SchMUSE 001

An Introductory Guide to SchMUSE 001

by
Natalya Cohen
with contributions from
Michael Blair
David LaMacchia
and A Cast of Thousands

1 Introduction

This is the basic guide to SchMUSE 001, the Scheme Multi-User Simulation Environment. The SchMUSE shares its basic ideas with conventional MUDs, MUCKs and MOOs, so if you have encountered any of these before, you have a pretty good idea of the kind of functionality we provide. There are some things about the SchMUSE, however, that distinguish it from the traditional MUD style systems. It is an object-oriented, class-based environment. We will talk a little about the object system below. The main thing to understand is that the SchMUSE deals with real objects, as opposed to plain text. As a consequence, it is a lot more powerful than the conventional MUDs, for it allows you to manipulate its parts, the objects which comprise it.

SchMUSE 001 simulates an adventure game. However, that is not the system's only application. It can just as easily be used to simulate real physical problems, such as signal-processing systems or electron interaction. As you familiarize yourself with the interface, keep in mind that it can be and will be expanded and used in many different ways. This guide focuses on the basic commands you will need to know in order to get started. There is much more to learn about the SchMUSE, however, than the information you will find on these pages.

In order for you to understand the basics, we will walk you through the SchMUSE world, from the moment you register on it, to the moment you log out. (In actuality, this may take an arbitrary amount of time; there is much more to do here than just trying out the basic commands you will find in the script, but we will leave it to you to go back and try bigger and better things.) We suggest that you follow the scenario both on paper and on your screen, for we'll be referring to some things you see on the screen that won't make much sense otherwise.

2 The Beginning

2.1 Prompts and Conventions

Let's start at the beginning. What is the first thing you see when you establish a connection to the SchMUSE?[1] Well, frankly, you see a friendly welcome message. This is an acknowledgment of the established connection. The only reasonable response to it is to type `(hello)`. Already, you are faced with one of the conventions followed throughout, and that is parentheses. This aspect of the MUSE should come as no surprise to any Scheme hacker. Pretty much every command you issue is enclosed in parens. They all look like Scheme expressions. This is no coincidence since they really are just Scheme-syntaxed expressions that the SchMUSE server knows how to interpret.

At any rate, typing `(hello)` produces several things on your screen. Since you will see them pretty regularly, it is worth it to understand their meaning. First, there is something that looks like `(; Processing: (hello))`. This is an acknowledgement by the server that your command is being processed. You will see this acknowledgement every time you issue a command, unless the server is down (which is pretty bad news, since you won't be able to do much else). Next, you see another friendly message, to be discussed below. In general, this will be any kind of visual output, like text describing some command in case of a help message or a displayed message in case of an utterance (we will encounter these later). Immediately following the displayed message is a line echoing the earlier processing acknowledgement, which looks like `(; Value: <fnord>)`. In this particular case, the value is just a symbol. In general, it can be something a little more useful, like an object ticket (again, we will encounter situations like that later), which can be used to get a handle on an object. Next, you see the `time` it took the server to process your message. Simple commands like `(hello)` take minimal time to process; registering, on the other hand, takes much longer since it requires initializing your account on the database.

At last, you see the SchMUSE 001 prompt (a pretty spacious one, we admit). The version number changes (rather rarely) as the SchMUSE gets updated. The last dump information is rather useless to you as a user. All you might want to know is that every once in a while (how often depends on the system's load, number of users logged in, etc.), the world gets dumped to a file from which it can be restored later—we don't want you to lose your state in case the server

[1] We will discuss later (Appendix A) what you do in your Scheme to establish a connection and send queries/responses to the SchMUSE server. For now, just relax and concentrate on the Big Picture.

crashes for some mysterious reason. `Clock/Date`, as you might have guessed, tells you the current date and time. `Up since:` shows you how long the server has been up and running.

But going back to our friendly message, which gives you a short list of initial commands. If you are not yet registered on the system, you really have two choices at this point: you can either register or type `(help)`. Let's be cautious and go for the second choice.

2.2 Using On-Line Help

The help message (or menu) is pretty self-explanatory. All the help info is subdivided into sections which can be accessed by typing `(help <section-name>)`. If you type `(help alias)`, for example, you will see a typical help message. A short description of the topic is at the top. Following it are the translations of the variable names mentioned in the available commands—in this case, <alias-name> and <object-ref> (enclosing these in angle brackets is another convention we follow throughout, to make it easy for you to spot the variables in all the provided commands). Lastly, there are the commands themselves, grouped in an obvious way.

The on-line help gets updated as the commands change. It is accurate and informative. We recommend it as a good reference, although you will probably want to familiarize yourself with the system enough so that you know what kind of help to look for before you actually use it.

One special help command is `(help summary)`. It is a listing of all the commands in alphabetical order, and is there to make it easy for you to find the command you are seeking without having to go through all the help messages.

Another interesting help command is `(help system)`, and this is what we should talk about next.

