MC28 Lab 7
Implementing a 3D-Modeling Language

Max Hailperin Karl Knight

Due: December 8, 1995

1 Introduction

The goal in this lab is to implement a small, special-purpose programming language designed for producing three-dimensional scenes for viewing. You will do this working in teams of two or three; since I have designed the implementation to be split among several files, and have specified the procedures to be provided by each file, it should be possible for the team members to work concurrently on the various modules. There are also a few modules I'll supply, because they seem uninteresting to the primary goals of the project.

In order to keep the operation of the implementation relatively easy to understand, I have decided that the program you write will not directly cause a graphics window to appear, but rather will write out a lower-level description of the scene to a file. That much can be done on either a NeXT or an SGI. You can then view the file on one of the SGIs using the ivview application. You'll be able to just double-click on it (on the SGI) to open it in ivview; Karl can demonstrate this. The way you'll run your language implementation on a particular program with output to a particular file is shown in the following simple example:

```
(run-to '((compose sphere
                   (cone where (point base-center is (0 -1 0))))
          where
          (material stuff is blue-plastic)
          (scalar radius is .5))
        "demo")
```

The considerably more complex example from the language-description handout is in the file `mannequins.scm`. This and all other files referred to in this lab handout are in the directory `~max/MC28`. Printouts of all the files except `mannequins.scm` are also attached at the end of the handout.

2 Pre-Lab

Before coming to the lab, you should read and understand the language description, and also the next section, which describe the modules that will be used to implement it. It would also be helpful if you formed a team and reached consensus on how the work would be divided up.

3 Module Specifications

The implementation of the modeling language can be broken down into the following modules, each in its own file. You should familiarize yourself with the interface specifications for all modules, since those that you aren't writing you'll be making use of. After a brief description of each module, there follows a subsection describing the procedures that each is to provide.

environments This is the implementation of environments, lifted straight out of the metacircular evaluator. I'm providing it.

structures This is the implementation of the various types of values: points, scalars, scenes, materials, deferred expressions, and primitives.

point-computations This is the implementation of necessary operations on points, such as adding them or finding the distance between them.

syntax This is the implementation of the predicates and selectors for the various syntactic structures, such as expressions.

evaluate This is the heart of the implementation, containing specialized evaluators for each syntactic type, e.g. point expressions.

lookup-typed-value This module supplies procedures for each value type that looks a name up in the environment and then ensures that the value retrieved is of the correct type. If the value retrieved turns out to be a deferred expression, which needs to be evaluated, or a primitive, which needs to be invoked, this module also takes care of that, after checking that the type of the expression or primitive is correct.

guts This module is primarily concerned with supplying the pre-defined names. It also defines the user-interface procedure `run-to`, and provides the composition operation for combining scenes. I'm supplying this module.

output This module, which I'm providing, takes care of the actual outputting of low-level graphics commands. It is used only by the guts module rather than having any direct interface to the parts of the program that you are writing, so I won't bother specifying its interface below. (However, the printout of it is attached, in case you're interested.)

3.1 Environments

This module is documented fully in section 4.1.3 of the textbook. It makes no use of procedures from the other modules. The procedures `lookup-variable-value` and `extend-environment` are defined by this module and used in other modules.

3.2 Structures

This module doesn't use anything defined in other modules. It provides representations for the types listed below:

Type	Components
point	x, y, z
scalar	value
scene	contents
material	ambient, diffuse, specular, emissive, shininess, transparency
deferred	type, expression
primitive	type, procedure

For each type, the constructor name starts with `make-`, for example, `make-point`. The constructor takes the arguments in the order listed in the components column above. Each type also has a predicate, the name of which is formed by adding a question mark to the end of the type name, e.g. `point?`. The selector names are formed from the type name and component name, e.g. `point-x`.

Although this module doesn't need to care what the individual components of the structures are, other modules must be in agreement. In particular, note that other existing modules assume the three components of a point are ordinary Scheme numbers, *not* scalars; in contrast, the components of a material are assumed to be points and scalars (the first four are points, the last two are scalars). The `deferred-type` and `primitive-type` components of those structures is assumed to be a symbol, namely one of `point`, `scalar`, `scene`, and `material`.

3.3 Point-computations

This module makes use of the point data type from the structures module. It provides three procedures for use elsewhere: `add-points`, `make-point-proportionally`, and `distance-between`. `Add-points` takes two points as arguments and returns a point formed by adding each coordinate. `Make-point-proportionally` takes a number and two points. It returns a point computed as described for point expressions of the form `(`⟨scalar expression⟩ `from` ⟨point expression⟩ `to` ⟨point expression⟩`)`. `Distance-between` takes two points and returns a number, namely the distance between the two points.

