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 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 string
                                ;; ’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. Function values are mappings from ordered sets of values, to values. 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 map ordered sets 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 $\Gamma \vdash$ `e` : τ_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 Γ_k : `x`k : τ_k, for $k \in 1, 2, ..., n$, we say that $\Gamma_{k+1} = \Gamma_k[$`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 `e`i as consumed in both `e` and all `e`k for $k > i$.

Evaluation
For all $i \in 1, 2, ..., n$, let $\rho_{i+1} = \rho_i[$`x`$i \mapsto v_i]$, where v_i is the value returned by evaluating `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 $\Gamma[$`n`$]$.

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

Evaluation
We return the value $\rho[$`n`$]$.

3.8 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 $\Gamma[$`n`$]$.

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

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

3.9 Definitions

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

$$\begin{array}{lll} \textit{def} & ::= & \textit{val-binding} \\ & \mid & \textit{function-definition} \\ & \mid & \textit{datatype-definition} \\ & \mid & \textit{type-annotation} \\ & \mid & \textit{use-declaration} \end{array}$$

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.

3.9.1 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.

3.9.2 Val Bindings

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

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.

3.9.3 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 $\rho[\texttt{x1} \mapsto v_1, ..., \texttt{x}n \mapsto v_n]$.

3.10 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.

3.11 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.

3.12 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.

3.12.1 Equality

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

Integer equality.

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

Boolean equality.

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

Symbol equality.

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

Unit equality. Always returns `true`.

3.12.2 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.

3.12.3 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.

3.12.4 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.

3.12.5 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.

3.12.6 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.

3.12.7 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.

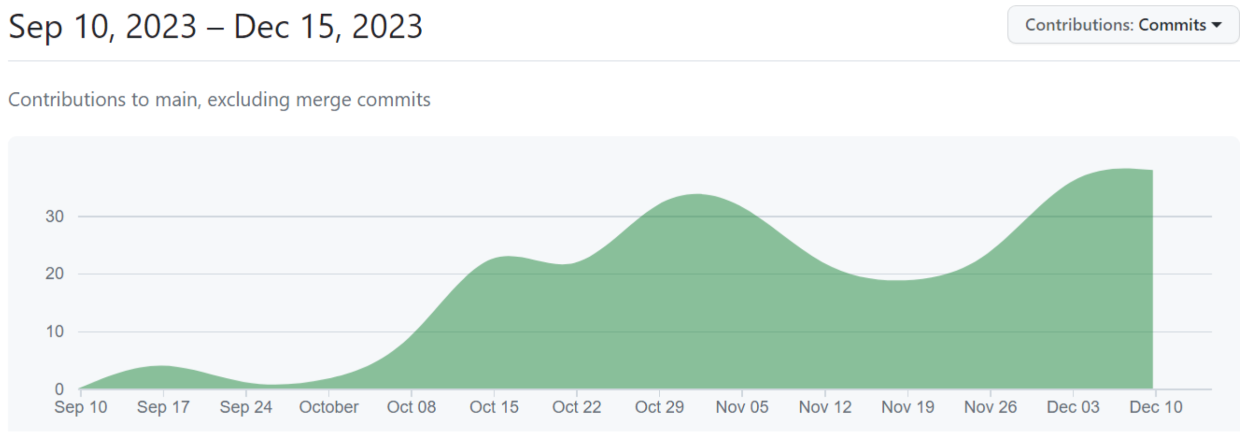
4.4 Tools Used

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).

