Compost Final Report

December 15, 2023

```
val name_email_map : (string * string) list =
[("Roger Burtonpatel", "roger.burtonpatel@tufts.edu");
 ("Randy Dang", "randy.dang@tufts.edu");
 ("Jasper Geer", "jasper.geer@tufts.edu");
 ("Jackson Warhover", "jackson.warhover@tufts.edu")]
```

Contents

1 Introduction

Compost is a statically-typed pure functional programming language with an affine type system. That is, the type system guarantees that no two live references ever exist to the same heap object. Programs in Compost include no explicit memory management and run without the need for a runtime garbage collector. This is because in a manner akin to Rust, the Compost compiler performs compile-time memory management, inserting memory-freeing directives and guaranteeing memory safety for all Compost programs.

In order to make this guarantee, we must place one major restriction on the programmer to ensure that the compiler is performing a decidable task: each variable can be used at most once in a given scope. That is, if a variable **could** have been referenced already in the current scope, the programmer is not allowed to reference it again. When we enforce this restriction, we can determine the point at which a variable in scope will not be used and insert free directives accordingly.

A memory safe language is useful because it guarantees that memory-related bugs will never be introduced by programmers; any such errors would be caught by the compiler ahead of time. This is an especially handy feature when writing implementations of critical systems (such as medical devices) where memory-related bugs could potentially be very costly. The lack of a need for automatic garbage collection also leads to better performance.

2 Language Tutorial

This section contains a brief tutorial of how to write simple programs in Compost utilizing some of the more important features. For a full specification of the language, see the Language Manual.

2.1 Notational Convention

In this section and the rest of this document, code listings will appear in "verbatim" as follows:

```
(define foo ()
    bar)
```

2.2 Compost Basics

Compost is a parenthesized functional language with a syntax similar to Scheme syntax, but it is a compiled language rather than interpreted. A Compost program consists of a sequence of definitions, which mainly include preprocessor macros, function definitions, and custom datatype definitions. For every function definition, there must also exist a type annotation that defines the argument and return types of that function (the argument types are surrounded by a single pair of parentheses and listed before the return type). The function with name `main` defines the entry point of the program, and it must take in no arguments and return type `unit`.

Function definitions are specified via the `define` keyword. Here is a program that simply prints the string "Hello, World!" (`print-sym` is a built-in function that takes in a single symbol argument and prints it to stdout, and any character between a semicolon and the end of a line, inclusive, is part of a comment).

```
(: main (-> () unit)) ;; type annotation: defines `main' as function taking no
                      ;; arguments and returning type `unit'
(define main ()                   ;; definition for `main' function, the entry
                                  ;; point of the program
    (print-sym 'Hello, World!'))  ;; prints out `Hello, World' symbol
```

Preprocessor macros are specified via the `val` keyword and can improve code readability and/or reduce code duplication. This program uses a preprocessor macro to accomplish the same functionality as above:

```
(val hello-str 'Hello, World!') ;; defines the name `hello-str' as the symbol
                                ;; 'Hello, World!'
                                ;; This is analogous to the following in C:
                                ;; #define hello-str "Hello, World!"
(: main (-> () unit)) ;; type annotation
(define main () ;; program entry point
    (print-sym hello-str))  ;; prints out `Hello, World' symbol
```

Functions are called in the same manner as they are in Scheme. Here is an example of the definiton of a function `compute` that performs arithmetic on two numbers and a `main` function that calls compute, passing in 2 and 3 as arguments.

```
(: compute (-> (int int) int)) ;; type annotation: defines `compute' as a
                               ;; function taking in two ints as arguments
                               ;; and returns an int
(define compute (x y)
    (+ (* x 2) y)) ;; multiply x by 2 and add y. Prefix arithmetic operators
                   ;; are built-in

(: main (-> () unit)) ;; type annotation
(define main () ;; program entry point
    (print-int (compute 2 3))) ;; prints the result of calling `compute' on
                               ;; the numbers 2 (bound to `x') and 3 (bound
                               ;; to `y'), as an integer. Result should be
                               ;; 7.
```

2.3 Custom Datatypes

The most interesting functionality provided to the user is the ability to define and use custom abstract data types. Such datatypes can be defined with the `datatype` keyword and the definitions of one or more variant constructors, which define ways that instances of that datatype can be created. For example, a linked list of integers can be defined as follows:

```
;; Definition of linked list of integers, which can be constructed in two
;; ways (one defines the case of a non-empty list, and the other defines
;; the case of an empty list)
(datatype int-list
    ([cons-int (int int-list)] ;; Variant constructor 1: create a non-empty
                               ;; int-list by applying `cons-int' to an int
                               ;; and another int-list.
     [nil-int-list ()]))       ;; Variant constructor 2: create an empty
                               ;; int-list by applying `nil-int-list' to
                               ;; nothing
```

If this datatype definition exists somewhere in the program, then `int-list` exists as a type and both `cons-int` and `nil-int-list` exist as constructors that can be called.

For example, a three-element linked list can be constructed as follows:

```
;; macro that constructs linked list with elements: [0, 1, 2]
(val len3list (cons-int 0 (cons-int 1 (cons-int 2 (nil-int-list)))))
```

To “unpack” the components of a custom datatype within a function, we support top-level pattern matching on the variant constructor definitions via `case` expressions. For example, below is a function that gets the length of an `int-list`.

```
;; Gets length of int-list `xxs' in terms of number of elements
(: len-int-list (-> (int-list) int)) ;; type annotation: takes in an int-list
                                     ;; as input and returns an int
(define len-int-list (xxs) ;; binds argument to name `xxs'
    (case xxs ;; begins pattern matching on the int-list `xxs'
        ([(cons-int x xs) ;; specify non-empty case with appropriate variant
                          ;; constructor
            (+ 1 (len-int-list xs))] ;; expression to evaluate in non-empty
                                     ;; case (add 1 to length of sub-list `xs')
         [(nil-int-list) ;; specify empty case with appropriate variant
                         ;; constructor
            0]))) ;; expression to evaluate in empty case (length is just 0)
```

If we wanted to print the length of `len3list` in our driver, we can do so as follows:

```
(: main (-> () unit)) ;; type annotation
(define main () ;; program entry point
    (print-int (len-int-list len3list))) ;; prints number of elements in
                                         ;; `len3list'. Should print 3.
```

2.4 How to Use Compiler

To use our compiler to compile Compost code, there should be a script called `gcc` (which stands for "Good Compost Compiler") in the top-level directory. Ensure that `cc` is symlinked to some version of `clang`, and simply execute that script as such:

```
./gcc file.com
```

where `file.com` is the name of a file containing a Compost program. An executable with the same name but the extension removed (`file` in the above case) will appear in the same directory as the Compost program. Run the compiled executable with:

```
./file
```

With the above example, our `gcc` script internally runs the following command:

```
dune exec compost file.com | llc -relocation-model=pic | cc -x assembler -o file -
```

3 Language Manual

3.1 Introduction

This language reference manual contains a formal description of Compost's syntax, along with an informal description of its semantics and type system. In addition, an initial basis for Compost programs is outlined.

3.2 More Notational Conventions

Grammar rules are written in extended Backus-Naur format, as follows:

rule	::=	(*nonterminal* `terminal`)
	|	{ *other-rule* }

Parentheses are concrete syntax, but any pair of balanced parentheses may be freely exchanged for a pair of square brackets. For example, the following two declarations are indistinguishable in the abstract syntax:

```
(val x 1)
```

```
[val x 1]
```

Note that braces are used in a manner akin to the Kleene closure, that is, a term enclosed in braces may be omitted or arbitrarily repeated.

3.3 Lexical Conventions

3.3.1 Whitespace

The following characters are considered as whitespace and, with one exception, ignored during tokenization: spaces, tabs, carriage returns, and newlines.

3.3.2 Comments

Comments are introduced by the character `;` and terminated by the newline character. Comments are treated as whitespace.

3.3.3 Literals

literal	::=	*integer-literal*
	|	*symbol-literal*
	|	*boolean-literal*
	|	*unit-literal*

Literals introduce values of Compost's primitive types. All literals are valid expressions.

3.3.4 Integer Literals

integer-literal	::=	token composed only of digits, possibly prefixed with a `+` or `-`.

The `+` prefix denotes a positive integer and the `-` prefix denotes a negative integer. The characters `1 2 3 4 5 6 7 8 9 0` are considered digits.

3.3.5 Symbol Literals

symbol-literal	::=	`'`{ *symbol-character* }`'`
symbol-character	::=	any unicode code point other than `'` and the backslash character unless escaped with a backslash.

That is, any sequence of `'`-delimited unicode characters is a valid symbol literal, as long as every instance of `'` or backslash are preceded by a backslash. This includes characters that would otherwise be treated as whitespace if they were found outside of the symbol literal setting. Escape sequences are replaced by their unescaped counterparts in the introduced symbol value. For example, the following are valid symbol literals:

```
'\'hello, world\''

'\\ is a backslash'

'I exist
on multiple lines'
```

The following are *not* valid symbol literals:

```
'Pini's Pizzeria' ;; the apostraphe should be escaped

'\ is missing an escape backslash'

'This is not a newline character: \n' ;; see above for proper usage of
                                      ;; multi-line strings
```

3.3.6 Other Literals

boolean-literal	::= `true` | `false`
unit-literal	::= `unit`

3.3.7 Reserved Words

The following tokens are considered reserved:

```
; ( ) [  ] : _ -> if val define datatype use case begin let dup int bool sym unit
```

3.4 Values

This section describes the kinds of values manipulated by Compost programs.

3.4.1 Integers

Integer values are 32-bit signed integers with a range of -2,147,483,648 to 2,147,483,647.

3.4.2 Symbols

Symbol values are interned immutable strings of unicode characters.

3.4.3 Booleans

Boolean values are either the boolean `true` or the boolean `false`.

3.4.4 Unit

Unit values are the value `unit`.

3.4.5 Variant Values

Variant values are either a constant constructor or a non-constant constructor applied to a series of value arguments. We write an arbitrary constant constructor c as (c) and an arbitrary non-constant constructor d applied to arguments $v_1 ... v_n$ as $(d\ v_1\ ...\ v_n)$.

Variant constructors are monomorphic, that is, for any constructor c, there exist types $\tau_1 ... \tau_n$ such that for any application of constructor c to arguments $v_1 ... v_n$, v_i must have type τ_i for all $i \in 1, 2, ..., n$.

3.4.6 Functions

Functions in Compost are globally defined objects. They can be passed in to other functions or returned from functions. Each function value is a mapping from an ordered set of values to a single value. That is, a function f, when applied to values $v_1 ... v_n$, produces a value v_r. Like variant constructors, functions are monomorphic, so the types of $v_1 ... v_n$ and v_r are fixed.

3.5 Names

Compost places relatively liberal constraints on the sequences of characters considered valid names.

name	::=	any token that is not an *int-lit*, does not contain whitespace (including a `;` character indicating the start of a comment), a `'`, bracket, or parenthesis, and is not a reserved word.

Names are bound to datatypes, functions, values, and variant constructors, and are used to refer to them at various points in a program.

3.6 Type Expressions

type-expression	::=	*function-type*
	|	*int-type*
	|	*bool-type*
	|	*sym-type*
	|	*unit-type*
	|	*datatype*

3.6.1 Primitive Types

int-type	::=	`int`
bool-type	::=	`bool`
sym-type	::=	`sym`
unit-type	::=	`unit`

`int` is the type of integer values.

`bool` is the type of boolean values.

`sym` is the type of symbol values.

`unit` is the type of unit values.

3.6.2 Function Types

function-type ::= `(->` ({ *type* }) *type*)

`(-> (t1 ... tn) tr)` is the type of function values which each map an ordered set of values $v_1 ... v_n$ of types `t1`...`t`n to value v_r of type `tr`.

3.6.3 Datatypes

datatype ::= *name*

Datatypes are the types of variant constructor values. Multiple variant constructors may share the same type. Datatypes and their constructors can be defined by the programmer with the following syntax:

datatype-definition ::= (`datatype` *name* ({ *variant-constructor-definition* }))

variant-constructor-definition ::= (*name* ({ *type-expression* }))

A *name* bound to the new type τ_d appears directly following the `datatype` keyword, and this is followed by a list of variant constructor definitions. Each of these provides a *name* bound to the constructor, c, followed by a list of *type-expression*s $\tau_1 ... \tau_n$ typing its arguments. Given this definition, a variant value (c v_1 ... v_n) of type τ_d may be introduced by applying function value c to $v_1 ... v_n$, where the type of v_i is τ_i for all $i \in 1, 2, ..., n$

The placement of a datatype or variant constructor's definition has no bearing on where it can be referenced, introduced, or eliminated. In fact, datatypes may be defined recursively, as in the following example:

```
(datatype int-list
  ([cons (int int-list)]
    [nil ()]))
```

This declaration can be read as: "an `int-list` is either `cons` applied to an `int` and an `int-list`, or `nil` applied to nothing".

3.7 Expressions

expr ::= *literal*
| *case-expression*
| *if-expression*
| *begin-expression*
| *apply-expression*
| *let-expression*
| *dup-expression*
| *name-expression*

Meaningful computation is encoded in Compost as *expr* syntactic forms, or expressions. These appear either as the right-hand side of `val` definitions (i.e. preprocessor macros) or as the bodies of functions.

We describe the semantics and typing rules of expressions largely informally but use formal notation to aid conciseness. Expressions are evaluated in an environment ρ mapping names to values. Initially, these environments contain the values and types of all globally bound names (functions and `val`-bound names). $\rho[x \mapsto v]$ is the modified environment ρ in which name x is bound to value v. $\rho[x]$ is the value mapped to by x in ρ.

There also exists a typing environment Γ mapping names to types. The same syntax is used to add bindings to Γ and denote the type mapped to by a name x. We also introduce a typing judgement $\Gamma \vdash e : \tau$ which can be read as "expression e has type τ in context Γ". When Γ is used in a subsection, it refers to the environment in which that particular expression is typed, rather than the initial typing environment. This typing judgement is defined inductively on the structure of expressions by the following subsections.

Certain expressions will "consume" names, effectively moving them out of scope. As a rule of thumb, any name that can be consumed can only be consumed once in a given program path of execution. Any names considered as consumed in a subexpression are considered consumed in the parent expression. Consumption is defined inductively on the structure of expressions by the following subsections.

Side effects are produced in evaluation order except in the case of `val`-bound names, which produce their associated expression's side effects at **every** reference.

3.7.1 Case Expressions

case-expression ::= (`case` *expr* ({ *case-branch* }))

case-branch ::= (*pattern expr*)

pattern ::= (*name* { *name* | _ })
| _

Note that we refer to instance of _ in patterns as "wildcards". Values of the form (c v_1 .. v_n) are eliminated by the *case-expression* syntactic form. Consider a case expression with n branches of the form:

```
(case e
  ([(c1 v11 v12 ...) e1]
    ...
    [(cn vn1 vn2 ...) en]))
```

Typing

We assert that the type of `e` must be a datatype. Suppose that $\Gamma \vdash \texttt{e} : \tau_d$. Each `c`$i$ must be a variant constructor of τ_d. For all $i \in 1, 2, ..., n$, let $\tau_{i1}, \tau_{i2}, ..., \tau_{im}$ be the types of `c`i's m arguments. We assert that the number of names and wildcards following `c`i must be precisely m. If any one of these names is not fresh (i.e. is already bound in a larger scope), then it shadows the existing binding in expression `e`i. Let Γ_i be $\Gamma[\texttt{v}i1 \mapsto \tau_{i1}, ..., \texttt{v}im \mapsto \tau_{im}]$. Note that wildcards are not bound. We assert that $\Gamma_i \vdash \texttt{e}i : \tau_r$. The type of the full `case` expression in context Γ is τ_r.

Consumption
Names marked as consumed in `e` are marked as consumed in all `e`i. Names marked as consumed in any `e`i are marked as consumed in the full `case` expression, but are *not* marked as consumed in any `e`j where $j \in 1, 2, ..., n$ and $j \neq i$.

Evaluation
Suppose evaluation of `e` in environment ρ yields a value $v = (c\ v_i\ ...\ v_m)$. If there exists some branch whose pattern is prefixed by c, it is evaluated in the environment ρ and its result is returned. Otherwise, a the program halts with a runtime error.

Suppose this branch is the *case-branch* containing the pattern prefixed by variant constructor `c`k. Evaluation of this branch yields the result of evaluating `e`k in the modified environment $\rho' = \rho[\texttt{v}k1 \mapsto v_1, ..., \texttt{v}km \mapsto v_m]$. Note that we do not bind wildcards in ρ'.

3.7.2 If Expressions

if-expression ::= (`if` *expr expr expr*)

Consider an if expression of the form:

```
(if e1 e2 e3)
```

Typing
We assert that $\Gamma \vdash \texttt{e1} : \texttt{bool}$. We further assert that $\Gamma \vdash \texttt{e2} : \tau_r$ and $\Gamma \vdash \texttt{e3} : \tau_r$. The type of the full `if` expression in context Γ is τ_r.

Consumption
Names marked as consumed in `e1` are marked as consumed in `e2` and `e3`. Names marked as consumed in any of `e1`, `e2`, or `e3` are marked as consumed in the full `if` expression, but names marked as consumed in `e2` are *not* marked as consumed in `e3`.

Evaluation
Suppose the evaluation of `e1` in environment ρ yields a boolean value v. If v is the value `true`, the expression `e2` is evaluated in environment ρ and its result is returned. If v is the value `false`, the expression `e3` is evaluated in environment ρ and its result is returned.

3.7.3 Begin Expressions

begin-expression ::= (`begin` { *expr* })

Consider a begin expression of the form:

```
(begin e1 ... en)
```

Typing
The type of this expression is the type of `en`.

Consumption
For all $i \in 1, 2, ..., n-1$, names marked as consumed in `e`i are marked as consumed in `e`$(i+1)$.

Evaluation
Each `e`i is evaluated in environment ρ in order from $1...n$. We return the result of evaluating `e`n in environment ρ.