2.3 System Status Commands

Status commands are designed to let you check the current system status. These are the only commands (excluding hello and help commands, as well as register and login) you can issue without logging in or even registering. You can try `(status users)`, for example, to find out how many users are currently connected to the SchMUSE (it may be that you are the only one and logging in at this time would be kind of useless).

Other commands, like `(who)` and `(all-users)`, mentioned in the `(help system)` menu, are probably of more interest to you. We will take a closer look at them in just a moment.

At this point, we have pretty much exhausted the commands available to you when you are not logged in. It's time to register.

3 Registration

So, let us initialize a Jack Random account. We do that by typing in something like:

```
(register jack "Jack Random" "foobar")
```

`(Help register)` will tell you that <username> (jack, in this case) has to be a symbol, whereas <pretty-name> and <password> are strings—i.e., they are enclosed in double quotes. In case you are wondering why you need two names, here is the rationale. You want to be able to simply type `jack` any time you want to get a handle on an object ticket to which the name refers. When you look around or converse with jack, however, his pretty name is displayed. This pretty name can be arbitrarily complex and/or amusing. Because it can be long and because it is a string (strings are used because they are case sensitive), a pretty name is not something you would want to type in every time you try to access the object. You can always change your pretty name or the pretty name of anything you have created (see the `change-name` command in `(help names)` menu). Similarly, you can always change your password with the `change-password` command. We will experiment with these later. First, let's talk about what happens when you register.

Essentially, issuing a register command creates you, a user, in the SchMUSE world. But what does being a user really mean?

3.1 Users and Their Characters

The thing to understand is this: a user exists mainly through his or her creations. Every user has a number of characters (people), whose role he/she can assume. You will see below how to create different characters, switch between your characters, and so on. For now, all you need to know is that, when you register, a character with the same short name (short names are symbols, just like usernames—in fact, username is a kind of short name) as your username and the same pretty name as yours is created for you. You see this character waking up as you register (in fact, you will see the Supreme Being waking up your creations every time you log in and the Grim Reaper putting them to sleep as you log out—we will talk about what it means for objects to go to sleep later on). You may choose to make your initial character your "main" character, since it has the same name as yours, but that is not in the least obligatory.

From the moment your registration is completed (note that registering automatically logs you in), you can issue any command you want, but most of them will be actually issued by your character. For example, when you look around in a room, it is your character who is doing it.

When you say something, it is your character who utters the words and hears him/herself. In fact, users don't even have a place or a way of communicating, except through their characters. Essentially, they exist for the sole purpose of creating their characters and other objects, such as places and things.

3.2 Basic User Commands

There are some commands you really do issue as a user, though. `Create` commands are like that, and we will deal with those later. There is also the issue of money, which we have already encountered in the process of registering.

3.2.1 Simple Money Matters

That is what "allowance" is all about. The idea is that every user is given a certain amount of money as he/she registers on the SchMUSE (it happens to be $205, or (* 5 41)—fourty acres and a mule for each digit on a human left hand[1]). Every time you create an object, any object, you spend some money on it. This is a safety measure against overloading the server. We will discuss the money policy more in depth as we move on to creating objects. This paragraph is here just to make sure you don't get overwhelmed by random Mr. Potatoeheads [sic] giving you (a Random user, mind you) random amounts of money for no apparent reason.

3.2.2 "Who" and "All-users"; Login Times

Now that you are logged in, you can try executing the rest of the commands in the `(help system)` menu. `(All-users)` will show you all the users registered on the server; `(who)` will only show the ones currently logged in. `(Who)` is especially useful if you are a friendly kind of person and seek communication with your fellow SchMUSEers. Both commands show each user's (including yourself) name, login time (if he or she is logged in) and his/her idleness (or the date and time the user was last seen SchMUSEing).

---

[1] All right, so this is a little obscure. Basically, that's what the Emancipation Proclamation promised every freed slave—a promise never fulfilled. We had to multiply it by 5, because... well, because we wanted to be generous and give you a reasonable amount of money. Now, is that better?

If you only want to see your own login time, use the command `(login-time)`. This gives you the date and the time your `login` request was processed.

3.2.3 "All-places"

The server also keeps track of all the places being created on the SchMUSE. If you execute the command `(all-places)`, you will see the names (both short and pretty) and unique IDs (see below for information on unique IDs and object references in general) of all of these places. This can be a helpful reference in case you decide to really explore the SchMUSE world and visit every single place in it.

3.3 Unique IDs

Another piece of information you are given when registering is your unique ID. You see something like this:

```
; Your unique ID is: 19920804160637076
```

This brings us to the issue of objects, object tickets, and object references (unique ID is just one of them). We will discuss this next, as we look around the room and notice the interesting things there.

