

Pure Garbage: The Compost Language Reference Manual

October 18, 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")]
```

1 Introduction

Compost is a statically-typed pure functional programming language with a uniqueness type system. That is, Compost guarantees that no two live references ever exist to the same heap object. As a result, the Compost compiler can perform **compile-time memory management**, inserting memory-freeing directives and guaranteeing memory safety for all Compost programs, bar runtime errors.

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

2 Notational Conventions

Code listings will appear in “verbatim” as follows:

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

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 brackets may be omitted or arbitrarily repeated.

3 Lexical Conventions

3.1 Whitespace

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

3.2 Comments

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

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.

3.4 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.

3.5 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. These 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'
```

3.6 Other Literals

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

unit-literal ::= `unit`

3.7 Reserved Words

The following tokens are considered reserved:

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

4 Values

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

4.1 Integers

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

4.2 Symbols

Symbol values are interned immutable strings of unicode characters.

4.3 Booleans

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

4.4 Unit

Unit values are the value *unit*.

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 $v_1 \dots v_n$ as $(d \ v_1 \dots \ v_n)$.

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

4.6 Functions

Functions in Compost are first-class objects. Function values are mappings from ordered sets of values, to values. That is, a function f , when applied to values $v_1 \dots v_n$, produces a value v_r . Like variant constructors, functions are monomorphic, so the types of $v_1 \dots v_n$ and v_r are fixed.

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, a ', bracket, or parenthesis, and is not a reserved word.

Names are bound to types, values, and variant constructors, and used to refer to them at various points in a program. Names are also used in the *use-declaration* syntactic form to refer to files.

6 Type Expressions

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

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.

6.2 Function Types

function-type ::= $(\rightarrow (\{ \text{type} \}) \text{ type})$

$(\rightarrow (\text{t}_1 \dots \text{t}_n) \text{ tr})$ is the type of function values which map ordered sets of values $v_1 \dots v_n$ of types $\text{t}_1 \dots \text{t}_n$ to values v_r of type tr .

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 ::= $(\text{datatype} \text{ name} (\{ \text{variant-constructor-definition} \}))$

variant-constructor-definition ::= $(\text{name} (\{ \text{type-expression} \}))$

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-expressions* $\tau_1 \dots \tau_n$ typing its arguments. Given this definition, a variant value $(c v_1 \dots v_n)$ of type τ_d may be introduced by applying function value *c* to $v_1 \dots v_n$, where the type of v_i is τ_i for all $i \in 1, 2, \dots, n$

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”.

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` declarations 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, `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 “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. 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.