3.7.4 Apply Expressions

apply-expression ::= (*expr* { *expr* })

Consider an apply expression of the form:

```
(e e1 ... en)
```

Typing
We assert that $\Gamma \vdash$ `e` : `(->` `(`t_1 ... t_n`)` t_r`)`. Each `e`i must be of type t_i for $i \in 1, 2, ..., n$. The type of this apply expression is t_r.

Consumption
For all $i \in 1, 2, ..., n-1$, names marked as consumed in `e`i are marked as consumed in `e`$i+1$.

Evaluation
Each `e`i is evaluated in environment ρ in order from $1...n$. Let $v_1...v_n$ be the values returned by evaluating each `e`i.

We return the result of applying `e` to arguments $v_1...v_n$.

3.7.5 Let Expressions

let-expression ::= (`let` ({ *let-binding* }) *expr*)

let-binding ::= (*name* *expr*)

Consider a let expression of the form:

```
(let
  ([x1 e1]
    ...
    [xn en])
  e)
```

Typing
Given that $\Gamma_k : \texttt{x}k : \tau_k$, for $k \in 1, 2, ..., n$, we say that $\Gamma_{k+1} = \Gamma_k[\texttt{x}k \mapsto \tau_k]$. As a base case, let $\Gamma_1 = \Gamma$. The type of this `let` expression is the type of `e` in context Γ_{n+1}.

Consumption
For any $i \in 1, 2, ..., n$ we mark any names consumed in $\texttt{e}i$ as consumed in both `e` and all $\texttt{e}k$ for $k > i$.

Evaluation
For all $i \in 1, 2, ..., n$, let $\rho_{i+1} = \rho_i[\texttt{x}i \mapsto v_i]$, where v_i is the value returned by evaluating $\texttt{e}i$ in environment ρ_i. As a base case, let $\rho_1 = \rho$. We return the result of evaluating `e` in environment ρ_{n+1}.

3.7.6 Name Expressions

name-expression ::= *name*

Consider a name expression of the form:

```
n
```

Typing
We assert that `n` be bound in Γ. We further assert that `n` not be marked as consumed. The type of this expression is $\Gamma[\texttt{n}]$.

Consumption
If the type of `n` in context Γ is a datatype, it is marked as consumed.

Evaluation
We return the value $\rho[\texttt{n}]$.

3.7.7 Dup Expressions

dup-expression ::= (`dup` *name*)

Consider a dup expression of the form:

```
(dup n)
```

Typing
We assert that `n` be bound in Γ. We further assert that `n` not be marked as consumed. The type of this expression is $\Gamma[\texttt{n}]$.

Consumption
`n` is **not** marked as consumed.

Evaluation
We return the value $\rho[\texttt{n}]$. If $\Gamma[\texttt{n}]$ is a datatype, the returned value is a deep copy.

3.8 Definitions

Syntactic forms in the *def* category are allowed only at the top level of a Compost program.

def	::=	*val-binding*
	|	*function-definition*
	|	*datatype-definition*
	|	*type-annotation*
	|	*use-declaration*

We retain the environment notation conventions from the previous subsection.

Compost maintains a global Γ_g and ρ_g which are mutated by type annotations, `val` bindings, and function definitions. Additional bindings may be added to these environments at code points. A change to either of these global environments at a given code point is reflected at all succeeding code points. To determine the initial Γ or ρ at a `val` binding or function definition, we take the Γ_g and ρ_g at its opening parenthesis.

3.8.1 Type Annotations

type-annotation ::= `(:` *name type-expression*`)`

Type annotations constrain the type of globally bound function names. Each such function name must have an associated type annotation. Consider a type annotation of the form:

```
(: n t)
```

We bind `n` to `t` in Γ_g at the first character of the file, i.e. the entire program has access to this binding regardless of where the the function `n` is defined.

3.8.2 Val Bindings

val-binding ::= `(val` *name expr*`)`

Consider a `val` binding of the form:

```
(val x e)
```

Let Γ, ρ be Γ_g, ρ_g at the opening parenthesis of the binding. Let Γ_c be Γ_g at the closing parenthesis of the binding.

We assert that `x` be free in ρ and bound in Γ_g. Given $\Gamma_c[\texttt{x}] = \tau$, we assert that $\Gamma \vdash \texttt{e} = \tau$.

Let v be the result of evaluating `e` in environment ρ. We bind `x` to v in ρ_g at the closing parenthesis.

Note that if `e` produces a side effect, it is produced **only** when `x` is referenced and **every** time `x` is referenced. That is, references to `val`-bound names behave as zero-arity function calls rather than references to `let`-bound names. The secret sauce here is that `val` bindings are simply macros.

3.8.3 Function Definitions

function-definition ::= (`define` *name* ({ *name* }) *exp*)

Consider a function definition of the form:

```
(define x (x1 ... xn) e)
```

Let Γ, ρ be Γ_g, ρ_g at the opening parenthesis of the binding.

We assert that `x` be bound in Γ and free in ρ. We assert that $\Gamma[\texttt{x}]$ = `(-> (`τ_1 ... τ_n`)` τ_r`)`. We assert that $\Gamma[\texttt{x1} \mapsto \tau_1, ..., \texttt{x}n \mapsto \tau_n] \vdash \texttt{e} : \tau_r$.

We bind `x`, in ρ_g at the first character of the file, to the function value that, when applied to arguments $v_1, ..., v_n$, returns the result of evaluating `e` in the environment $\rho[\texttt{x1} \mapsto v_1, ..., \texttt{x}n \mapsto v_n]$.

3.8.4 Use Declarations

use-declaration ::= (`use` *symbol-literal*)

Use declarations are thinly-veiled preprocessor directives which are replaced by the contents of the file whose path is specified as a symbol literal. The path must be hard-coded relative to the location where the compiler is run.

3.9 The Structure of Compost Programs

program ::= { *def* } *end-of-file*

Compost programs consist of a series of definitions. All executable programs must contain a function `main` of type `(-> () unit)`, which serves as the entry point for the program.

When a compiled Compost program is executed, `main` is invoked. The program terminates when `main` has been fully evaluated.

3.10 Initial Basis

Compost includes an initial basis providing those functions not possible or practical to define in terms of the rest of the core Compost language. A type annotation and description will be provided for each such function.

3.10.1 Equality

```
(: =i (-> (int int) bool))
```

Integer equality.

```
(: =b (-> (bool bool) bool))
```

Boolean equality.

```
(: =s (-> (sym sym) bool))
```

Symbol equality (recall that symbols are interned so physical and structural equality are one in the same).

```
(: =u (-> (unit unit) bool))
```

Unit equality. Always returns `true`.

3.10.2 Arithmetic

```
(: + (-> (int int) int))
```

Two's complement addition.

```
(: - (-> (int int) int))
```

Two's complement subtraction.

```
(: * (-> (int int) int))
```

Two's complement multiplication.

```
(: / (-> (int int) int))
```

Two's complement signed division.

```
(: % (-> (int int) int))
```

Two's complement signed modulus.

```
(: udiv (-> (int int) int))
```

Converts both of its arguments to 32-bit unsigned integers, performs unsigned division, and returns the result as a two's complement signed integer.

```
(: umod (-> (int int) int))
```

Converts both of its arguments to 32-bit unsigned integers, performs unsigned modulus, and returns the result as a two's complement signed integer.

```
(: neg (-> (int) int))
```

Two's complement negation.

3.10.3 Integer Comparison

`(: > (-> (int int) bool))`

Returns `true` if the first argument is greater than the second. Returns `false` otherwise.

`(: < (-> (int int) bool))`

Returns `true` if the first argument is less than the second. Returns `false` otherwise.

`(: >= (-> (int int) bool))`

Returns `true` if the first argument is greater than or equal to the second. Returns `false` otherwise.

`(: <= (-> (int int) bool))`

Returns `true` if the first argument is less than or equal to the second. Returns `false` otherwise.

3.10.4 Boolean Logic

`(: not (-> (bool) bool))`

Logical NOT.

`(: and (-> (bool bool) bool))`

Logical AND.

`(: or (-> (bool bool) bool))`

Logical OR.

`(: xor (-> (bool bool) bool))`

Logical XOR.

3.10.5 Bitwise Operators

`(: & (-> (int int) int))`

Bitwise AND.

`(: | (-> (int int) int))`

Bitwise OR.

`(: ^ (-> (int int) int))`

Bitwise XOR.

`(: << (-> (int int) int))`

Left bit shift first argument by second argument.

`(: >> (-> (int int) int))`

Right bit shift first argument by second argument.

`(: ~ (-> (int) int))`

Bitwise NOT, i.e. bitwise complement.

3.10.6 I/O: Printing

The following functions print representations of primitive values to stdout.

```
(: print-int (-> (int) unit))
```

Prints the digits of the decimal representation of the absolute value of its argument in order from most to least significant, prefixed with a `-` if it is less than `0`.

```
(: print-bool (-> (bool) unit))
```

Prints `true` to if its argument is the value `true` and prints `false` otherwise.

```
(: print-sym (-> (sym) unit))
```

Prints its symbol argument's associated string.

```
(: print-unit (-> (unit) unit))
```

Prints `unit`.

```
(: print-newline (-> () unit))
```

Prints a single newline character.

```
(: print-ascii (-> (int) unit))
```

Mods its argument by 256 and prints the ASCII character representation of the result.

3.10.7 I/O: Input

```
(: in (-> () int))
```

Returns the integer representation of a single ASCII character read from stdin.

4 Project Plan

4.1 Planning and Development Process

Most of the initial planning was done in "brainstorming sessions" that we held as a team early in the semester where we would discuss ideas for interesting features and the feasibility of such features (e.g. whether implementing such a feature would be decidable). We often made plans according to the course deliverable schedule, ensuring that we made language and design decisions by the time the relevant deliverable was due. Large-scale decisions were decided as a group, and decisions that affected only one or two passes of the compiler were made by the person or people assigned to those passes. Our language guru served as the ground truth for any decisions involving language-specific semantics.

We generally met as a full team about once every 1-2 weeks where we got caught up on each other's progress, held semantic and architectural debates, and made plans for next steps. We decided pretty early on that unlike microC's implementation, we wanted to implement our compiler in more than two passes. During the weeks before the deadline of the "Hello World" deliverable, we spent a fair amount of time up front hashing out the functionality of each pass and what we wanted the intermediate representation to look like between each pass. Once we had each IR explicitly defined, it became easy to assign different passes to different team members and implement those mostly independently (though we frequently discussed implementation strategies with teammates).

We have made several changes to some of the IRs since our initial design. Since such changes affected the implementer of either the previous or the next pass, we made sure to communicate our desires to make such changes during our meetings and/or on Slack.

4.2 Project Timeline

Below is our timeline of events for completing this project:

Date Range	**Task(s) Completed**
Sep 13 - Sep 20	Decide Features & Write Proposal
Oct 2 - Oct 9	Hash Out Language Grammar
Oct 9 - Oct 18	Implement Scanner/Parser Implement Testing Framework for Scanner/Parser Write Initial Language Reference Manual
Oct 23 - Oct 30	Hash Out Compiler Passes & Intermediate Representations
Oct 30 - Nov 8	For Each Pass, Implement Functionality for "Hello World" Implement Functionality for Primitive Operators Build Extendible Testing Framework
Nov 13 - Nov 29	Implement All Features *except* for Custom Datatypes & Associated Memory Safety (the hardest ones) Refine Extendible Testing Framework Rigorously Test Type Checker
Dec 4 - Dec 12	Implement Custom Datatypes & Memory Safety
Dec 12 - Dec 15	Prepare Presentation, Report, & Experiment with Writing Crazy Programs (e.g. Brainf**k Interpreter)

4.3 Roles and Responsibilities

We decided on assigning each team member to the following roles:

Randy Dang was assigned the role of *manager*, who was responsible for calling meetings, coordinating logistics such as setting up GitHub, and ensuring that we were making steady progress according to the course deliverable deadlines. He usually coordinated the submission process for such deliverables, writing necessary documentation and checking that our submissions met the criteria.

Jasper Geer was assigned the role of *language guru*, who was responsible for making semantic decisions surrounding the Compost language and planning and communicating the vision of the language. He was responsible for writing most of the Language Reference Manual, driving our "IR-driven" development process, and keeping the vision intact throughout our implementation.

Roger Burtonpatel and Jackson Warhover both took on the roles of *co-system architects* and *co-testers*, and they were responsible for planning out what passes needed to be done in our compiler as well as the role of each pass, all in line with the language's vision and serving to make future passes (and ultimately Codegen) easier to implement. They were also responsible for writing and architecting our testing framework, making it easy to isolate the testing of our compiler up to a specific pass, and writing pretty-printing functionality to make issues easy to debug.

In *addition* to the roles described above, we each took the lead on implementing at least one compiler pass; the specifics of who did which pass(es) is described in the overall architecture section.

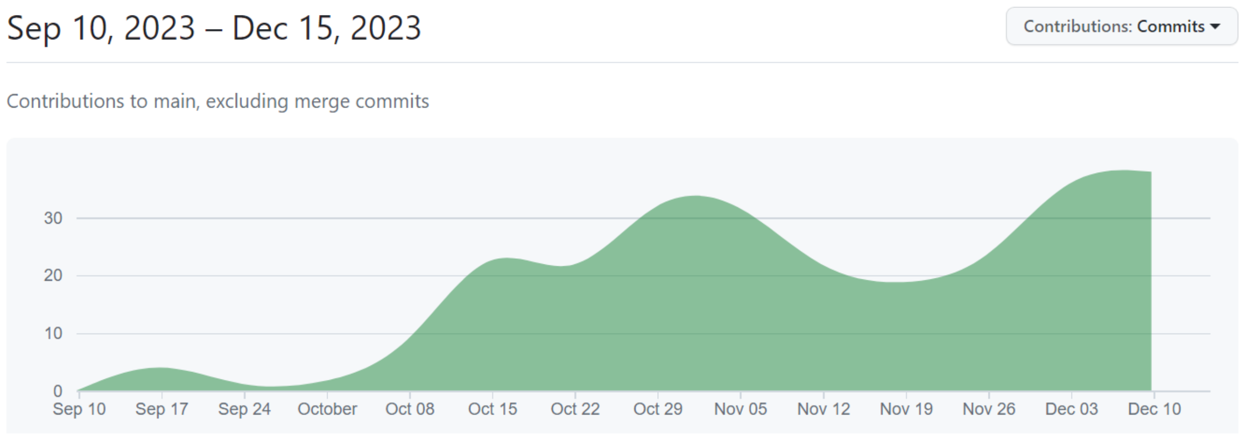
4.4 Tools Used

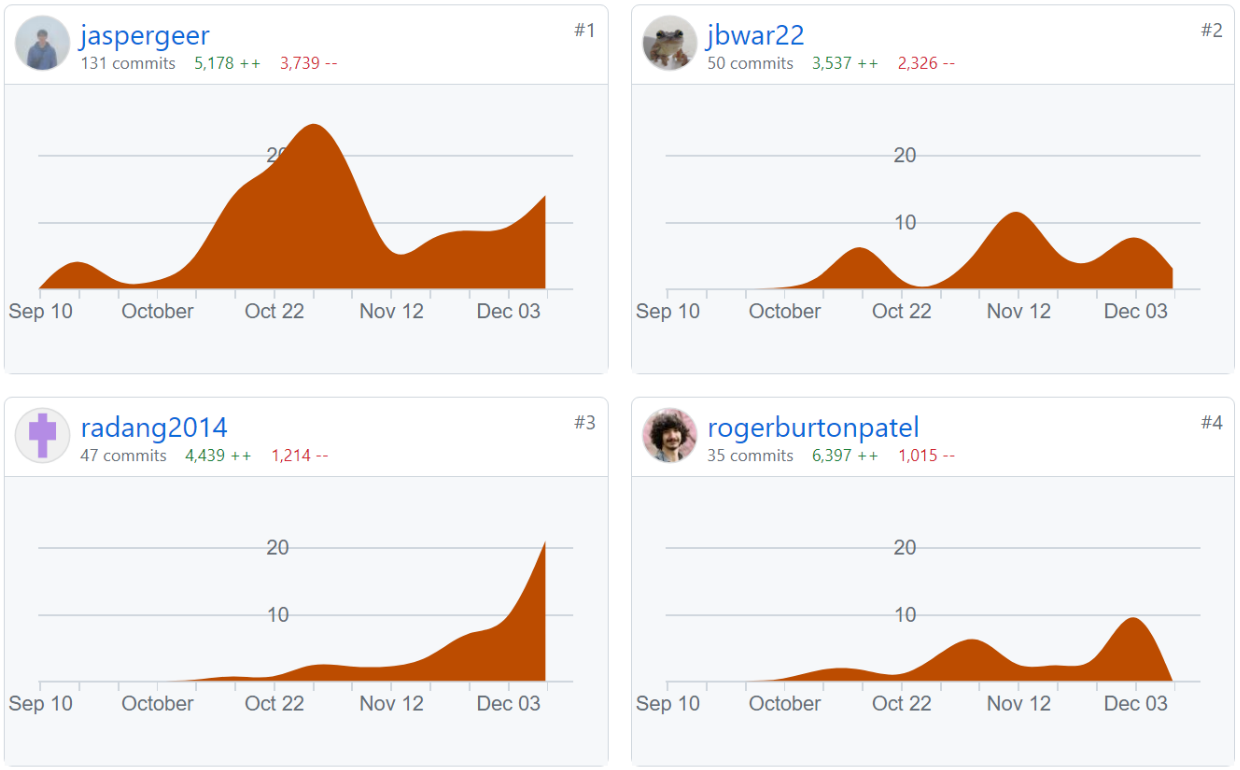
We used GitHub to set up a central, remote repository (that we would push to and pull from) containing all of our contributions, and we implemented our compiler in the OCaml language as directed. Our compiler builds with the Dune build system.

We didn't have a standardized IDE because each of us had different preferences. Vim, Emacs, and VSCode were among our editors of choice. Depending on our operating systems, some of us developed on our local UNIX-based machines, whereas those of us with Windows used Windows Subsystem for Linux (WSL).

4.5 Version Control Commits

Below are visualizations containing overall commit information and commit information by team member generated by GitHub. Note that not all commits carry the same level of significance.





