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

Briefly explain our language and the LRM.

2 Conventions

Code listings will appear 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 Syntax and Semantics

3.1 Types

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 3.2.

3.2 Definitions

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

def ::= (`val` *name exp*)
| (`define` *name* ({ *name* }) *exp*)
| (`datatype` *name* ({ *variant* }))
| (`:` *name type*)
| (`use` *filename*)

variant ::= (*name* ({ *type* }))

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 following code may reference `triangle-sides` in place of the literal `3`.

The `define` form introduces a function in the global scope. Note that the order of function declarations does not matter. 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 sequence of names within the parentheses are bound within the function body to its arguments.

3.3 Expressions

expr ::= *literal*
| *name*
| (`case` *expr* ({ *case-branch* }))
| (`if` *expr expr expr*)
| (`begin` { *expr* })
| (*expr* { *expr* })
| (`let` ({ (*name expr*) }) *expr*)
| (`dup` *name*)

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

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

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

4 Standard Library

Describe what we plan to include in a standard library (initial basis in our case?).