7.1 Case Expressions

case-expression ::= $(\text{case } \textit{expr} (\{ \text{case-branch} \}))$

case-branch ::= $(\text{pattern } \textit{expr})$

pattern ::= $(\text{name } \{ \text{name} \mid _ \})$
 | $_$

Note that we refer to instances of $_$ in patterns as “wildcards”. Values of the form $(c v_1 \dots v_n)$ 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]
  )
)
```

7.1.1 Typing

We assert that the type of \mathbf{e} must be a datatype. Suppose that $\Gamma \vdash \mathbf{e} : \tau_d$. Each ci must be a variant constructor of τ_d . For all $i \in 1, 2, \dots, n$, let $\tau_{i1}, \tau_{i2}, \dots, \tau_{im}$ be the types of ci 's m arguments. We assert that the number of names and wildcards following ci must be precisely m , and that all names must be fresh. Let Γ_i be $\Gamma[v_{i1} \mapsto \tau_{i1}, \dots, v_{im} \mapsto \tau_{im}]$. Note that wildcards are not bound. We assert that $\Gamma_i \vdash \mathbf{e}_i : \tau_r$. The type of this expression in context Γ is τ_r .

7.1.2 Consumption

Names marked as consumed in \mathbf{e} are marked as consumed in all \mathbf{e}_i .

7.1.3 Evaluation

Suppose evaluation of \mathbf{e} in environment ρ yields a value $v = (c v_1 \dots v_m)$. If there exists some branch whose pattern is prefixed by c , it is evaluated in the environment ρ and its result is returned. Otherwise, the program halts with a runtime error.

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

7.2 If Expressions

if-expression $::= (\text{if } \mathbf{expr} \mathbf{expr} \mathbf{expr})$

Consider an if expression of the form:

$(\text{if } \mathbf{e}_1 \mathbf{e}_2 \mathbf{e}_3)$

7.2.1 Typing

We assert that the $\Gamma \vdash \mathbf{e}_1 : \text{bool}$. We further assert that $\Gamma \vdash \mathbf{e}_2 : \tau_r$ and $\Gamma \vdash \mathbf{e}_3 : \tau_r$. The type of this expression in context Γ is τ_r .

7.2.2 Consumption

Names marked as consumed in \mathbf{e}_1 are marked as consumed in \mathbf{e}_1 and \mathbf{e}_2 .

7.2.3 Evaluation

Suppose evaluation of \mathbf{e}_1 in environments ρ yields a boolean value v . If v is the value `true`, the expression \mathbf{e}_1 is evaluated in environment ρ and its result is returned. If v is the value `false`, the expression \mathbf{e}_2 is evaluated in environment ρ and its result is returned.

7.3 Begin Expressions

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

Consider a begin expression of the form:

(**begin** *e*1 ... *en*)

7.3.1 Typing

The type of this expression is the type of *en*.

7.3.2 Consumption

For all $i \in 1, 2, \dots, n - 1$, names marked as consumed in *ei* are marked as consumed in *ei + 1*.

7.3.3 Evaluation

Each *ei* is evaluated in environment ρ in order from $1 \dots n$. We return the result of evaluating *en* in environment ρ .

7.4 Apply Expressions

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

Consider an apply expression of the form:

(**e** *e*1 ... *en*)

7.4.1 Typing

We assert that $\Gamma \vdash e : (\rightarrow (t_1 \dots t_n) t_r)$. Each *ei* must be of type t_i for $i \in 1, 2, \dots, n$. The type of this expression is t_r .

7.4.2 Consumption

For all $i \in 1, 2, \dots, n - 1$, names marked as consumed in *ei* are marked as consumed in *ei + 1*.

7.4.3 Evaluation

Each *ei* is evaluated in environment ρ in order from $1 \dots n$. Let $v_1 \dots v_n$ be the values returned by evaluating each *ei*.

We return the result of applying v_f to arguments $v_1 \dots v_n$.

7.5 Let Expressions

let-expression ::= $(\text{let } (\{ \text{let-binding} \}) \text{ expr})$

let-binding ::= $(name \text{ expr})$

Consider a let expression of the form:

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

7.5.1 Typing

We assert that each x_i for $i \in 1, 2, \dots, n$ be fresh. Given that $\Gamma_k : xk : \tau_k$, for $k \in 1, 2, \dots, n - 1$, we say that $\Gamma_{k+1} = \Gamma_k[xk \mapsto \tau_k]$. As a base case, let $\Gamma_1 = \Gamma$. The type of this expression is the type of en in context Γ_n .

7.5.2 Consumption

For any $i \in 1, 2, \dots, n$ we mark any names consumed in ei as consumed in both e and all ek for $k > i$.

7.5.3 Evaluation

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

7.6 Name Expressions

name-expression ::= *name*

Consider a name expression of the form:

n

7.6.1 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 $\Gamma[n]$

7.6.2 Consumption

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

7.6.3 Evaluation

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

7.7 Dup Expressions

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

Consider a dup expression of the form:

(dup n)

7.7.1 Typing

We assert that n be bound in Γ and that $\Gamma[n]$ be a datatype. We further assert that n not be marked as consumed. The type of this expression is $\Gamma[n]$

7.7.2 Consumption

n is **not** marked as consumed.

7.7.3 Evaluation

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

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 section.

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.

8.1 Type Annotations

type-annotation ::= $(: \text{ name } \text{ type-expression})$

Type annotations constrain the type of globally bound names. Each bound name, whether bound by `val` or `define`, must have an associated type annotation. Consider a type annotation of the form:

$(: \text{ n } \text{ t})$

If `n` is bound by `define`, then we bind `n` to `t` in Γ_g at the first character of the file. If `n` is bound by `val`, then we bind `n` to `t` in Γ_g at the closing parenthesis of the `val` binding.

8.2 Val Bindings

val-binding ::= $(\text{val } \text{ name } \text{ exp})$

Consider a `val` binding of the form:

$(\text{val } \text{ x } \text{ e})$

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 $\Gamma_c[\text{x}] = \tau$, we assert that $\Gamma \vdash \text{e} = \tau$.

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 type-checked macros.

8.3 Function Definitions

function-definition ::= $(\text{define } \text{ name } (\{ \text{ name } \}) \text{ exp})$

Consider a function definition of the form:

$(\text{define } \text{ x } (\text{x1} \dots \text{xn}) \text{ e})$

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 $\Gamma[x] = (\rightarrow (\tau_1 \dots \tau_n) \tau_r)$. We assert that $\Gamma[\text{x1} \mapsto \tau_1, \dots, \text{xn} \mapsto \tau_n] \vdash \text{e} : \tau_r$.

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

8.4 Use Declarations

use-declaration ::= (**use** *filename*)

Use declarations are a thinly-veiled preprocessor directive which are replaced by the contents of the named file.

9 The Structure of Compost Programs

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

Compost programs consists 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 Compost program is compiled and executed, **main** is invoked and the program terminates when **main** is fully evaluated.

10 Initial Basis

Compost includes a minimal 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 a description will be provided for each such function.

10.1 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**.

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 div.

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

Two's complement modulus.

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

Two's complement negation.

10.3 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.

10.4 Logic

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

Logical NOT.

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

Logical AND.

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

Logical OR.

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

Logical XOR.

10.5 Printing

The following functions print representations of primitive values to stdout.

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

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

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

10.6 Input