5 Architectural Design

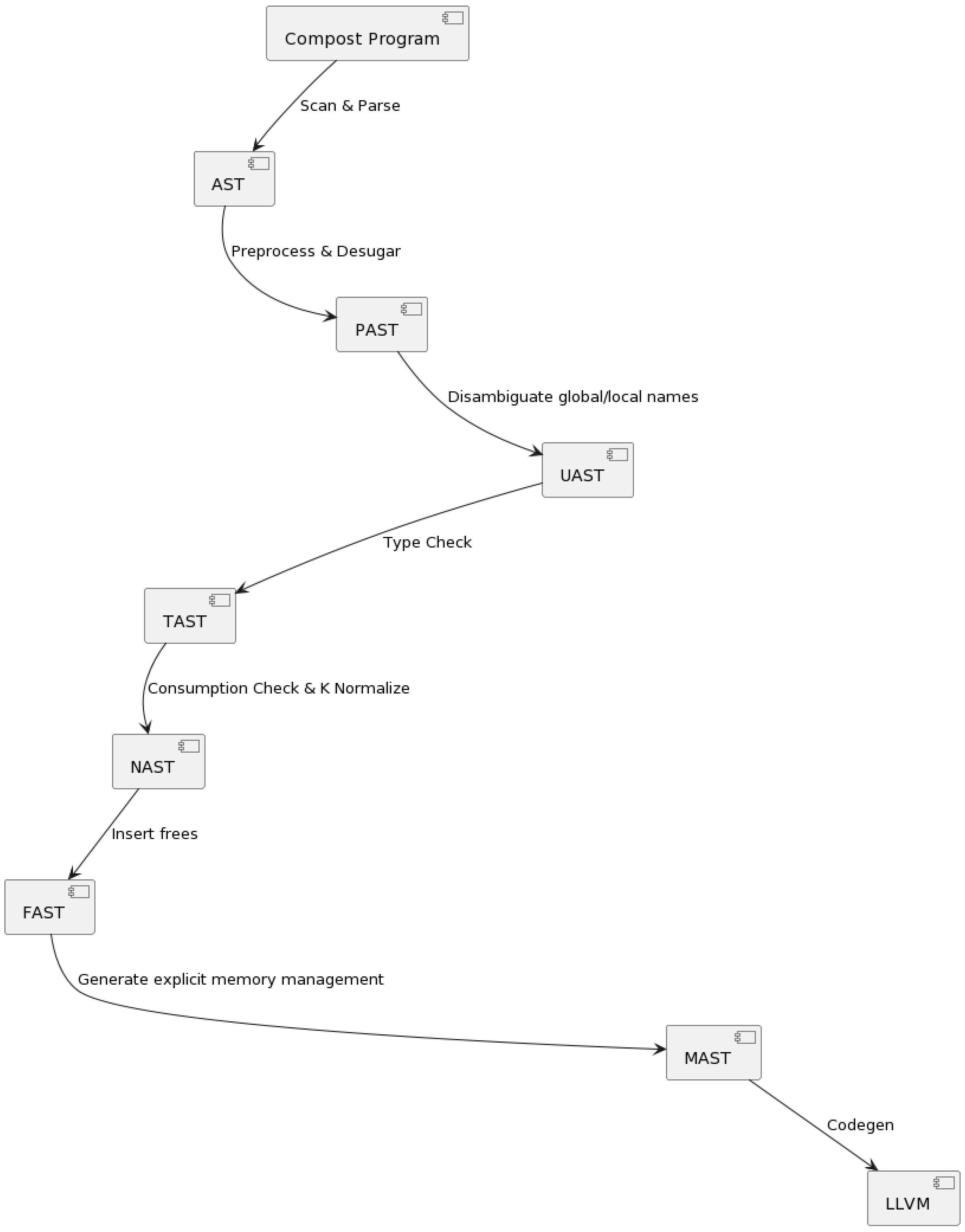
Give block diagram showing the major components of your compiler and the interfaces between them

Summarize how the language's "interesting" features were implemented

State who implemented each component

5.1 Overall Architecture

The overall architecture of our compiler can be described as a "pipe" that begins with the Compost program and ends with the generated LLVM code, undergoing numerous transformations along the way. The UML diagram below is a graph showing all of our compiler passes. Each internal node defines an intermediate representation, which is usually a tree storing some information about the program, and each edge defines a pass which is a transformation from one representation to another.



The following table maps the abbreviations of different intermediate representations used in the above nodes to what the abbreviations stand for.

Abbreviation	**What It Stands For**
AST	Abstract Syntax Tree
PAST	Preprocessed Abstract Syntax Tree
UAST	Unambiguous Abstract Syntax Tree
TAST	Type-Checked Abstract Syntax Tree
NAST	Normalized Abstract Syntax Tree
FAST	Frees-Inserted Abstract Syntax Tree
MAST	Memory-Managed Abstract Syntax Tree
LLVM	Low Level Virtual Machine

Each team member was assigned at least one pass to implement, although plenty of us made edits in each other's code after realizing the limitations of the implementations that we had initially planned for. The initial authors of each pass were as follows:

Pass	**Initial Author**
Scanning & Parsing (Compost $\rightarrow$ AST)	Randy Dang
Preprocessing & Desugaring (AST $\rightarrow$ PAST)	Jackson Warhover
Disambiguation (PAST $\rightarrow$ UAST)	Jasper Geer
Type Checking (UAST $\rightarrow$ TAST)	Roger Burtonpatel
Consumption Checking & K Normalization (TAST $\rightarrow$ NAST)	Jasper Geer
Insertion of Frees (NAST $\rightarrow$ FAST)	Jasper Geer
Generation of Explicit Memory Management Functionality (FAST $\rightarrow$ MAST)	Randy Dang
Codegen (MAST $\rightarrow$ LLVM)	Jasper Geer

In the following sections, we summarize how we implemented Compost's most interesting features.

5.2 Feature 1: Memory Safety under Affine Type System!

5.3 Feature 2: Partial Type Inference

5.4 Feature 3: Custom ADTs and Top-Level Pattern Matching

5.5 Other Miscellaneous Features

Tailcall optimization

Higher order functions

Include Global Object Value (.gov) Files

5.5.1 Tailcall Optimization

5.5.2 Higher Order Functions

5.5.3 Importing External Files

We have a fairly simple yet effective method for importing external files into a Compost program. As part of our original parsed abstract syntax tree, a use declaration can be provided, which takes in a single symbol. A symbol was chosen over a name because a symbol by definition can represent any character, and file paths needed to support some of the characters that were reserved such as parentheses. Use declarations get handled as part of our preprocessor, and the post-preprocessor subset of the Ast (the PAst) does not include use declarations in its definitions. We use the same process as the top-level driver `compost.ml` to load the data in the inputted file. Since a program is just a list of definitions, with use declarations being one of them, and a single use declaration essentially corresponds to a separate list of definitions, we simply expand out each use to build a final definition list. We do this recursively to handle nested use declarations, and the preprocessor will check to ensure that such nested declarations do not form a circular reference. Additionally, since we do not allow for duplicate globals, a single file will only ever be expanded a single time; the preprocessor will simply ignore any subsequent use declarations for that file. For example, if you have library `A.gov`, and libraries `B.gov` and `C.gov` both depend on `A.gov`, then both `B.gov` and `C.gov` can "use" `A.gov` while both being "use"d by some other program. We use the `.gov` extension in our provided library files (files meant to be "use"d by a program) to indicate that they are "global only values," but also because it's funny.

6 Test Plan

DONE: Explain how your group approached unit and integration testing, and what automation was used.

Show two or three representative source language programs along with the target language program generated for each (if you can provide syntax highlighting and nice formatting that's REALLY useful)

State who did what
We decided to write a custom bash script for unit and integration testing. This file had many iterations over the course of the project, and was continuously evolving to meet our testing needs.

The bash script is able to adapt to the operating system it is run on and change some things that need to be different. It begins by building compost to ensure that we aren't testing an older version. It then checks for provided command line flags and args. A single flag can be used to specify a single testing suite that we have developed. A suite consists of a folder inside of the tests folder, as well as a specific call to compost to test against. running with no flag runs all of the test suites. Additionally, a filename can be provided to the test script. This allowed us to run single tests, without waiting for the others to finish.

Our design for test files was simple. Each test is written as a .com file inside of our tests folder. Optionally, there is a file of the same name with .in as its extension. If such a file is present, then the script will automatically send the file in on stdin. Also optinally, there is a file with the same name with .out as its extension. This is to specify what the expected output of the test would be. For some tests, mostly Ast tests, this option is omitted, because the output is expected to be identical to the input.

The script runs each test, and if the output is different than the expected value, it prints the output and expected value, and ensures that the user knows which tests failed. Overall, it allowed for extremely quick and easy development of tests.

This works for unit testing as well as integration testing. The compost compiler supports pretty-printing most of the IRs, if given the specified flag. Our suites are designed to test specific IR outputs. Other than tests in the "run" suite, our tests are mostly unit tests, designed to test a specific function or a specific invariant in a specific IR. This is why a lot of the tests are fairly short, meant to be a short and concise example that adequately ensures that the behavior is what we expect. Tests in the "run" suite can be treated as integration tests, because they are actually compiled. These are the tests that typically use ".in" files. These test entire compost programs to ensure that they actually run without error. Asside from the script, we have done manual testing of most of these tests with valgrind to ensure memory safety of our generated code.

7 Lessons Learned

Each team member should explain their most important takeaways from working on this project

Include any advice the team has for future teams

7.1 Roger Burtonpatel

7.2 Randy Dang

My major takeaways from this project are that functional languages are powerful when implementing compilers and that there is a lot of benefit to planning ahead.

Since language definitions and ultimately the abstract tree representations are naturally recursive, having a language that is built for handling this recursive structure allowed for much more efficient programming (last year for my Capstone project, my group implemented a Python transpiler in an Object-Oriented manner, which was less clean approach). Using existing parsing tools (such as Menhir) that already implemented complex parsing algorithms also simplified the compiler implementation process, encouraging me not to “re-invent the wheel” when there already exists pretty sophisticated methods for solving a problem.

When we worked on a project as large as this one, we really benefitted from going into the project with a plan in mind for how we were planning on organizing our passes, as that ultimately made development more efficient in the long-run. On smaller scales, when we planned certain algorithms (implemented in certain passes) ahead of time, we thought of potential issues that led to algorithm revisions before even beginning to implement the pass. On the other hand, failing to consider certain issues ahead of time sometimes led to a need to undo a substantial amount of work to restructure the algorithm. Thus, the benefit of planning ahead really showed up in this project.

My advice for future teams would be to consider as many potential issues as possible up front when planning for the implementation rather than waiting for them to come up during the implementation process in order to avoid spending a lot of time stuck at dead ends (although it is also important to plan for the fact that there will inevitably be some issues only discovered during the implementation process). Hashing out the intermediate representations before implementing any compiler passes also makes it easy to break up the compilation process into concrete, manageable pieces that can also be an effective way to split up the work.

Also, use the OCaml language server to help with development. I was embarrassingly unaware of its existence until two weeks before the end of the semester (and have been coding without syntax highlighting up until then).

7.3 Jasper Geer

7.4 Jackson Warhover

A major takeaway I had was that initial planning and design can do wonders in streamlining the process of writing a complex project like a compiler. Focusing on the LRM in early stages and working through to make sure invariants were understood were absolutely critical steps.
We had a good plan from the start for exacrly how to implement IRs, which made creating them very simple. It also made it very easy to add new IRs as they were necessary. I think a lack of design considerations in the early stages would have made implementing new IRs much harder
I also learned that communication of invariants is important. For invariants that are not inheritly described by the structure of the types in the LRM, it is important for all members of a group to understand what the purpose of an IR is in order to reduce redundancy. Clear communication of ideas about what section is responsible for what means cleaner code in the end.
In terms of testing, I learned just how critical the initial work on testing infastructure can be. For us, a lot of this work was correctly put in early on, and refactors were made as necessary, which might have bren time consuming in the moment, but equated to a significant reduction in workload for tests going forward.
I also agree with Randy about how functional programing is perfect for writing a compiler. This might judt be because I'm used to it at this point, but I really cant imagine implementing something nearly as clean in a language like C.

8 Appendix: Code Listing

8.1 Representations

8.1.1 ast.ml

```
(* Abstract Syntax Tree and functions for printing it *)

(* Author: Randy Dang
 * Edited by: Jasper Geer, Jackson Warhover, Roger Burtonpatel
 *)

type name = string

type ty = FunTy of (ty list) * ty | SingleTy of name

type symlit = string
type filename = symlit

type literal =
    IntLit of int
  | BoolLit of bool
  | SymLit of symlit
  | UnitLit

type pattern =
    Pattern of name * name list
  | Name of name
  | WildcardPattern

and expr =
    Literal of literal
  | NameExpr of name
  | Case of expr * (pattern * expr) list
  | If of expr * expr * expr
  | Begin of expr list
  | Let of ((name * expr) list) * expr
  | Apply of expr * (expr list)
  | Dup of name

type def =
    Val of name * expr
  | Define of name * (name list) * expr
  | Datatype of name * (name * ty list) list
  | TyAnnotation of name * ty
  | Use of filename

type program = def list

(* Pretty printing functions *)

let rec string_of_namelist = function
    [] -> ""
  | name :: names -> name ^ " " ^ string_of_namelist names

let rec string_of_ty = function
    FunTy(tylist, ty) ->
      "(-> (" ^ String.concat " " (List.map string_of_ty tylist) ^ ") " ^
      ↪  string_of_ty ty ^ ")"
  | SingleTy(name) -> name
```

```
let string_of_symlit lit =
    let escape_backslashes = String.concat "\\\\" (String.split_on_char '\\'
    ↪  lit)
    in
    let escape_quotes = String.concat "\\'" (String.split_on_char '\''
    ↪  escape_backslashes)
    in
    "'" ^ escape_quotes ^ "'"

let string_of_lit = function
    IntLit(lit) -> string_of_int lit
  | BoolLit(lit) -> string_of_bool lit
  | SymLit(lit) -> string_of_symlit lit
  | UnitLit -> "unit"

let is_int = String.for_all (function '0' .. '9' -> true | _ -> false)

let string_of_nameorwildcard = function
    name when is_int name -> "_"
  | name -> name

let string_of_pattern = function
  | Pattern (name, []) ->
  "(" ^ name ^ ")"
  | Pattern (name, nameorwildcardlist) ->
      "(" ^ name ^ " " ^ String.concat " " (List.map string_of_nameorwildcard
      ↪  nameorwildcardlist) ^ ")"
  | WildcardPattern -> "_"
  | Name n -> n

let string_of_variant = function
    (name, tylist) -> "[" ^ name ^ " (" ^ String.concat " " (List.map
    ↪  string_of_ty tylist) ^ ")]"

let rec string_of_expr = function
    Literal(lit) -> string_of_lit lit
  | NameExpr(name) -> name
  | If(expr1, expr2, expr3) ->
      "(if " ^ string_of_expr expr1 ^ " " ^ string_of_expr expr2 ^ " " ^
      ↪  string_of_expr expr3 ^ ")"
  | Begin(exprlist) ->
      "(begin " ^ String.concat " " (List.map string_of_expr exprlist) ^ ")"
  | Let(bindlist, expr) ->
      "(let " ^ "(" ^ String.concat " " (List.map string_of_bind bindlist) ^ ")
      ↪  " ^ string_of_expr expr ^ ")"
  | Apply(expr, []) ->
    "(" ^ string_of_expr expr ^ ")"
  | Apply(expr, exprlist) ->
    "(" ^ string_of_expr expr ^ " " ^ String.concat " " (List.map
    ↪  string_of_expr exprlist) ^ ")"
  | Case(expr, casebranchlist) ->
    "(case " ^ string_of_expr expr ^ " (" ^ String.concat " " (List.map
    ↪  string_of_casebranch casebranchlist) ^ "))"
  | Dup(name) ->
    "(dup " ^ name ^ ")"

and string_of_bind = function
    (name, expr) -> "[" ^ name ^ " " ^ string_of_expr expr ^ "]"

and string_of_casebranch = function
    (pattern, expr) -> "[" ^ string_of_pattern pattern ^ " " ^ string_of_expr
    ↪  expr ^ "]"

let string_of_def = function
    Val(name, expr) -> "(val " ^ name ^ " " ^ string_of_expr expr ^ ")"
  | Define(name, namelist, expr) ->
    "(define " ^ name ^ " (" ^ String.concat " " namelist ^ ") " ^
    ↪  string_of_expr expr ^ ")"
  | Datatype(name, variantlist) ->
    "(datatype " ^ name ^ " (" ^ String.concat " " (List.map string_of_variant
    ↪  variantlist) ^ "))"
  | TyAnnotation(name, ty) ->
    "(: " ^ name ^ " " ^ string_of_ty ty ^ ")"
  | Use(filename) ->
    "(use " ^ filename ^ ")"

let string_of_program deflist = String.concat "\n" (List.map string_of_def
↪  deflist) ^ "\n"
```

8.1.2 past.ml

```
(* Processed Abstract Syntax Tree and functions for printing it *)

(* Author: Jackson Warhover
 * Edited by: Jasper Geer, Roger Burtonpatel
 *)

type name = Ast.name

type ty = Ast.ty
type literal = Ast.literal

type pattern = Ast.pattern

and expr =
  Literal of literal
| NameExpr of name
| Case of expr * (pattern * expr) list
| If of expr * expr * expr
| Let of name * expr * expr
| Apply of expr * (expr list)
| Dup of name

type def =
  Define of name * (name list) * expr
| Datatype of name * (name * ty list) list
| TyAnnotation of name * ty

type program = def list


(* Backwards to Ast & Printing *)

let rec pcb_to_acb = function (p, expr) -> (p, pexpr_to_aexpr expr)

and pexpr_to_aexpr = function
  Literal(lit) -> Ast.Literal(lit)
| NameExpr(name) -> Ast.NameExpr(name)
| If(expr1, expr2, expr3) -> Ast.If(pexpr_to_aexpr expr1, pexpr_to_aexpr expr2,
↪  pexpr_to_aexpr expr3)
| Let(name, expr1, expr2) -> Ast.Let([(name, pexpr_to_aexpr expr1)],
↪  pexpr_to_aexpr expr2)
| Apply(expr, exprlist) -> Ast.Apply(pexpr_to_aexpr expr, (List.map
↪  pexpr_to_aexpr exprlist))
| Case(expr, casebranchlist) -> Ast.Case(pexpr_to_aexpr expr, (List.map
↪  pcb_to_acb casebranchlist))
| Dup(name) -> Ast.Dup(name)

let pdef_to_adef = function
  Define(name, namelist, expr) -> Ast.Define(name, namelist, pexpr_to_aexpr
   ↪  expr)
| Datatype(name, variantlist) -> Ast.Datatype(name, variantlist)
| TyAnnotation(name, ty) -> Ast.TyAnnotation(name, ty)
```

```
let ast_of_program deflist = List.map pdef_to_adef deflist

let string_of_program deflist = Ast.string_of_program (ast_of_program deflist)
```