3.4 Syntax

The syntax module uses no procedures from the other modules. It provides predicates and selectors for use by the evaluate module. There is one predicate and one set of selectors for each syntactic form in the language. Note that it is useful to have each predicate only check that the top-level syntax is correct, not that each component is in turn correctly formed. That way the error messages will be more narrowly focused. (For example, "`Not a point expression: 7`" instead of "`Not a scene expression: (sphere where (point center is (offset 7 from (0 0 0))))`".)

One idea to consider for how to implement the predicates: it may be easiest if you use a general purpose pattern-matching procedure, such as the `matches?` procedure from MC27.

The predicates and selectors are as follows:

```
immediate-definition?
immediate-definition-type
immediate-definition-name
immediate-definition-expression

deferred-definition?
deferred-definition-type
deferred-definition-name
deferred-definition-expression

name?

composite-scene?
composite-scene-scenes

where?
where-body
where-definitions

constructed-point?
constructed-point-x
constructed-point-y
constructed-point-z

absolutely-offset-point?
absolutely-offset-point-offset
absolutely-offset-point-base

distance-offset-point?
distance-offset-point-distance
distance-offset-point-direction
distance-offset-point-base

proportional-point?
proportional-point-proportion
proportional-point-start
proportional-point-end

constant?

distance?
distance-start
distance-end
```

```
material-specification?
material-specification-ambient
material-specification-diffuse
material-specification-specular
material-specification-emissive
material-specification-shininess
material-specification-transparency
```

3.5 Evaluate

This module makes use of procedures from every other module except output. It provides the following evaluation procedures. Each takes an environment as a second argument.

Procedure	**First argument type**	**Result type**
evaluate-scene	scene expression	scene
evaluate-point	point expression	point
evaluate-scalar	scalar expression	scalar
evaluate-material	material expression	material

Although these are the only evaluators used by the other modules, it will probably help you develop this module if you include the following additional evaluators:

Procedure	**First argument type**	**Result type**
evaluate-definition	definition	environment
evaluate-definitions	list of definitions	environment
evaluate-scenes	non-empty list of scene expressions	scene

3.6 Lookup-typed-value

This module uses `lookup-variable-value` from the environment module, the data-type predicates and a few selectors from the structures module, and the evaluators from the evaluate module. It supplies the procedures `lookup-point-value`, `lookup-scalar-value`, `lookup-scene-value`, and `lookup-material-value`. Each takes a symbol and an environment. If the symbol's value in the environment is of the correct type, it is returned. Otherwise, if it is a deferred expression of the correct type, it is evaluated using the appropriate evaluator and the result returned. Alternatively, if it is a primitive of the right type, the primitive procedure (i.e., the `primitive-procedure` component) is applied to the current environment to get the value. If none of these cases applies, an error is signaled. [As further clarification on primitives: their procedure component is a normal Scheme procedure that expects one argument; that argument should be the environment.]

3.7 Guts

This module uses `evaluate-scene` from the evaluate module and various procedures from the output module. It provides the user-interface procedure `run-to`, and provides the procedure `compose-scenes` for use in evaluate. `Run-to` takes a program and a filename and puts the program's output in the file. `Compose-scenes` takes two scenes as arguments and produces a scene that is the result of combining together the two argument scenes.

4 In Lab

Implement the five remaining modules as specified above. I would suggest initially dividing the modules up between the team members. However, be flexible and help one another out when some modules prove more difficult or time consuming than expected. (Or less!) You may wind up, for example, partitioning the evaluate module into sub-modules, in order to make most effective use of your team's resources. Keep in mind throughout that the goal is for the team as a whole to develop a working system, and that division of labor is only good so long as it serves that goal. In other words, don't be too eager to say "that's not my job." One very important thing to do is to test each module separately, rather than just the whole system together, so as to avoid finger-pointing and needlessly open-ended searches for the problem, which can be confusing.

Test your resulting language system using the supplied examples, as well as any of your own devising. You may want to show the instructor the results first-hand, since it will be hard to adequately document the three-dimensional results in a two-dimensional lab report. (Since this was first written, this became even more true, as SGI broke the printing feature of the viewer.)

If you can find time, I'd really like to see everyone try to add an extra language feature of their own. Some of these could be quite simple to implement yet handy. One semester one of the teams implemented a repetition feature and used it to make a really cool centipede. Other equally cool ideas are still waiting to be thought of.

5 Post-Lab

I would like a lab report from each student documenting the project from that student's perspective. It should report on the modules or portions thereof that the student wrote, and should also report on the integrating, testing, and debugging process as seen by that student. I would also like to hear about any language extensions, changes, or improvements that you designed and/or implemented.