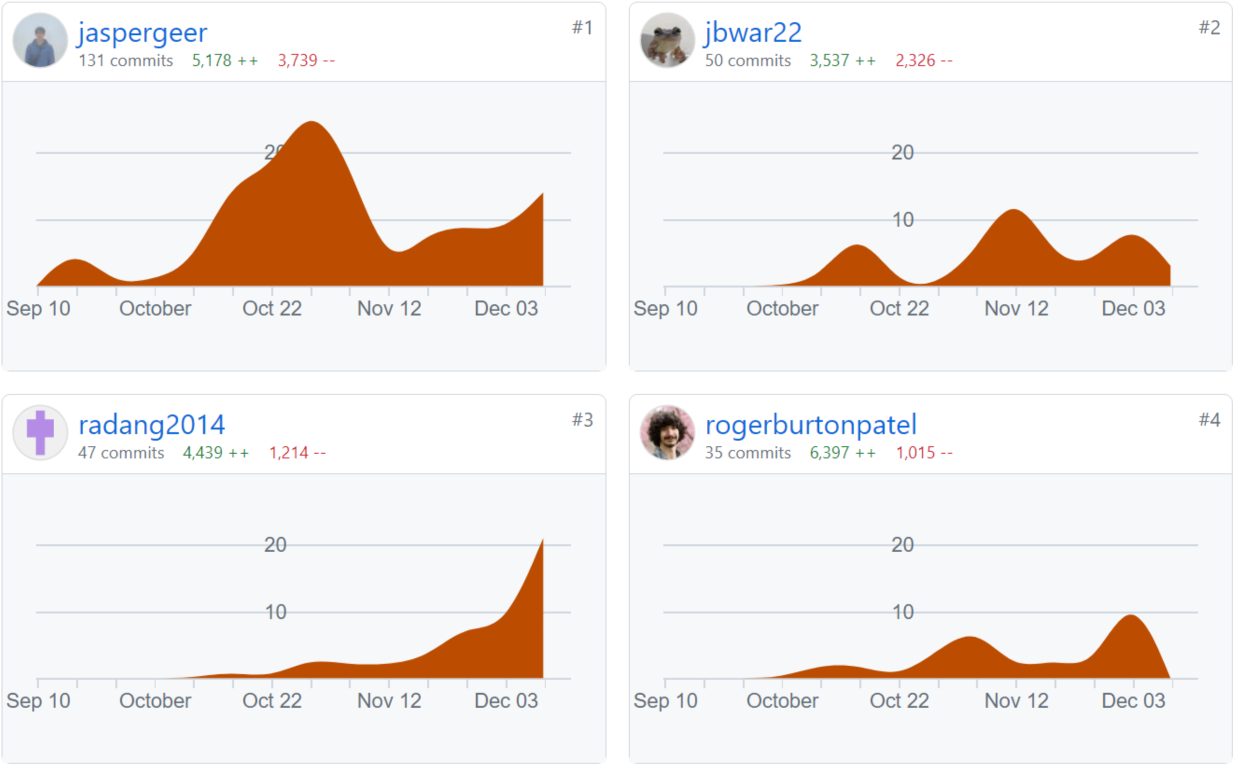
4.5 Version Control Commits

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.