8.1.3 uast.ml

```
(* Unambiguous Abstract Syntax Tree *)

(* Author: Jasper Geer
 * Edited by: Roger Burtonpatel, Jackson Warhover
 *)

type name = string

type ty =
    FunTy of (ty list) * ty
  | Int
  | Bool
  | Unit
  | Sym
  | CustomTy of name

type literal = Ast.literal

type pattern =
    Pattern of name * name list
  | Name of name * bool
                    (* if false, wildcard. *)

and expr =
    Literal of literal
  | Local of name
  | Global of name
  | Case of expr * (pattern * expr) list
  | If of expr * expr * expr
  | Let of name * expr * expr
  | Apply of expr * (expr list)
  | Dup of name

type def =
    Define of name * name list * expr
  | Datatype of name * (name * ty list) list
  | TyAnnotation of name * ty

type program = def list
```

```
(* Backwards to PAst & Printing *)

let up_to_pp = function
  | Pattern(cn, ns) -> Ast.Pattern(cn, List.map ((^) "%") ns)
  | Name (_, false) -> Ast.WildcardPattern
  | Name (n, true)  -> Ast.Name ("%" ^ n)

let rec ucb_to_pcb = function (p, expr) -> (up_to_pp p, uexpr_to_pexpr expr)

and uexpr_to_pexpr = function
  | Literal(lit) -> Past.Literal(lit)
  | Local(name) -> Past.NameExpr("%" ^ name)
  | Global(name) -> Past.NameExpr("@" ^ name)
  | If(expr1, expr2, expr3) -> Past.If(uexpr_to_pexpr expr1, uexpr_to_pexpr
  ↪  expr2, uexpr_to_pexpr expr3)
  | Let(name, expr1, expr2) -> Past.Let("%" ^ name, uexpr_to_pexpr expr1,
  ↪  uexpr_to_pexpr expr2)
  | Apply(expr, exprlist) -> Past.Apply(uexpr_to_pexpr expr, (List.map
  ↪  uexpr_to_pexpr exprlist))
  | Case(expr, casebranchlist) -> Past.Case(uexpr_to_pexpr expr, (List.map
  ↪  ucb_to_pcb casebranchlist))
  | Dup(name) -> Past.Dup(name)

let rec uty_to_pty = function
  | FunTy(tys, ty) -> Ast.FunTy(List.map uty_to_pty tys, uty_to_pty ty)
  | Int -> Ast.SingleTy("int")
  | Bool -> Ast.SingleTy("bool")
  | Unit -> Ast.SingleTy("unit")
  | Sym -> Ast.SingleTy("sym")
  | CustomTy(n) -> Ast.SingleTy(n)

let rec uvs_to_pvs = function
  | [] -> []
  | (n, tys) :: vs -> (n, List.map uty_to_pty tys) :: uvs_to_pvs vs

let udef_to_pdef = function
  | Define(name, namelist, expr) -> Past.Define("@" ^ name, namelist,
  ↪  uexpr_to_pexpr expr)
  | Datatype(name, variantlist) -> Past.Datatype(name, uvs_to_pvs variantlist)
  | TyAnnotation(name, ty) -> Past.TyAnnotation("@" ^ name, uty_to_pty ty)

let past_of_program deflist = List.map udef_to_pdef deflist
let string_of_program deflist = Past.string_of_program (past_of_program deflist)
```

8.1.4 tast.ml

```
(* Type-Checked Abstract Syntax Tree *)

(* Author: Jasper Geer
 * Edited by: Roger Burtonpatel, Randy Dang, Jackson Warhover
 *)

(* open Ast *)

type name = Ast.name

type filename = Ast.filename

type ty = Uast.ty

type 'a typed = 'a * ty

type literal = Ast.literal

type pattern =
    Pattern of name * (name typed) list
  | Name of name * bool

and expr =
    Literal of literal
  | Local of name
  | Global of name
  | Case of (expr typed) * (pattern * (expr typed)) list
  | If of (expr typed) * (expr typed) * (expr typed)
  | Let of name * (expr typed) * (expr typed)
  | Apply of (expr typed) * (expr typed) list
  | Dup of name
  | Err of string

type def =
    Define of name * ty * name list * (expr typed)
  | Datatype of name * (name * ty list) list

type program = def list
```

8.1.5 nast.ml

```
(* Normalized Abstract Syntax Tree *)

(* Author: Jasper Geer *)

type name = Ast.name

type filename = Ast.filename

type ty = Uast.ty

type literal = Ast.literal

type pattern =
    Pattern of name * (name * ty) list
  | Name of name

and expr =
    Literal of literal
  | Local of name
  | Global of name
  | Case of ty * name * (pattern * expr) list
  | If of name * expr * expr
  | Let of name * ty * expr * expr
  | Apply of name * name list
  | Dup of ty * name
  | Err of ty * string

type def =
    Define of name * ty * name list * expr
  | Datatype of name * (name * ty list) list

type program = def list
```

8.1.6 fast.ml

```
(* Explicit-Free Abstract Syntax Tree *)

(* Author: Jasper Geer
 * Edited by: Randy Dang, Jackson Warhover, Roger Burtonpatel
 *)

type name = Ast.name

type typename = name
type filename = Ast.filename

type ty = Uast.ty

type literal = Ast.literal

type pattern =
    Pattern of name * (name * ty) list
  | Name of name

and expr =
    Literal of literal
  | Local of name
  | Global of name
  (* Case no longer implicity frees the top level of its scrutinee *)
  | Case of expr * (pattern * expr) list
  | If of expr * expr * expr
  | Let of name * expr * expr
  | Apply of expr * expr list
  | Dup of ty * name
  (* Memory-Related *)
  | Free of ty * name * expr (* Corresponds to a call to `free()` *)
  | FreeRec of ty * name * expr (* Corresponds to a call to "_free_" ^ (name_of
  ↪   ty) *)
  | Err of ty * string

type def =
    Define of name * ty * name list * expr
  | Datatype of name * (name * ty list) list

type program = def list
```

8.1.7 mast.ml

```
(* Explicitly Memory Managed Abstract Syntax Tree *)

(* Author: Jasper Geer
 * Edited by: Randy Dang, Roger Burtonpatel
 *)
type name = Ast.name

type filename = Ast.filename

(* ty is now LLVM types *)
(* Note: FunTy and Ptr might be redundant, we will see *)

type ty = Fun of ty * ty list | Int of int | Ptr of ty | Struct of ty list

type literal = Ast.literal

type pattern =
    Pattern of int * (name * ty) list
  | Name of name

and bind = name * expr

and expr =
    Literal of literal
  | Local of name
  | Global of name
  | Case of expr * (pattern * expr) list
  | If of expr * expr * expr
  | Let of name * expr * expr
  | Apply of expr * expr list
  (* Memory-Related *)
  | Free of name * expr
  (* Allocates a struct with a given tag and fields *)
  (* populated by the values bound to the names in the list *)
  | Alloc of ty * int * expr list
  | Err of ty * string

type def = Define of name * ty * name list * expr
  (* Datatype definitions can be erased *)
  (* All necessary type information is encoded in the _alloc and _free functions
   ↪  *)

type program = def list

(* Pretty-printing functions *)

let rec string_of_ty = function
    Fun(ty, tylist) ->
      "(-> (" ^ String.concat " " (List.map string_of_ty tylist) ^ ") " ^
       ↪  string_of_ty ty ^ ")"
  | Int(int) -> "i" ^ string_of_int int
  | Ptr(ty) -> string_of_ty ty ^ " *"
  | Struct(tylist) ->
      "(struct (members " ^ String.concat " " (List.map string_of_ty tylist) ^
       ↪  "))"
```

```
let string_of_lit lit = Ast.string_of_lit lit

let is_int = String.for_all (function '0' .. '1' -> true | _ -> false)

let string_of_nameorwildcard (name, _) =
  if is_int name then "_"
  else name

let string_of_pattern = function
    Pattern(tag, nameorwildcardlist) ->
      "(" ^ string_of_int tag ^ " " ^ String.concat " " (List.map
      ↪  string_of_nameorwildcard nameorwildcardlist) ^ ")"
  | Name n -> n

let rec string_of_expr = function
    Literal(lit) -> string_of_lit lit
  | Local(name) -> "%" ^ name
  | Global(name) -> "@" ^ name
  | Case(expr, casebranchlist) ->
      "(case " ^ string_of_expr expr ^ " (" ^ String.concat "\n" (List.map
      ↪  string_of_casebranch casebranchlist) ^ "))"
  | If(expr1, expr2, expr3) ->
      "(if " ^ string_of_expr expr1 ^ " " ^ string_of_expr expr2 ^ " " ^
      ↪  string_of_expr expr3 ^ ")"
  | Let(name, expr1, expr2) ->
      "(let " ^ "([%" ^ name ^ " " ^ string_of_expr expr1 ^ "]) \n" ^
      ↪  string_of_expr expr2 ^ ")"
  | Apply(expr, exprlist) ->
      "(" ^ string_of_expr expr ^ " " ^ String.concat " " (List.map
      ↪  string_of_expr exprlist) ^ ")"
  | Free(name, expr) ->
      "(free %" ^ name ^ " " ^ string_of_expr expr ^ ")"
  | Alloc(ty, tag, exprlist) ->
      "(alloc (type " ^ string_of_ty ty ^ ") " ^ string_of_int tag ^ " [ " ^
      ↪  String.concat "; " (List.map string_of_expr exprlist) ^ " ] "
  | Err(ty, name) ->
    "(err (" ^ string_of_ty ty ^ ") "  ^ name ^ ")"

and string_of_bind = function
    (name, expr) -> "[" ^ name ^ " " ^ string_of_expr expr ^ "]\n"

and string_of_casebranch = function
    (pattern, expr) -> "[" ^ string_of_pattern pattern ^ " " ^ string_of_expr
    ↪  expr ^ "]"
```

```
let string_of_def = function
  | Define(name, ty, namelist, expr) ->
     "(define " ^ name ^ " (type " ^ string_of_ty ty ^ ") (" ^ String.concat "
      ↪  " namelist ^ ") \n" ^ string_of_expr expr ^ ")\n"

let string_of_program deflist = String.concat "\n" (List.map string_of_def
↪  deflist)
```

8.2 Helper Modules

8.2.1 difflist.ml

```
(* Author: Jackson Warhover *)

let singleton x xs = x :: xs
let empty tail = tail
let tolist f = f []
let cons f g x = f (g x)
```

8.2.2 freshnames.ml

```
(* Author: Jasper Geer *)

(* God I hate global mutable state but it is so very useful here *)
let counter = ref 0

let fresh_name () = begin counter := !counter + 1 ; Int.to_string !counter end
```

8.2.3 primitives.ml

```
(* Author: Jasper Geer
 * Edited by: Randy Dang, Jackson Warhover
 *)

(* Association list of primitive function names and their types *)
let primitives =
  [
    (* I/O *)
    ("print-newline", Uast.FunTy ([], Uast.Unit));
    ("print-sym", Uast.FunTy ([Uast.Sym], Uast.Unit));
    ("print-int", Uast.FunTy ([Uast.Int], Uast.Unit));
    ("print-ascii", Uast.FunTy ([Uast.Int], Uast.Unit));
    ("print-bool", Uast.FunTy ([Uast.Bool], Uast.Unit));
    ("print-unit", Uast.FunTy ([Uast.Unit], Uast.Unit));
    ("in", Uast.FunTy ([], Uast.Int));

    (* Equality *)
    ("=i", Uast.FunTy ([Uast.Int; Uast.Int], Uast.Bool));
    ("=s", Uast.FunTy ([Uast.Sym; Uast.Sym], Uast.Bool));
    ("=b", Uast.FunTy ([Uast.Bool; Uast.Bool], Uast.Bool));
    ("=u", Uast.FunTy ([Uast.Unit; Uast.Unit], Uast.Bool));

    (* Arithmetic *)
    ("+", Uast.FunTy ([Uast.Int; Uast.Int], Uast.Int));
    ("-", Uast.FunTy ([Uast.Int; Uast.Int], Uast.Int));
    ("*", Uast.FunTy ([Uast.Int; Uast.Int], Uast.Int));
    ("/", Uast.FunTy ([Uast.Int; Uast.Int], Uast.Int));
    ("%", Uast.FunTy ([Uast.Int; Uast.Int], Uast.Int));
    ("udiv", Uast.FunTy ([Uast.Int; Uast.Int], Uast.Int));
    ("umod", Uast.FunTy ([Uast.Int; Uast.Int], Uast.Int));
    ("neg", Uast.FunTy ([Uast.Int], Uast.Int));

    (* Comparison *)
    (">", Uast.FunTy ([Uast.Int; Uast.Int], Uast.Bool));
    ("<", Uast.FunTy ([Uast.Int; Uast.Int], Uast.Bool));
    (">=", Uast.FunTy ([Uast.Int; Uast.Int], Uast.Bool));
    ("<=", Uast.FunTy ([Uast.Int; Uast.Int], Uast.Bool));

    (* Boolean *)
    ("not", Uast.FunTy ([Uast.Bool], Uast.Bool));
    ("and", Uast.FunTy ([Uast.Bool; Uast.Bool], Uast.Bool));
    ("or", Uast.FunTy ([Uast.Bool; Uast.Bool], Uast.Bool));
    ("xor", Uast.FunTy ([Uast.Bool; Uast.Bool], Uast.Bool));

    (* Bitwise *)
    ("&", Uast.FunTy ([Uast.Int; Uast.Int], Uast.Int));
    ("|", Uast.FunTy ([Uast.Int; Uast.Int], Uast.Int));
    ("^", Uast.FunTy ([Uast.Int; Uast.Int], Uast.Int));
    ("<<", Uast.FunTy ([Uast.Int; Uast.Int], Uast.Int));
    (">>", Uast.FunTy ([Uast.Int; Uast.Int], Uast.Int));
    ("~", Uast.FunTy ([Uast.Int], Uast.Int));
  ]

let primitive_tys =
  [
    ("int", Uast.Int);
    ("bool", Uast.Bool);
    ("unit", Uast.Unit);
    ("sym", Uast.Sym);
```

```
 ]
```

8.3 Pass Implementations

8.3.1 scanner.mll

```
(* Ocamllex scanner for Compost *)

(* Author: Randy Dang
 * Edited by: Jasper Geer, Jackson Warhover
 *)

{
  open Parser

  let format_sym_lit sym_lit =
  let sym_len = String.length sym_lit in
  let sym_body = Str.string_after (Str.string_before sym_lit (sym_len - 1)) 1 in
  let escape_backslash = Str.global_replace (Str.regexp {|\\\\|}) "\\\\" in
  let escape_single_quote = Str.global_replace (Str.regexp {|\\'|}) "'" in
  escape_backslash (escape_single_quote sym_body)
}

let digit = ['0' - '9']
let digits = ('+' | '-' | "") digit+

let boolean = "true" | "false"

let string_char = [^'\'' '\\' ] | "\\'" | "\\\\"
let string_contents = string_char*

let symlit = '\'' string_contents '\''

rule token = parse
  [' ' '\t' '\r' '\n'] { token lexbuf } (* Whitespace *)
| ';'     { comment lexbuf }            (* Comments *)
| '('      { LPAREN }
| ')'      { RPAREN }
| '['      { LBRACKET }
| ']'      { RBRACKET }
| ':'      { COLON }
| '_'      { WILDCARD }
| "->"     { ARROW }
| "if"     { IF }
| "val"    { VAL }
| "define" { DEFINE }
| "datatype" { DATATYPE }
| "use"    { USE }
| "case"   { CASE }
| "begin"  { BEGIN }
| "let"    { LET }
| "dup"    { DUP }
| "unit"   { UNIT }
| digits as lxm { INTLIT(int_of_string lxm) }
| boolean as lxm { BOOLLIT(bool_of_string lxm) }
| symlit as lxm { SYMLIT(format_sym_lit lxm) }
| [^'(' ')' '[' ']' '\'' ' ' '\t' '\r' '\n' ';']+ as lxm { NAME(lxm) }
| eof { EOF }
| _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }

and comment = parse
  '\n' { token lexbuf }
| eof  { token lexbuf }
| _    { comment lexbuf }
```

