

CS 211, Winter 2004

Lab Assignment L4: Inheritance, Polymorphism, and Cheeziness

Assigned: Friday, Feb. 27, 11:00 AM, Due: Friday, Mar. 5, 11:00 AM

Brian M. Dennis, Instructor

Bin Lin, Tom Lechner, Rachel Goldsborough, Teaching Assistants

Aims and Goals

The purpose of this assignment is to introduce you to the use of inheritance and polymorphism. To do so, you will extend the `cheezo` interpreter discussed in class with some new datatypes and primitive functions.

After you complete this lab, you should be able to do the following:

- Define new derived classes, including inheriting from abstract base classes
- Define virtual functions
- Use the `dynamic_cast` operator
- Create new classes using composition and standard container libraries.
- Use standard container libraries like `vector` including iterating over these objects
- Use static members for classwide data

This lab mainly consists of extending the `cheezo` interpreter. The source code will be made available on the course Web site.

String Functions (10 Points)

Cheezo already has a string datatype, implemented with the `istring` class. However, there aren't any interesting functions at the Scheme level to operate on this datatype. This class has an instance of the standard C++ `string` class in it, which should make the string manipulations easy.

Add the following functions to the `cheezo` interpreter

- `string-length, s`, returns the length of string `s`

- `string-ref`, $s\ i$, returns a new string which has the one character at position i in s . Use zero based indexing and your function should do bounds checking
- `string-set!` $s\ i\ s2$, modifies s in place by inserting $s2$ at location i . Again do bounds checking. Also, make sure $s2$ is only of length 1.
- `substring` $s\ i\ j$, return a new string which is the substring of s from position i to j not inclusive. Make sure to check that i and j are within the bounds of s and that i is less than j .

Lists (20 points)

The cheezo version of Scheme is so cheezy that it doesn't even have lists! That's not very Schemish. Add a `cons` cell datatype to the cheezo compiler. You'll need to add a new class to the system. As part of your class, create a unique `null` object. Your class shouldn't need to use any of the standard library classes. Besides the `cons` cell datatype add the following functions

- `cons` $o1\ o2$, takes any two objects and returns a new cons cell.
- `car` c , returns the first element of the cons cell c
- `cdr` c , returns the second element of the cons cell
- `null`, returns the `null` object, we need this because our s-expr parser can't deal with syntax like `'()`
- `null?` o , returns 1 if o is the `null` object, 0 otherwise
- Also, cons cells should print as `(o1 . o2)` if the cdr of the cell is not null and `(o1)` if the cdr is null.

Vectors (20 Points)

Lists are nice, but sometimes you want vectors, which typically take up less space and are faster to access. Add a `vector` datatype to cheezo. The datatype will be similar to the `string` datatype except a vector can hold any type of object. The primitive functions to be added will be similar to the `string` object's functions.

- `vector` $o\ \dots$, takes a variable number of arguments and returns a new vector containing those args
- `vector-length`, v , returns the length of vector c
- `vector-ref`, $v\ i$, returns the object at index i in v . Be sure to check that i is within the bounds of v
- `vector-set!` $v\ i\ o$, modifies v in place by inserting o at location i . Again do bounds checking.
- Also add a special `print_form` method for your vector datatype

The Let Special Form (10 Points)

Currently in `cheezo` there's no way to bind local variables. Add a new special form `let` to the the interpreter.

- `(let (bindings ...) body_exprs ...)`
- `binding --> (sym . expr)`

The `let` special form traverses `bindings` and for each `sym` evaluates `expr` in the current environment. Then the current environment is extended so that `sym` is bound to the result of that evaluation. Evaluation of all of the binding `exprs` are evaluated before the current environment is extended.

Once the environment is extended, each of the `body_exprs` expressions is evaluated, in the new environment. The result of the entire `let` expression is the result of evaluating the last `body_expr`. Once the `let` is finished the environment is returned to its previous state.