Sep 10, 2023 – Dec 15, 2023

Contributions to main, excluding merge commits



jaspergeer #1
131 commits 5,178 ++ 3,739 --

jbwar22 #2
50 commits 3,537 ++ 2,326 --

radang2014 #3
47 commits 4,439 ++ 1,214 --

rogerburtonpatel #4
35 commits 6,397 ++ 1,015 --

5 Architectural Design

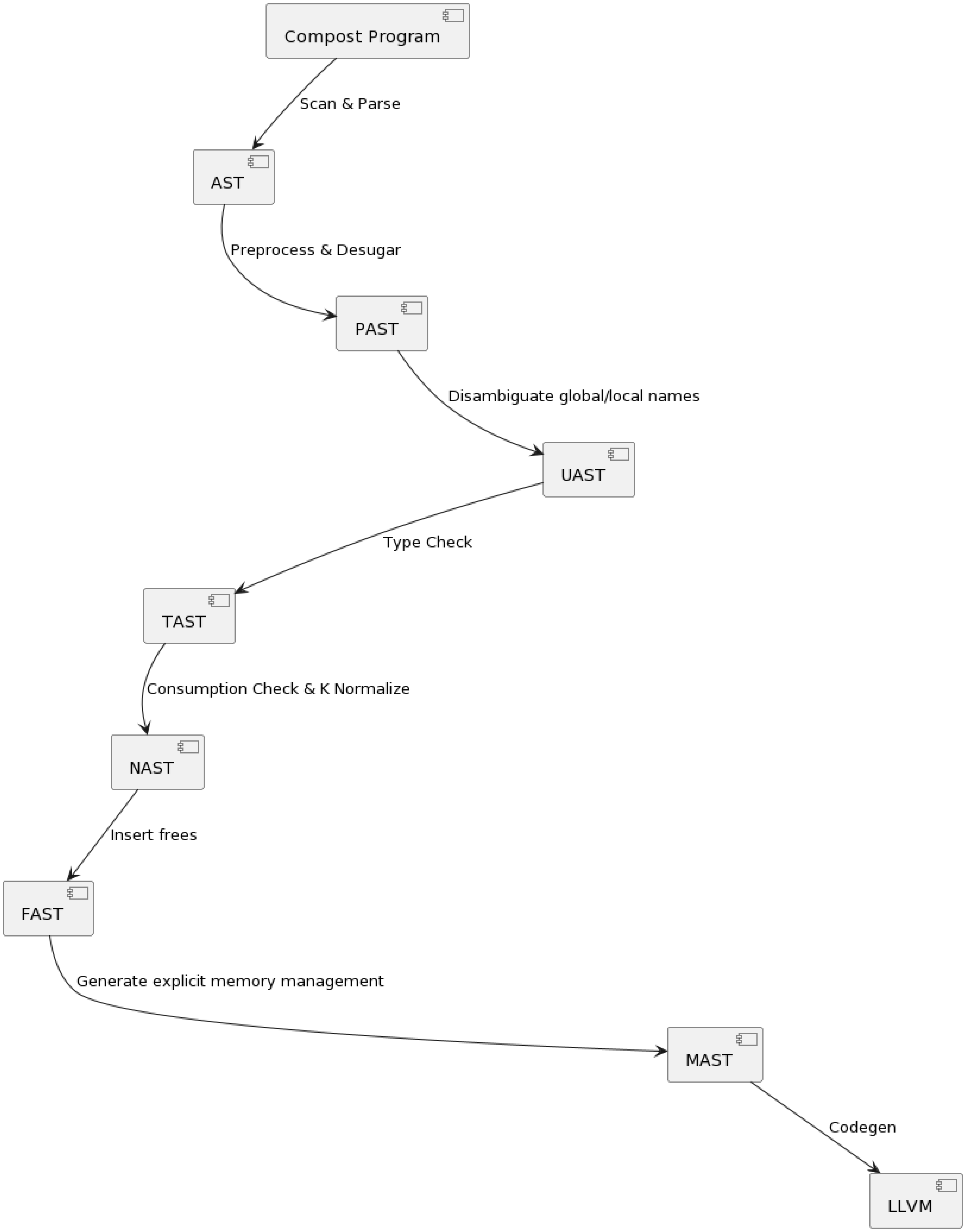
Give block diagram showing the major components of your compiler and the interfaces between them

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 → AST)	Randy Dang
Preprocessing & Desugaring (AST → PAST)	Jackson Warhover
Disambiguation (PAST → UAST)	Jasper Geer
Type Checking (UAST → TAST)	Roger Burtonpatel
Consumption Checking & K Normalization (TAST → NAST)	Jasper Geer
Insertion of Frees (NAST → FAST)	Jasper Geer
Generation of Explicit Memory Management Functionality (FAST → MAST)	Randy Dang
Codegen (MAST → 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 Miscellaneous Features

Tailcall optimization

Higher order functions

Include Global Object Value (.gov) Files

6 Test Plan

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

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

7.3 Jasper Geer

7.4 Jackson Warhover

8 Appendix

Attach a complete code listing of your translator with each module signed by its author(s)

Do not include any automatically generated files, only the sources.