8.3.2 parser.mly

```
/* Ocamlyacc parser for MicroC */

/* Author: Randy Dang
   Edited by: Jasper Geer, Roger Burtonpatel, Jackson Warhover
  */

%{
  open Ast
  open Freshnames
%}

%token LPAREN RPAREN LBRACKET RBRACKET
%token VAL DEFINE DATATYPE USE COLON
%token CASE IF BEGIN LET
%token UNIT
%token WILDCARD ARROW DUP
%token EOF

%token <string> NAME SYMLIT
%token <int> INTLIT
%token <bool> BOOLLIT

%start program
%type <Ast.program> program
```

```
/* No need to specify any associativity or precedence in our syntax because
    our parenthesized syntax makes everything explicit
  */

%%

program:
  defs EOF { $1 }

defs:
    /* nothing */ { [] }
  | def defs { $1 :: $2 }

def:
    LPAREN definternal RPAREN { $2 }
  | LBRACKET definternal RBRACKET { $2 }

definternal:
    VAL NAME expr { Val($2, $3) }
  | DEFINE NAME parennamelist expr { Define($2, $3, $4) }
  | DATATYPE NAME parenvariantlist { Datatype($2, $3) }
  | COLON NAME ty { TyAnnotation($2, $3) }
  | USE filename { Use($2) }

parennamelist:
    LPAREN namelist RPAREN { $2 }
  | LBRACKET namelist RBRACKET { $2 }

namelist:
    /* nothing */  { [] }
  | NAME namelist { $1 :: $2 }

parenvariantlist:
    LPAREN variantlist RPAREN { $2 }
  | LBRACKET variantlist RBRACKET { $2 }

variantlist:
    /* nothing */ { [] }
  | variant variantlist { $1 :: $2 }

variant:
    LPAREN variantinternal RPAREN { $2 }
  | LBRACKET variantinternal RBRACKET { $2 }
```

```
variantinternal: NAME parentylist { ($1, $2) }

parentylist:
    LPAREN tylist RPAREN { $2 }
  | LBRACKET tylist RBRACKET { $2 }

tylist:
    /* nothing */ { [] }
  | ty tylist { $1 :: $2 }

ty:
    LPAREN funtyinternal RPAREN { $2 }
  | LBRACKET funtyinternal RBRACKET { $2 }
  | UNIT { SingleTy("unit") }
  | NAME { SingleTy($1) }

funtyinternal:
    ARROW parentylist ty { FunTy($2, $3) }

filename: SYMLIT { $1 }

expr:
    literal { Literal($1) }
  | NAME { NameExpr($1) }
  | LPAREN exprinternal RPAREN { $2 }
  | LBRACKET exprinternal RBRACKET { $2 }

exprinternal:
    IF expr expr expr { If($2, $3, $4) }
  | BEGIN exprlist { Begin($2)}
  | expr exprlist { Apply($1, $2) }
  | LET parenbindlist expr { Let($2, $3) }
  | CASE expr parencasebranchlist { Case($2, $3) }
  | DUP NAME { Dup($2) }

exprlist:
    /* nothing */ { [] }
  | expr exprlist { $1 :: $2 }

parenbindlist:
    LPAREN bindlist RPAREN { $2 }
  | LBRACKET bindlist RBRACKET { $2 }

bindlist:
    /* nothing */ { [] }
  | LPAREN bind RPAREN bindlist { $2 :: $4 }
  | LBRACKET bind RBRACKET bindlist { $2 :: $4 }

bind: NAME expr { ($1, $2) }

casebranch:
    LPAREN pattern expr RPAREN { ($2, $3) }
  | LBRACKET pattern expr RBRACKET { ($2, $3) }

parencasebranchlist:
    LPAREN casebranchlist RPAREN { $2 }
  | LBRACKET casebranchlist RBRACKET { $2 }

casebranchlist:
    /* nothing */ { [] }
  | casebranch casebranchlist { $1 :: $2 }

pattern:
    LPAREN patterninternal RPAREN { $2 }
  | LBRACKET patterninternal RBRACKET { $2 }
  | WILDCARD { WildcardPattern }
  | NAME { Name $1 }

patterninternal: NAME nameorwildcardlist { Pattern($1, $2) }

nameorwildcardlist:
    /* nothing */ { [] }
  | nameorwildcard nameorwildcardlist { $1 :: $2 }

nameorwildcard:
    NAME { $1 }
  | WILDCARD { fresh_name () }

literal:
    INTLIT { IntLit($1) }
  | SYMLIT { SymLit($1) }
  | BOOLLIT { BoolLit($1) }
  | UNIT { UnitLit }
```

8.3.3 preprocess.ml

```
(* Author: Jackson Warhover
 * Edited by: Jasper Geer, Roger Burtonpatel
 *)
```

```
module P = Past
module A = Ast
module D = Difflist

module Prim = Primitives

module S = Set.Make(String)
module SM = Map.Make(String)

(* to make pretty printing errors nicer later *)
exception RecursiveUse
exception DuplicateGlobal
exception TypeNameUsage
let except_ru c = if c then raise RecursiveUse else ()
let except_dg c = if c then raise DuplicateGlobal else ()
let except_tnu c = if c then raise TypeNameUsage else ()

let fold_prim l (n, _) = S.add n l
let vb  = ref (SM.empty)                                          (* val
↪  binding *)
let gbs = ref (List.fold_left fold_prim S.empty Prim.primitives)     (* global
↪  binding set *)
let lbs = ref (S.empty)                                           (* local
↪  binding set *)
let dts = ref (List.fold_left fold_prim S.empty Prim.primitive_tys) (* datatype
↪  set *)
let use = ref (S.empty)                                           (* use set
↪  *)

(* lbr: local bindings (recursive) - locals bound in this context *)
(* pattern: Ast - pattern * expr list *)
(* return: Past - pattern * expr list *)
let rec apes_to_ppes lbr = function
    [] -> []
  | ((A.Pattern (n, ns), e) :: pes) ->
    lbs := List.fold_right (fun n2 lbs ->
      let () = except_tnu (S.mem n2 !dts) in
      S.add n2 lbs
    ) ns !lbs ;
    let lbr2 = List.fold_right S.add ns lbr in
    (A.Pattern(n, ns), ae_to_pe lbr2 e) :: (apes_to_ppes lbr pes)
  | ((A.WildcardPattern, e) :: pes) ->
    (A.WildcardPattern, ae_to_pe lbr e) :: (apes_to_ppes lbr pes)
  | ((A.Name n, e) :: pes) ->
    let () = except_tnu (S.mem n !dts) in
    lbs := S.add n !lbs ;
    let lbr = S.add n lbr in
    (A.Name n, ae_to_pe lbr e) :: (apes_to_ppes lbr pes)

(* lbr: local bindings (recursive) - locals bound in this context *)
(* pattern: Ast - expr *)
(* return: Past - expr *)
and ae_to_pe lbr = function
  | A.Begin([]) -> P.Literal(A.UnitLit)
  | A.Begin([e]) -> ae_to_pe lbr e
  | A.Begin(e :: es) -> P.Let(Freshnames.fresh_name (), ae_to_pe lbr e, ae_to_pe
  ↪  lbr (A.Begin(es)))
  | A.Let([], e) -> ae_to_pe lbr e
  | A.Let(((abn, abe) :: abs), e) ->
    let () = except_tnu (S.mem abn !dts) in
    lbs := S.add abn !lbs ;
    let lbr2 = S.add abn lbr in
    P.Let(abn, ae_to_pe lbr abe, ae_to_pe lbr2 (A.Let(abs, e)))
  | A.Literal(l) -> P.Literal(l)
  | A.NameExpr(n) ->
    (* if S.mem n !dts then raise TypeNameUsage else (* not really necessary *)
    ↪  *)
    lbs := S.add n !lbs ;
    if (not (S.mem n lbr)) && (SM.mem n !vb)
    then SM.find n !vb
    else P.NameExpr(n)
  | A.Case(e, pes) -> P.Case(ae_to_pe lbr e, apes_to_ppes lbr pes)
  | A.If(e1, e2, e3) -> P.If(ae_to_pe lbr e1, ae_to_pe lbr e2, ae_to_pe lbr e3)
  | A.Apply(e, es) -> P.Apply(ae_to_pe lbr e, List.map (ae_to_pe lbr) es)
  | A.Dup(n) ->
    (* if S.mem n !dts then raise TypeNameUsage else (* not really necessary *)
    ↪  *)
    P.Dup(n)

(* use_r: use statements (recursive) - checking for recursive use *)
(* pattern: Ast - def *)
(* return: Past - def difflist *)
let rec ad_to_pdl use_r = function
  | A.Use(filename) ->
    let () = except_ru (S.mem filename use_r) in
    if S.mem filename !use then D.empty else
    let channel = open_in filename in
    let lexbuf = Lexing.from_channel channel in
    let ast = Parser.program Scanner.token lexbuf in
    use := S.add filename !use ;
    let use_r = S.add filename use_r in
    adl_to_pdl ast use_r
  | A.Val(n, e) ->
    let () = except_dg (SM.mem n !vb ) in
    let () = except_dg (S.mem n !gbs) in
    let () = except_dg (S.mem n !dts) in
    let pe = ae_to_pe S.empty e in
    vb := SM.add n pe !vb ;
    D.empty
  | A.Define(n, ns, e) ->
    let () = except_dg (SM.mem n !vb ) in
    let () = except_dg (S.mem n !gbs) in
    let () = except_dg (S.mem n !dts) in
    gbs := S.add n !gbs ;
    lbs := List.fold_right (fun n2 lbs ->
      let () = except_tnu (S.mem n2 !dts) in
      S.add n2 lbs
    ) ns !lbs ;
    let lbr = List.fold_right S.add ns S.empty in
    let pe = ae_to_pe lbr e in
    D.singleton (P.Define(n, ns, pe))
  | A.Datatype(n, ntss) ->
    let () = except_dg (SM.mem n !vb ) in
    let () = except_dg (S.mem n !gbs) in
    let () = except_dg (S.mem n !dts) in
    let () = except_tnu (S.mem n !lbs) in
    dts := S.add n !dts ;
    gbs := List.fold_right (fun (n2, _) gbs ->
        let () = except_dg (SM.mem n2 !vb ) in
        let () = except_dg (S.mem n2 gbs) in
        let () = except_dg (S.mem n2 !dts) in
        S.add n2 gbs
    ) ntss !gbs ;
    D.singleton (P.Datatype(n, ntss))
  | A.TyAnnotation(n, t) -> D.singleton (P.TyAnnotation(n, t)) (* TODO? *)

(* use_r: use statements (recursive) - checking for recursive use *)
(* pdl: Past - def difflist (FOLD ACCUM) *)
(* ad: Ast - def (FOLD LIST ELEM) *)
(* return: Past - def difflist (FOLD ACCUM) *)
and fold_adl_to_pdl use_r pdl ad = D.cons pdl (ad_to_pdl use_r ad)

(* adl: Ast - def list *)
(* use_r: use statements (recursive) - checking for recursive use *)
(* return: Past - def difflist *)
```

```
and adl_to_pdl adl use_r =
  List.fold_left (fold_adl_to_pdl use_r) D.empty adl

(* ENTRY POINT FROM COMPOST.ML *)
(* adeflist: Ast - def list *)
let preprocess adeflist = D.tolist (adl_to_pdl adeflist S.empty)
```

8.3.4 disambiguate.ml

```
(* Author: Jasper Geer
 * Edited by: Roger Burtonpatel, Jackson Warhover
 *)

module P = Past
module U = Uast
module A = Ast

module Prim = Primitives

module S = Set.Make(String)
module StringMap = Map.Make(String)

(* let rec freeIn n = function
   | U.Literal _ -> false
   | U.Global n' | U.Local n' -> n = n'
   | U.Case *)

let primty = List.fold_right (fun (n, ty) pts ->
    StringMap.add n ty pts
) Prim.primitive_tys StringMap.empty

let rec aty_to_uty = function
  | A.FunTy(tys, ty) -> U.FunTy(List.map aty_to_uty tys, aty_to_uty ty)
  | A.SingleTy(n) ->
    if StringMap.mem n primty then StringMap.find n primty else
    U.CustomTy(n)

let rec expr locals renamings = function
  | P.Literal l -> U.Literal l
  | P.NameExpr n when S.mem n locals ->
    if StringMap.mem n renamings
    then U.Local (StringMap.find n renamings)
    else U.Local n
  | P.NameExpr n -> U.Global n
  | P.Case (e, branches) ->
```

```
    let e' = expr locals renamings e in
    let branch (p, body) = match p with
      | A.Pattern (n, bindings) ->
        let rename_binding renamings name =
          if S.mem name locals
          then
            let new_name = Freshnames.fresh_name () in
            (StringMap.add name new_name renamings, new_name)
          else (renamings, name)
        in
        let (renamings', bindings') = List.fold_left_map rename_binding
        ↪  renamings bindings in
        let locals' = S.union (S.of_list bindings) locals in
        (U.Pattern (n, bindings'), expr locals' renamings' body)
        (* THESE NEED TO BE FIXED *)
      | A.WildcardPattern -> (U.Name (Freshnames.fresh_name (), false), expr
      ↪  locals renamings body)
      | A.Name n -> (U.Name (n, true), expr locals renamings body)
    in
    let branches' = List.map branch branches in
    U.Case (e', branches')
  | P.If (e1, e2, e3) ->
    let e1' = expr locals renamings e1 in
    let e2' = expr locals renamings e2 in
    let e3' = expr locals renamings e3 in
    U.If (e1', e2', e3')
  | P.Let (n, e, body) when S.mem n locals ->
    let n' = Freshnames.fresh_name () in
    let e' = expr locals renamings e in
    let renamings' = StringMap.add n n' renamings in
    let body' = expr (S.add n locals) renamings' body in
    U.Let (n', e', body')
  | P.Let (n, e, body) ->
    let e' = expr locals renamings e in
    let body' = expr (S.add n locals) renamings body in
    U.Let (n, e', body')
  | P.Apply (e, es) ->
    let e' = expr locals renamings e in
    let es' = List.map (expr locals renamings) es in
    U.Apply (e', es')
  | P.Dup n -> U.Dup n

let rec vs_to_utyvs = function
  | [] -> []
  | (n, tys) :: vs -> (n, List.map aty_to_uty tys) :: vs_to_utyvs vs
```

```
let def = function
  | P.Define (n, args, body) -> U.Define (n, args, expr (S.of_list args)
  ↪  StringMap.empty body)
  | P.Datatype (n, variants) -> U.Datatype (n, vs_to_utyvs variants)
  | P.TyAnnotation (n, ty) -> U.TyAnnotation (n, aty_to_uty ty)

let disambiguate = List.map def
```

