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 automatic compile-time memory management enabled by its uniqueness type system.

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* }

Note that terms enclosed in brackets, as seen in the second line of our example grammar, may be omitted or repeated.

3 Lexical Conventions

4 Values

5 Types

5.1 Type Expressions

Syntactic forms in the *type* category may appear in type annotations (see section 4.2.2) or to type datatype fields (see section 4.2.3).

type ::= (`->` ({ *type* }) *type*)
| `int`
| `bool`
| `sym`
| *name*

Compost provides three primitive types. These are `int`: 32-bit signed integers, `bool`: boolean values `true` or `false`, and `sym`: interned immutable strings. In addition, there is a single type constructor, the `->` (arrow) type, which is used to construct the types of functions. For example, the type of the `+` function, which returns the sum of two integers, would be:

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

Within the pair of parentheses following the arrow are the types of the function's arguments, and the type following the closing parenthesis is the function's return type.

Note that a type may be a name other than those of the primtive types. Here the programmer can name a datatype, declared by the `datatype` form discussed in section 4.2.

6 Expressions

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.

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

The grammar for expressions is somewhat more complex and requires some explanation, so we have divided it among our subsections. The grammar above contains only those top-level rules.

6.1 Literals

The simplest expressions are literals, which introduce values of our primitive types.

literal ::= *int-lit*
| *sym-lit*
| *bool-lit*
| `unit`

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

bool-lit ::= `true` | `false`

sym-lit ::= `'{` *sym-char* `}'`

sym-char ::= any unicode code point other than `'` unless escaped with a \.

name ::= any token that is not an *int-lit*, does not contain a `'` or bracket, and is not one of the reserved words shown in typewriter font

Note the definitions of *sym-char* and *name*. Unlike some common languages, there are very few restrictions on what constitutes a valid name. For example, `.`, `1+1`, and `$#@!` are all valid names for functions, variables, and value constructors. For characters found in symbols, we permit unicode code points along with the standard set of ASCII values.

6.2 Case Expressions

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

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

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

6.3 If Expressions

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

6.4 Begin Expressions

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

6.5 Apply Expressions

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

6.6 Let Expressions

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

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

6.7 Dup Expressions

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

7 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*

7.1 Global Bindings

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

The `val` form introduces an immutable globally scoped binding. Note that the bound name is in scope only in code located below the declaration. For example the programmer may declare a global constant:

```
(val triangle-sides 3)
```

Now, any successive code may use `triangle-sides` in place of the literal `3`.

7.2 Function Definitions

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

The `define` form defines a function in the global scope. Any function may be referenced from any location in the program. For example, the following introduces a function that returns the greater of its two arguments:

```
(define max (x y)
  (if (> x y)
      x
      y
  )
)
```

The name directly following the `define` keyword is bound to the function in the global scope, and the list of names within the parentheses are bound, within the function body, to its arguments.

Note that the order of function declarations does not matter. Thus, the programmer may write mutually recursive programs like so:

```
(: is-even (-> (int) bool))
(define is-even (x)
  (if (= x 0)
      true
      (not (is-odd (- x 1)))
```

```
  )
)

(: is-odd (-> (int) bool))
(define is-odd (x)
  (if (= x 1)
      true
      (is-even (- x 1))
  )
)
```

That is, despite `is-odd` being declared after `is-even`, `is-even` is in scope within the body of `is-odd`.

7.3 Type Annotations

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

Type annotations constrain the type of globally bound names. In Compost, such names appear in two forms: the names of functions and names bound globally by `val`. Take the definition of function `max` from the previous section. We left out one key point, that this definition cannot stand on its own. Each function definition must have a corresponding type annotation that gives its type. This annotation may appear before or after its corresponding definition, but it must exist. For example, the definition of `max` may be succeeded by the following annotation:

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

That is, `max` is a function that takes two integer arguments and returns an integer. Though the type of a function must be an "arrow" type, `->` may appear more than once in a function's type annotation. For example, consider the following:

```
(: add-or-sub (-> (bool) (-> (int int) int)))
(define add-or-sub (cond)
  (if cond
      +
      -
  )
)
```

Given that `+` and `-` are binary addition and subtraction on integers as defined in the initial basis, `add-or-sub` is a function that, given a boolean argument, returns a functions from pairs of integers to integers. Note that Compost has no support for closures, so we cannot express the full range of higher-order functions.

As mentioned, the other use of a type annotation is to type a `val`-bound name. As with function definitions, all `val` bindings must have an associated type annotation. However, the scoping rules here are more subtle. Although a `val` binding's type annotation may be located before or after the binding itself, the bound name is in scope only after the `val` binding itself. The location of the associated type annotation has no effect in this regard.

7.4 Datatype Definitions

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

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

The `datatype` form allows the programmer to introduce new types. The name of the new type appears directly following the `datatype` keyword, and this is followed by a list of variant constructor definitions. Each of these provides a name for the constructor, followed by a list of types for its arguments. Like functions, the placement of a datatype'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 a cons-cell containing an `int` and an `int-list`, or the empty list, `nil`".

Tags serve two purposes. For every tag there exists a function of the same name which constructs a value of the corresponding datatype. In addition, when top-level pattern matching on a value of a datatype, the pattern in each branch is prefixed by a different tag. We will discuss top-level pattern matches in more detail in section 4, but here is an example using our already defined `int-list` datatype:

```
(: car (-> (intlist) int))
(define car (xxs)
  (case xxs
    ((cons x xs) x)
    ((nil) (error 'tried to take car of nil'))
  )
)

(: main (-> () unit)
(define main ()
  (let
    (
      [xs (cons 1 nil)]
      [y (car xs)]
    )
    (print-int y)
  )
)
```

Here, `car` performs a top-level pattern match on its argument `xxs`. If `xxs` was constructed with a call to `cons`, then the first element of that `cons` is returned. Otherwise, if `xxs` was constructed with a call to `nil`, we err. `main` then uses `cons` to construct an `intlist` from `1` and `nil` and calls `car` on the result.

7.5 Use Declarations

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

8 Scoping Rules

9 Initial Basis