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 run-time 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

A short explanation telling a novice how to use your language (consider this an informal version of a Language Reference Manual).

Explains how to use the compiler in its simplest form and run a compiled program.

Incrementally introduces how the language works through informal, well-documented code examples.

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).

```
(: 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 is called within `main`.

```
(: 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 calling ‘cons-int’ on an int
                               ;; and another int-list.
     [nil-int-list ()]))       ;; Variant constructor 2: create an empty
                               ;; int-list by calling ‘nil-int-list’ with
                               ;; no arguments
```

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’
```

2.4 How to Use Compiler

To use our compiler to compile Compost code, there should be a script called `compile-compost` in the top-level directory. Ensure that `cc` is symlinked to some version of `clang`, and simply execute that script as such:

```
./compile-compost 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 executable with:

```
./file
```

With the above example, the `compile-compost` script internally runs the following command:

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

3 Language Manual

Include your language reference manual. Make sure it's been updated if you've made *any* changes since the first LRM deliverable was turned in. I **will** use this to try to write my own programs in your language!

4 Project Plan

Identify process used for planning, specification, and development

Show your project timeline

Identify roles/responsibilities/contributions of each team member

Describe the software development environment used (tools and languages)

If possible, include a visualization of version control commits (but not a dump of a commit log)

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

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

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.