8.3.5 typecheck.ml

```
(*
 * Author: Roger Burtonpatel
 * Edited by: Jasper Geer, Randy Dang, Jackson Warhover
 *)

module A = Ast
module U = Uast
module T = Tast

module StringMap = Map.Make(String)
exception Impossible of string
exception Todo
exception NotFound of string
exception TypeError of string


let rec eqType t1 t2 = match (t1, t2) with
| (U.Int, U.Int) | (U.Bool, U.Bool) | (U.Sym, U.Sym) | (U.Unit, U.Unit) -> true
| (U.FunTy (arg_ts, ret_t)), (U.FunTy (arg_ts', ret_t')) ->
                    eqTypes arg_ts arg_ts' && eqType ret_t ret_t'
| (U.CustomTy n, U.CustomTy n') -> n = n'
| _ -> false
and eqTypes ts ts' = List.equal eqType ts ts'

let rec tyString = function
| U.Int -> "int"
| U.Bool -> "bool"
| U.Sym -> "symbol"
| U.Unit -> "Unit"
| U.FunTy (arg_ts, ret_t) ->
                    "(-> ("
                    ^ String.concat " " (List.map tyString arg_ts) ^ ") "
                    ^ tyString ret_t ^ ")"
| U.CustomTy name -> name
```

```
(* let typesMatchOrError t1 t2 metainfo =
  if eqType t1 t2
  then true
  else raise
      (TypeError ("type mismatch: expected "
                  ^ tyString t1
                  ^ " but got "
                  ^ tyString t2 ^ metainfo)) *)


let checkFunTypes n param_ts arg_ts ret_t =
  let rec go t_s t_s' =
  match (t_s, t_s') with
| ([], []) -> ()
| (tau::taus, tau'::taus') ->
      if eqType tau tau'
      then go taus taus'
      else
        let funtysMismatchError n ts ts' ret_t =
          "type mismatch: expected "
                                  ^ tyString (U.FunTy (ts, ret_t))
                                  ^ " but got "
                                  ^ tyString (U.FunTy (ts', ret_t))
                                  ^ " in application of function " ^ n
        in raise (TypeError (funtysMismatchError n param_ts arg_ts ret_t))
| (_, _) -> raise (Impossible "mismatch in number of types in checkFunArgTypes")
    in go param_ts arg_ts

let extendGammaWithPat topty gamma delta pat =
  match pat with
  | U.Name (_, false) -> gamma
  | U.Name (n, true) -> StringMap.add n topty gamma
  | U.Pattern (pn, ns) ->
    let (typ_args, _) = StringMap.find pn delta in
    let gamma' =
      List.fold_left2 (fun g n t -> StringMap.add n t g)
                          gamma ns typ_args in
      gamma'

let aPatofTPat = function
  | T.Name (_, false)     ->  A.WildcardPattern
  | T.Name (n, true)      ->  A.Name n
  | T.Pattern (n, ns_tys) ->  let (names, _) = List.split ns_tys in
                              A.Pattern (n, names)
```

```
let print_warning warn =
  let _ = Printf.eprintf ("\n") in
  let _ = Printf.eprintf ("\027[1;38;5;90m") in
  let _ = Printf.eprintf ("Warning:") in
  let _ = Printf.eprintf ("\027[0m") in
  let _ = Printf.eprintf (" %s\n") warn in
  flush stderr;
  ()

let typerror err =
  let _ = Printf.eprintf ("\027[1;31m") in
  let _ = Printf.eprintf ("Error:") in
  let _ = Printf.eprintf ("\027[0m") in
  let _ = Printf.eprintf (" %s\n") err in
  exit 1
let pruneBranchesWith (branches : (T.pattern * (T.expr T.typed)) list)
                      (possibleVariants : A.name list) =
  let checkBranch (newbranches, foundVariants, warn) branch =
    match branch with
    | (T.Name _, _) ->
        (* wildcard or name says "all possible variants have been found" *)
        (List.append newbranches [branch], possibleVariants, warn)
    | (T.Pattern (vcon, _) as pat, _) ->
        (* only add if not yet found *)
        let (warn', pats') = if List.exists (fun vc -> vc = vcon) foundVariants
          then ("unreachable pattern \""
                ^ A.string_of_pattern (aPatofTPat pat) ^ "\"", newbranches)
        else ("", List.append newbranches [branch])
      in (pats', vcon::foundVariants, warn')
  in
  let (branches', foundVariants, warn) =
    List.fold_left checkBranch ([], [], "") branches in
  let () =
    if not (warn = "") then print_warning warn else
    let (ps, fs) = (List.sort String.compare possibleVariants,
                    List.sort String.compare foundVariants) in
    if not (List.equal String.equal ps fs)
    then print_warning "pattern matching is non-exhaustive"
  in
  branches'


let curry f x y = f (x, y)

(* gamma: name -> ty *)
(* delta: value constructor name -> (types-of-its-arguments, type-it-constructs)
↪  *)

let rec typeof gamma delta expr =
  let rec typ = function
  | U.Literal l ->
    (match l with A.IntLit _ -> U.Int
                            | A.BoolLit _ -> U.Bool
                            | A.SymLit _ -> U.Sym
                            | A.UnitLit -> U.Unit)
  (* NOTE: Do we want a sanity check that all globals are Funty? *)
  | U.Local n | U.Global n ->
                  if not (StringMap.mem n gamma)
                  then
                    raise (NotFound ("unbound name \"" ^ n ^ "\""))
                  else StringMap.find n gamma
  | U.If (e1, e2, e3) ->
    (match (typ e1, typ e2, typ e3) with
                        | (U.Bool, t1, t2) ->
                          if t1 = t2
                          then t1
                          else raise
                            (TypeError "mismatched types in if branches")
                        | _ -> raise
                            (TypeError ("condition failed to typecheck to "
                                        ^ "boolean in \"if\" expression")))
  | U.Let (n, e, e') ->
    let rhs_t = typ e in
                      let extended_gamma = (StringMap.add n rhs_t gamma) in
                      typeof extended_gamma delta e'
  | U.Apply (f, es) ->
    let funty = typeof gamma delta f in
    (match funty with
    | U.FunTy (arg_ts, ret_t) ->
      let n = (match f with U.Global n' | U.Local n' -> "\"" ^ n' ^ "\""
                                        | _ -> "") in
        let n_expected = List.length arg_ts in
                  let n_given    = List.length es in
                  if n_expected != n_given
                  then raise (TypeError ("mismatch in number of arguments in "
                                       ^ "application of " ^ n ^ ": expected "
                                       ^ Int.to_string n_expected
                                       ^ " but "
                                       ^ Int.to_string n_given
                                       ^ " were given."))
```

```
                (* typecheck arguments - purely side-effecting *)
                 else
                   let () = checkFunTypes n (List.map typ es) arg_ts ret_t in
                 ret_t (* type is return type *)
        | _ ->  raise (TypeError ("attempted to apply non-function of type " ^
        ↪  tyString funty)))

 | U.Dup n -> if not (StringMap.mem n gamma)
   then raise (NotFound ("attempted to dup unbound name \"" ^ n ^ "\""))
   else StringMap.find n gamma

 | U.Case (_, []) -> raise (TypeError "empty case expression")
 | U.Case (e, branches) ->
   let typ_scrutinee = typ e in
   (* scrutinee MUST be custom type; no literal pattern matching *)
     (match typ_scrutinee with U.CustomTy sname ->
       let (patterns, rhss) = List.split branches in
       (* check all patterns to be well-formed with regards to the scrutinee *)
       let typeCheckPattern = function
         | U.Name _ -> ()
         | U.Pattern (pname, _) ->
           (* ensure pattern maps to a type *)
           if not (StringMap.mem pname delta)
           then raise (TypeError ("unknown type constructor \"" ^ pname
                                  ^ "\" in case branch"))
           else
           (* ensure the type it matches to is correct *)
           let (_, typ_of_pat) = StringMap.find pname delta in
             if not (eqType typ_scrutinee typ_of_pat)
             then raise (TypeError ("scrutinee in case has type \"" ^ sname
                                    ^ "\" but a branch is a pattern of type "
                                    ^ pname ^ " \""))
             else () (* success *)
       in
       (* typeCheckRHS to be mapped over branches.
       1. extends rhs environments with pattern-introduced names and types
       2. typechecks rhss
       3. ensure all types are equal *)
       let typeCheckRHS pattern rhs =
         (* extends gamma with bindings introduced by pat *)
             let extended_gamma =
               extendGammaWithPat typ_scrutinee gamma delta pattern in
             (* print_endline "Typechecking rhs with gamma: \n";
             StringMap.iter (fun s t -> print_endline (s ^ " -> " ^ tyString
↪  t))
```

```
              extended_gamma ; *)
              typeof extended_gamma delta rhs
            in

        (* make bindings over pattern types *)
        let _           = List.iter typeCheckPattern patterns in
        let typs_rhss   = List.map2 typeCheckRHS patterns rhss in

        let typ_fst_rhs = List.hd typs_rhss in
        (* check all rhs's to be of the same type *)
        let check_rhs_ty_match rhs' =
          if not (eqType rhs' typ_fst_rhs)
          then raise (TypeError ("a case expression's first branch has type "
                      ^ tyString typ_fst_rhs ^ ", but a later branch has type "
                      ^ tyString rhs'))
        in
        let _ = List.iter check_rhs_ty_match typs_rhss in
        typ_fst_rhs
        | _ -> raise (TypeError
                        ("expected custom datatype in case expression but got "
                        ^ tyString typ_scrutinee)))
    in typ expr

let addWildcard ty = function
    | [] -> raise (Impossible "empty pat list")
    | pats ->
      if not (List.exists (function | (T.Name (_, false), _) -> true
                                    | _ -> false) pats)
      then List.append pats [(T.Name ("__MATCH_FAILCASE__", false),
                              (T.Err "pattern match failed", ty))]
    else pats

let rec exp gamma delta expr =
  let typeof' = typeof gamma delta in
  let rec exp' e =
    (begin
    match e with
    | U.Literal l -> T.Literal l
    | U.Local n   -> let _ = typeof' e in T.Local  n
    | U.Global n  -> let _ = typeof' e in T.Global n
    | U.Case (ex, branches) ->
        let ty_branch     = typeof' e in
        let ty_ex = typeof' ex in
        let (pats, rhss)  = List.split branches in
        let patconvert    = function
```

```
          | U.Name (n, isWildcard) -> T.Name (n, isWildcard)
          | U.Pattern (n, ns) ->
              let (vartys, _) = StringMap.find n delta in
              let names_tys = List.combine ns vartys in
              T.Pattern (n, names_tys)
          in
        let rhs_es =
          List.map2 (fun pat rhs ->
                        let extended_gamma =
                            extendGammaWithPat ty_ex gamma delta pat in
                            exp extended_gamma delta rhs) pats rhss in
        let pats'           = List.map patconvert pats in
        let branches'       = List.combine pats' rhs_es in
        (* let branches'      = List.map (fun (pat, (e, t)) ->
                                        T.CaseBranch (pat, (e, t)))
                             branches_full in   *)
        (match ty_ex with (U.CustomTy n) ->
        (* extract all value constructors from gamma *)
        let possibleVariants = StringMap.fold
        (fun vconname (_, ty) variants ->
          match ty with U.CustomTy n' when n = n' -> vconname::variants
          | _ -> variants) delta [] in
          let prunedbranches = pruneBranchesWith branches' possibleVariants in
          T.Case (exp' ex, addWildcard ty_branch prunedbranches)
          | _ -> raise (Impossible "failed to extract custom name from type"))
    | U.If (e1, e2, e3) ->
        let _ = typeof' e in
          T.If (exp' e1, exp' e2, exp' e3)
    | U.Let (n, e1, e') ->  let ty_e    = typeof' e1 in
                            let gamma' = StringMap.add n ty_e gamma in
                            let _       = typeof gamma' delta e' in
                            T.Let (n, exp gamma delta e1,
                                            exp gamma' delta e')
    | U.Apply (e, es) as app -> let _ = typeof' app in
                                  let es' = List.map exp' es in
                                  T.Apply (exp' e, es')
  | U.Dup n -> let _ = typeof' e in T.Dup n
  end
      , typeof' e)
  in exp' expr

let typecheckDef (defs, gamma, delta) = function
| U.Define (n, args, body) ->
  if not (StringMap.mem n gamma)
  then raise (TypeError
              ("definition of function \"" ^ n
              ^ "\" with no prior type annotation."))
  else let known_annotated_ty = StringMap.find n gamma in
  (match known_annotated_ty with
    | (U.FunTy (argtys, expected_ret_ty)) ->
      let known_argscount = List.length argtys in
      let given_argscount = List.length args in
      if not (known_argscount = given_argscount)
      then raise (TypeError ("prior annotation defined function \""
                      ^ n ^ "\" has " ^ Int.to_string known_argscount
                      ^ " arguments, but its definition has "
                      ^ Int.to_string given_argscount ^ " arguments."))
      else
      let extended_gamma =
        List.fold_left2 (* insane folding *)
            (fun env name ty -> StringMap.add name ty env) gamma args argtys in
      let ret_ty = typeof extended_gamma delta body in
        if not (eqType expected_ret_ty ret_ty)
        then raise (TypeError ("prior annotation defined function \""
                        ^ n ^ "\" to be of type \""
                        ^ tyString known_annotated_ty
                        ^ "\" but a definition was given that has type \""
                        ^ tyString (U.FunTy (argtys, ret_ty )) ^ "\""))
      else
        let funty = Uast.FunTy (argtys, ret_ty) in
        let def' = T.Define (n, funty, args, exp extended_gamma delta body) in
        (List.append defs [def'], gamma, delta)
    | _ -> raise (Impossible "found non-func name in top-level environment"))
| U.Datatype (n, variants) ->
  let check_variant delta' (vname, ts)  =
    if not (StringMap.mem vname delta')
    then    StringMap.add vname (ts, U.CustomTy n) delta'
    else let (_, existing_type) = StringMap.find vname delta' in
        raise (TypeError ("duplicate type constructor \""
                    ^ vname ^ "\" in user-defined datatype \""
                    ^ n ^ "\": constructor already exists for type \""
                    ^ tyString existing_type ^ "\""))
    in let extended_delta = List.fold_left check_variant delta variants in
    let add_variant gamma' (vname, ts) =
      StringMap.add vname (Uast.FunTy (ts, U.CustomTy n)) gamma'
    in
    let extended_gamma = List.fold_left add_variant gamma variants in
    let datatype' = T.Datatype (n, variants) in
      List.append defs [datatype'], extended_gamma, extended_delta
```

```
| U.TyAnnotation (n, ty) ->
  if not (StringMap.mem n gamma)
  then let extended_gamma = StringMap.add n ty gamma in
  (defs, extended_gamma, delta)
  else let found_typ = StringMap.find n gamma in
    if not (eqType ty found_typ)
    then raise (TypeError ("prior annotation defined function \"" ^ n ^
    "\" to be of type \""
    ^ tyString found_typ
    ^ "\" but a second annotation was given that has type \""
    ^ tyString ty ^ "\""))

else (defs, gamma, delta)
(* walks the program, building environments and typechecking against them. *)
let typecheck prog =
  let gamma =
    let prim_constraints = List.fold_right
        (fun (prim_name, ty) -> StringMap.add prim_name ty)
        Primitives.primitives StringMap.empty
    in
    let fun_constraints = List.fold_right
        (function
          | U.TyAnnotation (n, ty) -> StringMap.add n ty
          | _ -> fun env -> env)
        prog StringMap.empty
    in
    StringMap.union
      (fun n _ _ -> raise
                      (TypeError ("attempted to define a function with "
                                  ^ "name \"" ^ n ^ "\", but a primitive "
                                  ^ "function with that name already exists.")))
      prim_constraints fun_constraints
  in
  let delta = StringMap.empty in
  let defs = [] in
  List.fold_left typecheckDef (defs, gamma, delta) prog
```

8.3.6 normalize.ml

```
(* Author: Jasper Geer *)

module T = Tast
module N = Nast

let typeof (_, ty) = ty
let rec normalize_expr (expr, ty) = match expr with
  | T.Err n -> N.Err (ty, n)
  | T.Literal l -> N.Literal l
  | T.Local n -> N.Local n
  | T.Global n -> N.Global n
  | T.Case (e, branches) ->
    let pattern = function
      | T.Pattern (n, ns) -> N.Pattern (n, ns)
      | T.Name (n, _) -> N.Name n
    in
    let branches' =
      List.map (function (pat, e) -> (pattern pat, normalize_expr e)) branches
      ↪  in
    let e' = normalize_expr e in
    let n = Freshnames.fresh_name () in
    (N.Let (n, typeof e, e',
            N.Case (ty, n, branches')))
  | T.If (e1, e2, e3) ->
    let e1' = normalize_expr e1 in
    let e2' = normalize_expr e2 in
    let e3' = normalize_expr e3 in
    let n = Freshnames.fresh_name () in
    (N.Let (n, typeof e1, e1',
            N.If (n, e2', e3')))
  | T.Let (n, e, body) ->
    N.Let (n, typeof e, normalize_expr e, normalize_expr body)
  | T.Apply (e1, es) ->
    let e1' = normalize_expr e1 in
    let fun_name = Freshnames.fresh_name () in
    let names = List.init (List.length es) (fun _ -> Freshnames.fresh_name ())
    ↪  in
    let binds = List.combine names es in
    let apply = N.Let (fun_name, typeof e1, e1', N.Apply (fun_name, names)) in
    List.fold_right (fun (n, e) acc -> N.Let (n, typeof e, normalize_expr e,
    ↪  acc)) binds apply
  | T.Dup n -> N.Dup (ty, n)

let normalize_def = function
  | T.Define (n, ty, params, body) -> N.Define (n, ty, params, normalize_expr
  ↪  body)
  | T.Datatype (n, variants) -> N.Datatype (n, variants)

let normalize = List.map normalize_def
```

8.3.7 consumptioncheck.ml

```
(* Consumption Checker *)

(* Author: Jasper Geer
 * Edited by: Randy Dang, Jackson Warhover
 *)
module N = Nast
module F = Fast
module T = Tast

module S = Set.Make(String)
module StringMap = Map.Make(String)

let unions sets = List.fold_right S.union sets S.empty

exception NameAlreadyConsumed of string
exception Impossible of string

(* Check affine-typeness *)

let is_dataty = function
  | Uast.CustomTy _ -> true
  | _ -> false

let rec check_affine live dead =
  let has_dataty n = is_dataty (StringMap.find n live) in
  let check_name n =
    if S.mem n dead && has_dataty n
    then raise (NameAlreadyConsumed n)
    else () in
  function
  | N.Err _ -> dead
  | N.Local n -> check_name n; S.add n dead
  | N.Global _ -> dead
  | N.Dup _ -> dead
  | N.Literal _ -> dead
  | N.If (n, e1, e2) ->
    check_name n;
    let dead' = S.add n dead in
    let d1 = check_affine live dead' e1 in
    let d2 = check_affine live dead' e2 in
    S.union d1 d2
  | N.Let (n, ty, e, body) ->
    let dead' = check_affine live dead e in
    let live' = StringMap.add n ty live in
    check_affine live' dead' body
  | N.Case (scrutinee_ty, scrutinee, branches) ->
    check_name scrutinee;
    let dead' = S.add scrutinee dead in
    let branch (pattern, body) = match pattern with
      | N.Pattern (_, binds) ->
        let live' = List.fold_right
            (fun (n, ty) acc -> StringMap.add n ty acc) binds live
        in
        check_affine live' dead' body
      | N.Name n ->
        let live' = StringMap.add n scrutinee_ty live in
        check_affine live' dead' body
    in
    unions (List.map branch branches)
  | N.Apply (n, ns) ->
    List.iter check_name (n :: ns);
    List.fold_right (fun n acc -> S.add n acc) (n :: ns) dead

let rec not_free =
  function
  | N.Err _ -> S.empty
  | N.Local n -> S.singleton n
  | N.Global _ -> S.empty
  | N.Dup _ -> S.empty
  | N.Literal _ -> S.empty
  | N.If (n, e1, e2) -> S.add n (unions [not_free e1; not_free e2])
  | N.Let (_, _, e, body) -> unions [not_free e; not_free body]
  | N.Apply (n, ns) -> S.of_list (n :: ns)
  | N.Case (_, n, branches) -> S.add n (unions (List.map (function (_, branch)
  ↪  -> not_free branch) branches))


let merge = StringMap.union
    (fun n ty1 ty2 ->
        if ty1 = ty2
        then Some ty1
        else raise (Impossible ("type of " ^ n ^ " is not consistent.")))

let consume_in to_consume expr =
  StringMap.fold (fun n ty acc ->
      if is_dataty ty
      then F.FreeRec (ty, n, acc)
      else acc) to_consume expr
```

```
(* Automatically insert free directives into well-typed code *)

let rec insert_frees to_consume =
  function
  | N.If (n, e1, e2) ->
    let to_consume' = StringMap.remove n to_consume in
    let e1' = insert_frees to_consume' e1 in
    let e2' = insert_frees to_consume' e2 in
    F.If (F.Local n, e1', e2')
  | N.Case (ty, scrutinee, branches) ->
    let to_consume' = StringMap.remove scrutinee to_consume in
    let branch (pattern, body) = match pattern with
      | N.Pattern (tag, binds) ->
        let introduced =
          List.fold_right
            (fun (n, ty) acc -> StringMap.add n ty acc) binds StringMap.empty
        in
        let body' = insert_frees (merge introduced to_consume') body in
        (F.Pattern (tag, binds), F.Free (ty, scrutinee, body'))
      | N.Name n ->
        let body' = insert_frees (StringMap.add n ty to_consume) body in
        (F.Name n, F.Let (n, F.Local scrutinee, body'))
    in
    F.Case (F.Local scrutinee, List.map branch branches)
  | N.Let (n, ty, e, body) ->
    let consumed_in_e = not_free e in
    let to_consume_e = StringMap.filter (fun n _ -> S.mem n consumed_in_e)
    ↪  to_consume in
    let to_consume_body = StringMap.add n ty
        (StringMap.filter (fun n _ -> not (S.mem n consumed_in_e)) to_consume)
    in
    let e' = insert_frees to_consume_e e in
    let body' = insert_frees to_consume_body body in
    F.Let (n, e', body')
  | N.Apply (n, ns) ->
    let to_consume' = List.fold_right (fun n acc -> StringMap.remove n acc) (n
    ↪  :: ns) to_consume in
    consume_in to_consume' (F.Apply (F.Local n, List.map (fun n -> F.Local n)
    ↪  ns))
  | N.Dup (_, n) when StringMap.mem n to_consume ->
    let to_consume' = StringMap.remove n to_consume in
    consume_in to_consume' (F.Local n)
  | N.Dup (ty, n) ->
    let to_consume' = StringMap.remove n to_consume in
    consume_in to_consume' (F.Dup (ty, n))
  | N.Literal l -> consume_in to_consume (F.Literal l)
  | N.Global n -> consume_in to_consume (F.Global n)
  | N.Local n ->
    let to_consume' = StringMap.remove n to_consume in
    consume_in to_consume' (F.Local n)
  | N.Err (ty, msg) -> consume_in to_consume (F.Err (ty, msg))

let insert_frees_def =
  function
  | N.Define (fun_name, fun_ty, params, body) ->
    let param_tys = match fun_ty with
                    | Uast.FunTy (param_tys, _) -> param_tys
                    | _ -> raise (Impossible "function does not have function
                    ↪  type")
    in
    let typed_params = List.combine params param_tys in
    let params_to_consume = List.fold_right
        (fun (n, ty) acc -> StringMap.add n ty acc) typed_params StringMap.empty
        ↪  in
    let _ = check_affine params_to_consume S.empty body in
    let body' = insert_frees params_to_consume body in
    F.Define(fun_name, fun_ty, params, body')
  | N.Datatype (n, variants) -> F.Datatype (n, variants)

let consumption_check = List.map insert_frees_def
```