```
(: in (-> () sym))
```

Returns a symbol containing a single character read from stdin. Note that all printable ASCII characters are initialized as symbols.

10.7 Lists and Helpers

10.7.1 Integers

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

Lisp-style lists of integers.

```
(: filter-list-int (-> ((-> (int) bool) list-int) list-int))
(define filter-list-int (f xxss)
  (case xxss
    (
      [(cons-int x xs)
       (if (f x)
           (cons-int x (filter-list-int xs))
           (filter-list-int f xs)
         )
      ]
      [(nil-int) (nil-int)]
    )
  )
)
```

Returns a list containing all elements of `xxss` satisfying predicate `f`.

```
(: exists-list-int (-> ((-> (int) bool) list-int) bool))
(define exists-list-int (f xxss)
  (case xxss
    (
      [(cons-int x xs)
       (if (f x)
           true
           (exists-list-int f xs)
         )
      ]
      [(nil-int) false]
    )
  )
)
```

Returns `true` if some element of `xxss` satisfies predicate `f`, `false` otherwise.

```
(: concat-list-int (-> (list-int list-int) list-int))
(define concat-list-int (xxs ys)
  (case xxs
    (
      [(cons-int x xs) (cons-list-int x (concat-list-int xs ys))]
      [(nil-int) ys]
    )
  )
)
```

Concatenates two lists of integers.

10.7.2 Symbols

```
(datatype string
  (
    [append (int string)]
    [empty ()]
  )
)
```

Lisp-style lists of symbols.

```
(: filter-string (-> ((-> (int) bool) string) string))
(define filter-string (f xxs)
  (case xxs
    (
      [(append x xs)
       (if (f x)
           (append x (filter-string xs))
           (filter-string f xs))
      )
    ]
    [(empty) (empty)]
  )
)
```

Returns a string containing all elements of `xxs` satisfying predicate `f`.

```
(: exists-string (-> ((-> (int) bool) string) bool))
(define exists-string (f xxs)
  (case xxs
    (
      [(append x xs)
       (if (f x)
           true
           (exists-string f xs))
      )
    ]
    [(empty) false]
  )
)
```

Returns `true` if some element of `xxs` satisfies predicate `f`, `false` otherwise.

```
(: concat-string (-> (string string) string))
(define concat-string (xxs ys)
  (case xxs
    (
      [(append x xs) (cons-string x (concat-string xs ys))]
      [(empty) ys]
    )
  )
)
```

Concatenates two strings.

10.7.3 Booleans

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

Lisp-style lists of booleans.

```
(: filter-list-bool (-> ((-> (bool) bool) list-bool) list-bool))
(define filter-list-bool (f xxs)
  (case xxs
    (
      [(cons-bool x xs)
       (if (f x)
           (cons-bool x (filter-list-bool xs))
           (filter-list-bool f xs))
      )
    ]
    [(nil-bool) (nil-bool)]
  )
)
```

Returns a list containing all elements of `xxs` satisfying predicate `f`.

```
(: exists-list-bool (-> ((-> (bool) bool) list-bool) bool))
(define exists-list-bool (f xxs)
  (case xxs
    (
      [(cons-bool x xs)
       (if (f x)
           true
           (exists-list-bool f xs))
      )
    ]
    [(nil-bool) false]
  )
)
```

Returns `true` if some element of `xxs` satisfies predicate `f`, `false` otherwise.

```
(: concat-list-bool (-> (list-bool list-bool) list-bool))
(define concat-list-bool (xxs ys)
  (case xxs
    (
      [(cons-bool x xs) (cons-list-bool x (concat-list-bool xs ys))])
      [(nil-bool) ys]
    )
  )
)
```

Concatenates two lists of booleans.