | Unimplemented s -> Some ("\027[1;31mUnimplemented:\027[0m " ^ s)
113
114     | _ -> None)
115 *)

```

## 11.17 git etiquette

Author(s): Nik

1 Saturday, February 12, 2022

2  
3 git - Best Practices

4 Commits

5 Commits should be small but frequent | whenever you make a logical change to the code commit and write a

6 Commit messages include:

7 Why the code was updated/changed

8 What has been altered

9 Branching

10 Every branch should represent a feature we are working on, we will have:

11 A main branch | where the submission version of our code is stored

12 A development branch | where we will test all the new features as one before merging with main branch (i

13 Multi-feature branches | these can be the different keywords we are implementing, or other parts of the c

14 Tags

15 When we are ready for submission we will tag the code with the name of the deliverable we are submitting

16 This lets us:

17 Keep track of each version of our project (by version I mean completed deliverable)

18 We can compare the differences between one version of the project and another

19 Can record what the version includes and it is easy to go back to that version if ever we run into troub

20  
21  
22  
23  
24 Workflow

25 Status check | Where am I?

26 git status | shows changed files that are staged and unstaged

27 git log | shows git history

28  
29 Branch lifecycle

30 git checkout -b <branch\_name> | create a new branch

31 git checkout <branch\_name> | check out an available branch

32 git fetch | if you want to check out a branch that is only available on the remote repository because a

33 Keeps all available branches in sync with the local machine but NOT the commits to those branches, this

34 git branch -d <branch\_name> | delete a branch on your local machine

35 git push origin -d <branch\_name> | delete a branch on the remote

36 Be careful when deleting a branch on the remote repository because most likely the branch will want to b

37 After we merge we can rest easy and delete the branch if we want

38  
39 Feature Branch

40 Like mentioned above create a new feature branch from the development branch using the command: git chec

41 Then you will want to implement whatever you plan to implement. Once finished run the following commands

42 git add . |moves all changed/added/deleted files to the staging area for the next commit, the \." means

43 There are variations of this command, for example, you can choose to just move all changed files to the

```
44
45 git commit -m \<commit_message>"
46 Message Convention:
47 <type>(<which_file_or_domain>) <detailed_comment>
48 Type:
49 feat | feature implementation
50 style | code style, cleaning up the look of the code
51 test | test implementation
52 fix | fixing a bug
53 chore | no code changes but changing git configuration | like .gitignore
54 git push origin <branch_name> | pushes all committed changes to the remote repository
55
56 Keeping branch up-to-date
57 You will eventually need to update your local branch, this is how:
58 As mentioned above if the branch is not available locally to you yet use the command: git fetch then nav
59 Note: if you have already made changes commit them first
60 git pull |rebase origin <branch_name> | this is a little different from git pull, from what I have read
61 If you run into merge conflicts we will need to resolve them, afterwards run the commands to keep the hi
62 git add .
63 git rebase |continue
64
65
66
67 Merging the feature branch
68 git checkout development | move to development branch
69 git merge |no-ff <branch_name> | similar to git merge but it keeps the history of commits if they were n
70 git push origin development | pushes the merged development branch to the repository
```

---