8.3.8 memorymanage.ml

```
(* Converts explicit free AST to explicit memory managed AST *)

(* Author: Randy Dang
 * Edited by: Jasper Geer, Jackson Warhover
 *)

(*
 * Notes: Convert types to LLVM types
 * Generate alloc and free functions for each datatype
 * Convert all dup calls to the appropriate alloc function (and
 * use the "primitive" alloc where necessary)
 * Convert all freerec calls to the appropriate free function
 * Keep all free calls the way they are
 * Delete datatype definitions
 * Prefix user functions with "_"
 *)
```

```
module F = Fast
module M = Mast

module StringMap = Map.Make(String)

let typeof (_, ty) = ty

let id x = x

(* Defined in case we want to throw an error *)
exception InvalidDup of string
exception Unimplemented of string
exception Impossible of string

(* Convert Compost type `ty` to the appropriate LLVM type *)
let rec convert_builtin_ty ty =
    match ty with
    | Uast.FunTy(tylist, ty) ->
        let convert_param_ty = function
            | Uast.FunTy _  as fun_ty -> M.Ptr(convert_builtin_ty fun_ty)
            | other_ty -> convert_builtin_ty other_ty
        in
        M.Fun(convert_builtin_ty ty, List.map convert_param_ty tylist)
    | Uast.Unit -> M.Int(1)
    | Uast.Int -> M.Int(32) (* 32-bit integer *)
    | Uast.Bool -> M.Int(1) (* 1-bit integer *)
    | Uast.Sym -> M.Ptr(Int(8)) (* pointer to a 8-bit integer *)
    | Uast.CustomTy(_) -> raise (Impossible "Erroneously called
    ↪  convert_builtin_ty on a custom datatype")

(* Convert an integer to a generated variable name corresponding to that
↪  integer, e.g. 1 -> "var1" *)
let varname_of_int int = "var" ^ string_of_int int

let mast_of_fast fast =
    (* Get all datatype definitions *)
    let datatypes =
        let add_datatype map def =
            match def with
            | F.Datatype(name, variants) -> StringMap.add name variants map
            | _ -> map
        in
        List.fold_left add_datatype StringMap.empty fast
    in
    (* Get all variant constructor definitions and generate indices *)
```

```
let variant_tags =
    let add_variant (map, tag) (name, _) = (StringMap.add name tag map, tag
    ↪  + 1) in
    let add_datatype_variant map def =
        match def with
        | F.Datatype(_, variants) ->
            let (map', _) = List.fold_left add_variant (map, 0) variants in
            map'
        | _ -> map (* No change if not a datatype definition *)
    in
    List.fold_left add_datatype_variant StringMap.empty fast
in
(* Convert fast ty to mast ty *)
let rec convert_ty ty =
    match ty with
    (* LLVM type for a custom type is a pointer to a struct containing a
      * 32-bit variant identifier (i.e. tag), followed by £n£ i64s, where
      * £n£ equals the maximum number of arguments for a variant constructor
      * of this type
      *)
    | Uast.CustomTy(tyname) ->
        let variants = StringMap.find tyname datatypes in
        (* Get maxnum_variantargs, which is the maximum possible number of
        ↪  arguments to
          * a variant constructor of this type
          *)
        let update_maxnum_variantargs currmax currvariant =
            let (_, currvarianttys) = currvariant in
            max currmax (List.length currvarianttys)
        in
        let maxnum_variantargs = List.fold_left update_maxnum_variantargs 0
        ↪  variants in
        M.Ptr(M.Struct(M.Int(32) :: List.init maxnum_variantargs (fun _ ->
        ↪  M.Int(64))))
    | Uast.FunTy(tylist, ty) ->
        let convert_param_ty = function
            | Uast.FunTy _  as fun_ty -> M.Ptr(convert_ty fun_ty)
            | other_ty -> convert_ty other_ty
        in
        M.Fun(convert_param_ty ty, List.map convert_param_ty tylist)
    | _ -> convert_builtin_ty ty
in
(* Convert fast expr to mast expr *)
let rec convert_expr fast_expr =
    match fast_expr with
    | F.Literal(lit) -> M.Literal(lit)
    | F.Local(name) -> M.Local(name)
    | F.Global("main") -> M.Global("main")
    | F.Global(name) when List.mem_assoc name Primitives.primitives ->
    ↪  M.Global(name)
    | F.Global(name) -> M.Global("_" ^ name)
    | F.Case(expr, casebranches) ->
        let convert_casebranch (pattern, pexpr) =
            let convert_pattern pattern =
                match pattern with
                | F.Pattern(name, names) ->
                    M.Pattern(StringMap.find name variant_tags, List.map
                     ↪  (fun (name, ty) -> (name, convert_ty ty)) names)
                | F.Name n -> M.Name n
            in
            (convert_pattern pattern, convert_expr pexpr)
        in
        M.Case(convert_expr expr, List.map convert_casebranch casebranches)
    | F.If(expr1, expr2, expr3) ->
        M.If(convert_expr expr1, convert_expr expr2, convert_expr expr3)
    | F.Let(name, expr, body) -> M.Let(name, convert_expr expr, convert_expr
    ↪  body)
    | F.Apply(expr, exprlist) -> M.Apply(convert_expr expr, List.map
    ↪  convert_expr exprlist)
    | F.Dup(ty, name) ->
        (match ty with
          | CustomTy(tyname) -> M.Apply(M.Global("dup_" ^ tyname),
          ↪  [M.Local(name)])
          | _ -> M.Local(name) (* no-op for now that returns the name;
          ↪  another option is to throw InvalidDup exception *))
    | F.FreeRec(ty, name, expr) ->
        (match ty with
          | CustomTy(tyname) -> M.Let(Freshnames.fresh_name (),
          ↪  M.Apply(M.Global("free_" ^ tyname), [M.Local(name)]),
          ↪  convert_expr expr)
          | _ -> raise (Impossible "Erroneously called FreeRec on something
          ↪  that was not a custom type"))
    | F.Free(_, name, expr) -> M.Free(name, convert_expr expr)
    | F.Err (ty, msg) -> M.Err (convert_ty ty, msg)
in
(* Converts a fast definition to a _list_ of mast definitions *)
let convert_defs fast_def =
    match fast_def with
    | F.Define("main", fun_ty, params, body) ->
        [ M.Define("main", convert_ty fun_ty, params, convert_expr body) ]
```

```
| F.Define(name, fun_ty, params, body) ->
    (* Prefix each function name with "_" to guarantee it does not
    ↪  conflict with generated functions *)
    [ M.Define("_" ^ name, convert_ty fun_ty, params, convert_expr body)
    ↪  ]
| F.Datatype(name, variants) ->
    (* Generate all relevant functions for this datatype *)
    let data_ty = match convert_ty (Uast.CustomTy(name)) with
      | M.Ptr ty -> ty
      | _ -> raise (Impossible "custom type is not pointer to struct")
    in
    let data_ty_ptr = M.Ptr data_ty in
    let dup_func =
        let func_type = M.Fun(data_ty_ptr, [data_ty_ptr]) in
        let param_names = ["instance"] in
        let body =
            let gen_casebranch variant_idx (_, variant_tys) =
                (* Let variant_varnames just be a sequence of integers
                ↪  starting from 0, prepended by "var" *)
                let variant_varnames = List.init (List.length
                ↪  variant_tys) varname_of_int in
                let pattern = M.Pattern(variant_idx, List.combine
                ↪  variant_varnames (List.map convert_ty variant_tys))
                ↪  in
                let expr =
                    let alloc_ty index ty =
                    match ty with
                      | Uast.CustomTy(name) -> (index + 1,
                      ↪  M.Apply(Global("dup_" ^ name),
                      ↪  [Local(varname_of_int index)]))
                      | _ -> (index + 1, Local(varname_of_int index))
                    in
                    let (_, alloc_expr) = List.fold_left_map alloc_ty 0
                    ↪  variant_tys
                    in
                    M.Alloc(data_ty, variant_idx, alloc_expr)
                in
                (variant_idx + 1, (pattern, expr))
            in
            let (_, casebranches) = List.fold_left_map gen_casebranch 0
            ↪  variants in
            let casebranches' = List.append casebranches [(M.Name
            ↪  (Freshnames.fresh_name ()), M.Err (data_ty_ptr,
            ↪  "IMPOSSIBLE: inexhaustive match in dup"))] in
            M.Case(Local("instance"), casebranches')
        in
        M.Define("dup_" ^ name, func_type, param_names, body)
    in
    let free_func =
        let func_type = M.Fun(convert_ty Uast.Unit, [data_ty_ptr]) in
        let param_names = ["instance"] in
        let body =
            let gen_casebranch variant_idx (_, variant_tys) =
                (* Let variant_varnames just be a sequence of integers
                ↪  starting from 0, prepended by "var" *)
                let variant_varnames = List.init (List.length
                ↪  variant_tys) varname_of_int in
                let pattern = M.Pattern(variant_idx, List.combine
                    ↪  variant_varnames (List.map convert_ty variant_tys))
                    ↪  in
                let expr =
                    let unitlit = M.Literal(Ast.UnitLit) in
                    let gen_free_call varname ty =
                        match ty with
                      | Uast.CustomTy(name) ->
                      ↪  M.Apply(Global("free_" ^ name),
                      ↪  [Local(varname)])
                      | _ -> unitlit
                    in
                    (* Get all calls to free functions *)
                    let free_calls = List.filter ((<>) unitlit)
                    ↪  (List.map2 gen_free_call variant_varnames
                    ↪  variant_tys) in
                    let rec free_expr_of_calls free_calls =
                        match free_calls with
                      | [] -> unitlit
                      | [call] -> call
                      | call :: calls -> M.Let(Freshnames.fresh_name
                      ↪  (), call, free_expr_of_calls calls)
                    in
                    M.Free("instance", free_expr_of_calls free_calls)
                in
                (variant_idx + 1, (pattern, expr))
            in
            let (_, casebranches) = List.fold_left_map gen_casebranch 0
            ↪  variants  in
            let casebranches' = List.append casebranches [(M.Name
            ↪  (Freshnames.fresh_name ()), M.Err (M.Int 1, "IMPOSSIBLE:
            ↪  inexhaustive match in dup"))] in
            M.Case(Local("instance"), casebranches')
            in
            M.Define("free_" ^ name, func_type, param_names, body)
        in
        let alloc_variant_funcs =
            let alloc_variant_func (variant_name, variant_tys) =
                (* Let argument names of function just be a sequence of
                ↪  integers starting from 0, prepended by "var" *)
                let func_type = M.Fun(data_ty_ptr, (List.map convert_ty
                ↪  variant_tys)) in
                let func_argnames = List.init (List.length variant_tys)
                ↪  varname_of_int in
                let alloc_ty index _ = (index + 1, M.Local(varname_of_int
                ↪  index)) in
                let (_, alloc_expr) = List.fold_left_map alloc_ty 0
                ↪  variant_tys in
                let alloc_call = M.Alloc(data_ty, StringMap.find
                ↪  variant_name variant_tags, alloc_expr) in
                M.Define("_" ^ variant_name, func_type, func_argnames,
                ↪  alloc_call)
            in
            let variants = StringMap.find name datatypes in
            List.map alloc_variant_func variants
        in
        dup_func :: free_func :: alloc_variant_funcs
in
let defs = List.map convert_defs fast in
List.flatten defs
```

8.3.9 codegen.ml

```
(* Author: Jasper Geer
 * Edited by: Randy Dang
 *)

module L = Llvm
module M = Mast
module P = Primitives

module StringMap = Map.Make(String)
module StringSet = Set.Make(String)

let ctx = L.create_context

exception Impossible of string
```

```
let codegen program =
  let context = L.global_context () in
  let i64_t = L.i64_type context
  and i32_t = L.i32_type context
  and i8_t = L.i8_type context
  and i1_t = L.i1_type context
  and void_t = L.void_type context
  and the_module = L.create_module context "Compost" in

  let rec lltype_of_ty = function
    | M.Int i -> L.integer_type context i
    | M.Fun (ret_ty, param_tys) ->
      let ret_ty' = lltype_of_ty ret_ty in
      let param_tys' = List.map lltype_of_ty param_tys in
      L.function_type ret_ty' (Array.of_list param_tys')
    | M.Struct tys ->
      let tys' = List.map lltype_of_ty tys in
      L.struct_type context (Array.of_list tys')
    | M.Ptr ty -> L.pointer_type (lltype_of_ty ty)
  in
  let unions sets = List.fold_right StringSet.union sets StringSet.empty in

  let symbols =
    let rec get_sym_lits = function
      | M.Literal (Ast.SymLit str) -> StringSet.singleton str
      | M.Case (e, branches) ->
        let branch_lits = unions (List.map (fun (_, e) -> get_sym_lits e)
        ↪  branches) in
        StringSet.union (get_sym_lits e) branch_lits
      | M.If (e1, e2, e3) -> unions [get_sym_lits e1; get_sym_lits e2;
      ↪  get_sym_lits e3]
      | M.Let (_, e1, e2) -> unions [get_sym_lits e1; get_sym_lits e2]
      | M.Apply (e, args) -> unions ((get_sym_lits e) :: (List.map get_sym_lits
      ↪  args))
      | M.Free (_, e) -> get_sym_lits e
      | _ -> StringSet.empty
    in
    let sym_lits = unions (List.map (fun (M.Define (_, _, _, body)) ->
    ↪  get_sym_lits body) program) in
    let build_symbol sym_lit syms =
      let sym_value = L.const_stringz context sym_lit in
      let sym_var = L.define_global "sym_lit" sym_value the_module in
      StringMap.add sym_lit sym_var syms
    in
    StringSet.fold build_symbol sym_lits StringMap.empty
```

```
in

let abort_t = L.function_type void_t [| |] in
let abort_func = L.declare_function "abort" abort_t the_module in

let free_t = L.function_type void_t [| L.pointer_type i8_t |] in
let free_func = L.declare_function "free" free_t the_module in

(* Association list of primitive functions names and how to build them *)
let primitives =
  let printf_t = L.var_arg_function_type i32_t [| L.pointer_type i8_t |] in
  let printf_func = L.declare_function "printf" printf_t the_module in

  let getchar_t = L.function_type i32_t [| |] in
  let getchar_func = L.declare_function "getchar" getchar_t the_module in

  let unit_value = L.const_int i1_t 0 in
  [
    ("print-newline", fun builder ->
        function
        | [| |] ->
          let newline = L.build_global_stringptr "\n" "newline" builder in
          let _ = L.build_call printf_func [| newline |] "tmp" builder in
          unit_value
        | _ -> raise (Impossible "print-sym has 1 parameter")

    );
    ("print-sym", fun builder ->
        function
        | [| s |] ->
          let _ = L.build_call printf_func [| s |] "tmp" builder in
          unit_value
        | _ -> raise (Impossible "print-sym has 1 parameter")
    );
    ("print-int", fun builder ->
      function
      | [| i |] ->
          let fmt_int = L.build_global_stringptr "%d" "fmt_int" builder in
          let _ = L.build_call printf_func [| fmt_int; i |] "tmp" builder in
          unit_value
      | _ -> raise (Impossible "print-int has 1 parameter")
    );
    ("print-ascii", fun builder ->
      function
      | [| i |] ->
```

```
          let fmt_int = L.build_global_stringptr "%c" "fmt_char" builder in
          let _ = L.build_call printf_func [| fmt_int; i |] "tmp" builder in
          unit_value
      | _ -> raise (Impossible "print-int has 1 parameter")
  );
  ("print-bool", fun builder ->
        function
        | [| b |] ->
          let true_str = L.build_global_stringptr "true" "true" builder in
          let false_str = L.build_global_stringptr "false" "false" builder in
          let to_print = L.build_select b true_str false_str "tmp" builder in
          let _ = L.build_call printf_func [| to_print |] "tmp" builder in
          unit_value
        | _ -> raise (Impossible "print-bool has 1 parameter")
  );
  ("print-unit", fun builder ->
        function
        | [| _ |] ->
          let unit_str = L.build_global_stringptr "unit" "unit" builder in
          let _ = L.build_call printf_func [| unit_str |] "tmp" builder in
          unit_value
        | _ -> raise (Impossible "print-unit has 1 parameter")
  );
  ("in", fun builder ->
        function
        | [| |] -> L.build_call getchar_func [| |] "tmp" builder
        | _ -> raise (Impossible "in has no parameters")
  );

  (* Equality *)
  ("=i", fun builder ->
        function
        | [| a; b |] -> L.build_icmp L.Icmp.Eq a b "tmp" builder
        | _ -> raise (Impossible "i= has 2 parameters")
  );
  ("=s", fun builder ->
        function
        | [| a; b |] -> L.build_icmp L.Icmp.Eq a b "tmp" builder
        | _ -> raise (Impossible "s= has 2 parameters")
  );
  ("=b", fun builder ->
        function
        | [| a; b |] -> L.build_icmp L.Icmp.Eq a b "tmp" builder
        | _ -> raise (Impossible "b= has 2 parameters")
  );
```

```
("=u", fun _ ->
    function
    | [| _; _ |] -> L.const_int i1_t 1
    | _ -> raise (Impossible "u= has 2 parameters")
);

(* Arithmetic *)
("+", fun builder ->
    function
    | [| a; b |] -> L.build_add a b "tmp" builder
    | _ -> raise (Impossible "+ has 2 parameters")
);
("-", fun builder ->
    function
    | [| a; b |] -> L.build_sub a b "tmp" builder
    | _ -> raise (Impossible "- has 2 parameters")
);
("*", fun builder ->
    function
    | [| a; b |] -> L.build_mul a b "tmp" builder
    | _ -> raise (Impossible "* has 2 parameters")
);
("/", fun builder ->
    function
    | [| a; b |] -> L.build_sdiv a b "tmp" builder
    | _ -> raise (Impossible "/ has 2 parameters")
);
("udiv", fun builder ->
    function
    | [| a; b |] -> L.build_udiv a b "tmp" builder
    | _ -> raise (Impossible "udiv has 2 parameters")
);
("%", fun builder ->
    function
    | [| a; b |] -> L.build_srem a b "tmp" builder
    | _ -> raise (Impossible "% has 2 parameters")
);
("umod", fun builder ->
    function
    | [| a; b |] -> L.build_urem a b "tmp" builder
    | _ -> raise (Impossible "umod has 2 parameters")
);
("neg", fun builder ->
    function
    | [| a |] -> L.build_neg a "tmp" builder
    | _ -> raise (Impossible "neg has 2 parameters")
);

(* Comparison *)
(">", fun builder ->
    function
    | [| a; b |] -> L.build_icmp L.Icmp.Sgt a b "tmp" builder
    | _ -> raise (Impossible "> has 2 parameters")
);
("<", fun builder ->
    function
    | [| a; b |] -> L.build_icmp L.Icmp.Slt a b "tmp" builder
    | _ -> raise (Impossible "< has 2 parameters")
);
(">=", fun builder ->
    function
    | [| a; b |] -> L.build_icmp L.Icmp.Sge a b "tmp" builder
    | _ -> raise (Impossible ">= has 2 parameters")
);
("<=", fun builder ->
    function
    | [| a; b |] -> L.build_icmp L.Icmp.Sle a b "tmp" builder
    | _ -> raise (Impossible "<= has 2 parameters")
);

(* Boolean *)
("not", fun builder ->
    function
    | [| a |] -> L.build_not a "tmp" builder
    | _ -> raise (Impossible "not has 1 parameter")
);
("and", fun builder ->
  function
  | [| a; b |] -> L.build_and a b "tmp" builder
  | _ -> raise (Impossible "and has 2 parameters")
);
("or", fun builder ->
  function
  | [| a; b |] -> L.build_or a b "tmp" builder
  | _ -> raise (Impossible "and has 2 parameters")
);
("xor", fun builder ->
  function
  | [| a; b |] -> L.build_xor a b "tmp" builder
  | _ -> raise (Impossible "xor has 2 parameters")
```

```
    );

    (* Bitwise *)
    ("&", fun builder ->
        function
        | [| a; b |] -> L.build_and a b "tmp" builder
        | _ -> raise (Impossible "& has 2 parameters")
    );
    ("|", fun builder ->
        function
        | [| a; b |] -> L.build_or a b "tmp" builder
        | _ -> raise (Impossible "| has 2 parameters")
    );
    ("^", fun builder ->
        function
        | [| a; b |] -> L.build_xor a b "tmp" builder
        | _ -> raise (Impossible "^ has 2 parameters")
    );
    ("<<", fun builder ->
        function
        | [| a; b |] -> L.build_shl a b "tmp" builder
        | _ -> raise (Impossible "^ has 2 parameters")
    );
    (">>", fun builder ->
        function
        | [| a; b |] -> L.build_lshr a b "tmp" builder
        | _ -> raise (Impossible ">> has 2 parameters")
    );
    ("~", fun builder ->
        function
        | [| a |] -> L.build_not a "tmp" builder
        | _ -> raise (Impossible "~ has 1 parameter")
    );
  ]
in

(* Build the set of function declarations *)
let functions =
  let primitive_decls =
    let build_primitive (fun_name, build_fun) decls =
      let fun_ty = List.assoc fun_name P.primitives in
      let fun_lltype = lltype_of_ty (Memorymanage.convert_builtin_ty fun_ty)
      ↪  in
      let decl = L.define_function fun_name fun_lltype the_module in
      let builder = L.builder_at_end context (L.entry_block decl) in
      let return_val = build_fun builder (L.params decl) in
      let _ = L.build_ret return_val builder in
      StringMap.add fun_name decl decls
    in
    List.fold_right build_primitive primitives StringMap.empty
  in
  let function_decl (M.Define (fun_name, fun_ty, _, _)) decls =
    let fun_lltype = lltype_of_ty fun_ty in
    StringMap.add fun_name (L.define_function fun_name fun_lltype the_module)
     ↪  decls
  in
  let decls = List.fold_right function_decl program StringMap.empty in
  let name_conflict = Impossible "user-defined function and primitive function
   ↪  share the same name" in
  StringMap.union (fun _ _ -> raise name_conflict) decls primitive_decls
in
let build_function_body (M.Define (n, _, params, body)) =
  let the_function = StringMap.find n functions in
  let make_bogus_val ty builder = match L.classify_type ty with
    | L.TypeKind.Pointer ->
      let bogus_int = L.const_int i64_t (-1) in
      L.build_inttoptr bogus_int ty "bogus" builder
    | L.TypeKind.Integer ->
      let bitwidth = L.integer_bitwidth ty in
      let bogus_int = L.const_int (L.integer_type context bitwidth) (-1) in
      bogus_int
    | _ -> raise (Impossible "Non-pointer, non-integer return type")
  in

  (* Recursively build the return value of the function *)
  let rec expr is_tail locals builder =
    let non_tail = expr false in
    let tail = expr is_tail in
    let terminate ret_val builder =
      if is_tail
        then
          let _ = L.build_ret ret_val builder in
          (ret_val, builder)
        else
          (ret_val, builder)
    in
    function
    | M.Literal l ->
      let (lit_val, builder) =
      match l with
      | Ast.IntLit i -> (L.const_int i32_t i, builder)
      | Ast.SymLit s ->
        let sym = StringMap.find s symbols in
        (L.build_bitcast sym (L.pointer_type i8_t) s builder, builder)
      | Ast.BoolLit b -> if b
        then (L.const_int i1_t 1, builder)
        else (L.const_int i1_t 0, builder)
      | Ast.UnitLit -> (L.const_int i1_t 0, builder)
      in
      terminate lit_val builder
    | M.Local n ->
      let local_val = StringMap.find n locals in
      terminate local_val builder
    | M.Global n ->
      let global_val = StringMap.find n functions in
      terminate global_val builder
    | M.Let (n, e, body) ->
      let (e_val, builder') = non_tail locals builder e in
      let locals' = StringMap.add n e_val locals in
      tail locals' builder' body
    | M.Apply (M.Global n, args) when List.mem_assoc n primitives ->
      let (arg_vals, builder') = List.fold_left
          (fun (arg_vals, b) arg ->
            let (arg_val, b') = non_tail locals b arg in
            (arg_vals @ [arg_val], b')
          ) ([], builder) args in
      let call_val = List.assoc n primitives builder' (Array.of_list arg_vals)
      ↪  in
      terminate call_val builder'
    | M.Apply (f, args) ->
      let (f_val, builder') = non_tail locals builder f in
      let (arg_vals, builder'') = List.fold_left
          (fun (arg_vals, b) arg ->
            let (arg_val, b') = non_tail locals b arg in
            (arg_vals @ [arg_val], b')
          ) ([], builder') args in
      let call_val = L.build_call f_val (Array.of_list arg_vals)
       ↪  "apply_result" builder'' in
      let _ = L.set_tail_call true call_val in
      terminate call_val builder''
    | M.Free (n, e) ->
      let to_free = L.build_bitcast (StringMap.find n locals) (L.pointer_type
       ↪  i8_t) "to_free" builder in
      let _ = L.build_call free_func [| to_free |] "" builder in
      tail locals builder e
```

```
| M.If (cond, b1, b2) when is_tail ->
  let (cond_val, builder') = non_tail locals builder cond in

  let then_bb = L.append_block context "then" the_function in
  let then_builder = L.builder_at_end context then_bb in
  let (then_val, then_builder') = tail locals then_builder b1 in
  let _ = L.build_ret then_val then_builder' in

  let else_bb = L.append_block context "else" the_function in
  let else_builder = L.builder_at_end context else_bb in
  let (else_val, else_builder') = tail locals else_builder b2 in
  let _ = L.build_ret else_val else_builder' in

  let _ = L.build_cond_br cond_val then_bb else_bb builder' in

  let branch_ty = L.type_of else_val in
  let bogus_val = make_bogus_val branch_ty builder' in
  (* Throw up something for the enclosing call to use - we will never
  ↪  return this *)
  (bogus_val, builder')

| M.If (cond, b1, b2) ->
  let (cond_val, builder') = non_tail locals builder cond in

  let merge_bb = L.append_block context "merge" the_function in
  let branch_instr = L.build_br merge_bb in

  let then_bb = L.append_block context "then" the_function in
  let then_builder = L.builder_at_end context then_bb in
  let (then_val, then_builder') = non_tail locals then_builder b1 in
  let then_val' = then_val in
  let _ = branch_instr then_builder' in

  let else_bb = L.append_block context "else" the_function in
  let else_builder = L.builder_at_end context else_bb in
  let (else_val, else_builder') = non_tail locals else_builder b2 in
  let else_val' = else_val in
  let _ = branch_instr else_builder' in

  let _ = L.build_cond_br cond_val then_bb else_bb builder' in
  let merge_builder = L.builder_at_end context merge_bb in

  (L.build_phi [(then_val', L.insertion_block then_builder');
                (else_val', L.insertion_block else_builder')]
    "if_result" merge_builder, merge_builder)
```

```
| M.Alloc (ty, tag, args) ->
  let struct_val = L.build_malloc (lltype_of_ty ty) "struct" builder in
  let tag_val = L.const_int i32_t tag in
  let tag_ptr = L.build_struct_gep struct_val 0 "tag_ptr" builder in
  let _ = L.build_store tag_val tag_ptr builder in
  let (builder', _) = List.fold_left
      (fun (b, i) arg ->
        let (arg_val, b') = non_tail locals b arg in
        let convert_to_i64 v = match L.classify_type (L.type_of v) with
          | L.TypeKind.Pointer -> L.build_ptrtoint v i64_t "tmp" b'
          | _ -> L.build_zext v i64_t "tmp" b'
        in
        let converted_val = convert_to_i64 arg_val in
        let elem_ptr = L.build_struct_gep struct_val i "elem_ptr" b' in
        let _ = L.build_store converted_val elem_ptr b' in
        (b', i + 1)
      ) (builder, 1) args in
  terminate struct_val builder'
| M.Err (ty, msg) ->
  let msg_str = L.build_global_stringptr msg "err_msg" builder in
  let _ = List.assoc "print-sym" primitives builder [| msg_str |]in
  let abort_call = L.build_call abort_func [| |] "" builder in
  let _ = if is_tail then L.set_tail_call true abort_call in
  let bogus_val = make_bogus_val (lltype_of_ty ty) builder in
  let _ = L.set_tail_call true abort_call in
  terminate bogus_val builder

| M.Case (scrutinee, branches) when is_tail ->
  let (scrutinee_val, builder') = non_tail locals builder scrutinee in
  let tag_ptr = L.build_struct_gep scrutinee_val 0 "tag_ptr" builder' in
  ↪  (* error here *)
  let tag_val = L.build_load tag_ptr "tag_val" builder' in
  let default_bb = L.append_block context "default" the_function in
  let switch = L.build_switch tag_val default_bb (List.length branches)
  ↪  builder' in
  let build_branch (pat, body) = match pat with
      | M.Pattern(tag, names) ->
        let branch_bb = L.append_block context "case_branch" the_function
        ↪  in
        let branch_builder = L.builder_at_end context branch_bb in
        let convert_i64 ty v = match ty with
          | M.Int n ->
            let in_ty = L.integer_type context n in
            L.build_trunc v in_ty "tmp" branch_builder
```

```
              | _ -> L.build_inttoptr v (lltype_of_ty ty) "tmp" branch_builder
            in
            let (locals', _) =
              List.fold_left
                (fun (locals, i) (n, ty) ->
                  let arg_ptr = L.build_struct_gep scrutinee_val i "arg_ptr"
                  ↪  branch_builder in
                  let arg_val = L.build_load arg_ptr "arg_val" branch_builder
                  ↪  in
                  let converted_val = convert_i64 ty arg_val in
                  (StringMap.add n converted_val locals, i + 1)
                ) (locals, 1) names
            in
            let (body_val, _) = tail locals' branch_builder body in
            let idx_val = L.const_int i32_t tag in
            let _ = L.add_case switch idx_val branch_bb in
            body_val
          | M.Name _ ->
            let branch_builder = L.builder_at_end context default_bb in
            let (body_val, _) = tail locals branch_builder body in
            body_val
      in
      let branches' = List.map build_branch branches in
      let branch_ty = L.type_of (List.hd branches') in
      (make_bogus_val branch_ty builder', builder')

    | M.Case (scrutinee, branches) ->
      let (scrutinee_val, builder') = non_tail locals builder scrutinee in
      let tag_ptr = L.build_struct_gep scrutinee_val 0 "tag_ptr" builder' in
      ↪  (* error here *)
      let tag_val = L.build_load tag_ptr "tag_val" builder' in
      let default_bb = L.append_block context "default" the_function in
      let switch = L.build_switch tag_val default_bb (List.length branches)
      ↪  builder' in

      let merge_bb = L.append_block context "merge" the_function in
      let branch_instr = L.build_br merge_bb in

      let build_branch (pat, body) = match pat with
          | M.Pattern(tag, names) ->
            let branch_bb = L.append_block context "case_branch" the_function
            ↪  in
            let branch_builder = L.builder_at_end context branch_bb in
            let convert_i64 ty v = match ty with
              | M.Int n ->
```

```
                let in_ty = L.integer_type context n in
                L.build_trunc v in_ty "tmp" branch_builder
              | _ -> L.build_inttoptr v (lltype_of_ty ty) "tmp" branch_builder
            in
            let (locals', _) =
              List.fold_left
                (fun (locals, i) (n, ty) ->
                  let arg_ptr = L.build_struct_gep scrutinee_val i "arg_ptr"
                  ↪  branch_builder in
                  let arg_val = L.build_load arg_ptr "arg_val" branch_builder
                  ↪  in
                  let converted_val = convert_i64 ty arg_val in
                  (StringMap.add n converted_val locals, i + 1)
                ) (locals, 1) names
            in
            let (body_val, body_builder) = non_tail locals' branch_builder
            ↪  body in
            let idx_val = L.const_int i32_t tag in
            let _ = L.add_case switch idx_val branch_bb in
            let _ = branch_instr body_builder in
            (body_val, L.insertion_block body_builder)
          | M.Name _ ->
            let branch_builder = L.builder_at_end context default_bb in
            let (body_val, body_builder) = non_tail locals branch_builder body
            ↪  in
            let _ = branch_instr body_builder in
            (body_val, L.insertion_block body_builder)
      in
      let merge_builder = L.builder_at_end context merge_bb in
      let branches' = List.map build_branch branches in
      (L.build_phi branches' "case_result" merge_builder, merge_builder)

  in
  let builder = L.builder_at_end context (L.entry_block the_function) in

  let init_locals =
    let param_values = L.params the_function in
    let bindings = List.mapi (fun i n -> (n, Array.get param_values i)) params
    ↪  in
    List.fold_right (fun (n, v) m -> StringMap.add n v m) bindings
    ↪  StringMap.empty
  in
  let _ = expr true init_locals builder body in
  ()
in
```

```
  List.iter build_function_body program;
  the_module
```

8.4 Program Driver

8.4.1 compost.ml

```
(* Top-level of the Compost compiler *)

(* Authors: Randy Dang, Jasper Geer, Roger Burtonpatel, Jackson Warhover *)

(* Force dune to build some stuff *)
module D = Disambiguate
module T = Typecheck
module C = Consumptioncheck
module N = Normalize
module M = Memorymanage
module G = Codegen
module Pre = Preprocess

type action = Ast | PAst | UAst | TAst | MAst | Compile

let () =
  let action = ref Compile in
  let set_action a () = action := a in
  let speclist = [
    ("-a", Arg.Unit (set_action Ast), "Print the AST");
    ("-p", Arg.Unit (set_action PAst), "Preprocess & Print the AST");
    ("-u", Arg.Unit (set_action UAst), "Print the UAST");
    ("-t", Arg.Unit (set_action TAst), "Typecheck and print UAst");
    ("-m", Arg.Unit (set_action MAst), "Typecheck, analyze consumption, add
    ↪  explicit memory management, and print MAst");
    ("-c", Arg.Unit (set_action Compile),
      "Check and print the generated LLVM IR (default)");
  ] in
  let usage_msg = "usage: dune exec -- compost [-a|-d|-t|-m|-c] [file.com]" in
  let channel = ref stdin in
  Arg.parse speclist (fun filename -> channel := open_in filename) usage_msg;

  let lexbuf = Lexing.from_channel !channel in

  let output =
    let ast = Parser.program Scanner.token lexbuf in
    if !action = Ast then (Ast.string_of_program ast) else

    let past = Pre.preprocess ast in
    if !action = PAst then (Past.string_of_program past) else

    let uast = D.disambiguate past in
    if !action = UAst then (Uast.string_of_program uast) else

    let (type_checked, _, _) = T.typecheck uast in
    if !action = TAst then (Uast.string_of_program uast) else

    let normalized = N.normalize type_checked in

    let consumption_checked = C.consumption_check normalized in
    let memory_managed = M.mast_of_fast consumption_checked in
    if !action = MAst then (Mast.string_of_program memory_managed) else

    let m = G.codegen memory_managed in
    (* Llvm_analysis.assert_valid_module m; *)
    Llvm.string_of_llmodule m
  in print_string output
```