4 Objects and Object Tickets

4.1 The Look-around Command

Your initial character enters the game at The Lobby, which is one of the places in the SchMUSE world. As in real life, the first thing you will probably want to do as you find yourself in a new place is look around. You can do that by issuing a (are you ready?) `(look-around)` command. This is a kind of thing you get back when you look around:

```
;---------------------------------------------------------------------------
; [jack (a.k.a. Jack Random) hears]
;
; At The Lobby : Jack Random thinks --
;  `` Hmmm...
;  the birthplace of all users
;  The Lobby contains:
;
; 0. zig [19920727225339280]: Zigamorph
; 1. rock [19920727225348212]: A very small rock (that floats)
; 2. dml [19920727230205668]: Daveman
; 3. the-forger [1992072520354524]: The Artful Dodger
; 4. the-terminator [1992072520345659]: The Grim Reaper
; 5. the-creator [1992072520342645]: The Supreme Being ''
;---------------------------------------------------------------------------
```

As you see, this shows you all the visible objects in the room—i.e., people and things. Objects are indexed (for example, zig's index is 0, whereas rock's index is 1[1])—you will see why when we talk about object references. The last three are server provided characters (see the section on server-defined objects in Appendix C). They make important announcements (for example, The Supreme Being notifies all the current logged in users of the new user registering on the SchMUSE), page you when needed (you might have noticed that The Supreme Being was the one who gave you the information about the current registered users when you issued the `(all-users)` command, for instance), and so on and so forth.

[1] Zero indexing, by the way, is another one of the followed conventions. You probably won't wonder why if you have any experience at all with computer science.

Another thing you might have noticed is the "description" of the place, in this case, '`the birthplace of all users`', which also shows up every time you look around. We will talk later about how you can give any object you create a description.

Finally, in addition to seeing this beautifully formatted glossy picture of the things in the room, you got back a list of the stuff in the room (recall that the SchMUSE returns a value for every command you issue). This list is not a pretty picture but a list of actual "tickets" that you can use to frob and poke objects on the server. Not surprisingly, we call these object tickets "object tickets." Let's look more closely at what these are.

4.2 Objects and Object Tickets

To be an object is to "have state." In other words, an object's behavior is influenced by its history. Your character is an object, in so far as it keeps its state as it accumulates possessions, for example. If jack takes `A very small rock (that floats)` from the lobby and then logs out, he will still have the rock in his possession when he logs back in. Every object type has specific "properties" which describe its state and specific "messages" it accepts, and we will talk of those when we get to locks.

SchMUSE is comprised of objects. People you create, places they go to, things they possess are all objects. This creates a need for a unifying representation of objects—what we call object tickets. You can see what an object ticket looks like by typing in any of the short names (for example) of the people or things you see in the lobby. Here is what you get when you type `zig`:

```
(; Value: (*the-object-ticket-tag* (unique-id . 19920727225339280)
(machine-location *here-i-am*) (type . person) (prop-list)))
```

Every object ticket is unique (that is why it has a unique ID, you might suspect). Besides the unique ID, an important thing to notice about an object ticket is its "object type" slot. `Zig`, for example, is a '`person`'. Rock's object ticket would indicate that it is a '`thing`' and lobby's, that it is a '`place`'. For now, these are the types you should know about. We will deal with other object types, such as '`locks`' and '`portals`', a little bit later.

There are a lot of ways to get a handle on an object ticket. Almost every interesting command you send to the server will return an object ticket or a list of object tickets. Normally you would just name these tickets and lists of tickets in your Scheme by using DEFINE or LET and so on. For example, you might type:

```
(let ((stuff (schmuse '(look-around)))
  (schmuse (list 'NAME (car stuff))))
```

where `(schmuse <schmuse-command>)` sends the indicated command to the SchMUSE server. Notice how we composed a new SchMUSE command from components of the result of a previous SchMUSE command that we issued. This is the standard idiom for interacting with the server. It is presented in more detail in Appendix A. For now, the only important thing to know is that you normally will not need (or want) any other means of getting an object ticket from the SchMUSE server aside from issuing a command to the server.

Unfortunately, if we restricted the remainder of this document to examples that rely only on commands to get a hold of object tickets, the presentation would get fairly tedious. Therefore, for the remainder of this document, most of our examples will resort to using an object's short name in place of a full object ticket. If you a Scheme client, however, keep in mind that, whenever we refer to typing something like `<SchMUSE-command>`, we really mean (schmuse '<SchMUSE-command>), as in the example above.

There are some other ways to tell the server to which object you are referring besides supplying its short name. Collectively, we call them "object references." Thus, both full object tickets and object short names are examples of object references. If you want to explore the many other fancy ways of referring to objects, consult Appendix C.

If you are wondering why we care about object tickets and object references, you are asking the right question. True, object tickets by themselves would be kind of useless. However, object references are passed as arguments to almost any command, however simple, you issue. And it is time for us to find out what some of these simple commands are.

5 Basic Commands

We have already encountered the `look-around` command. Now that you know what objects the lobby contains, you can get more information about them.

5.1 Names

To find out someone's or something's pretty name, you would type: `(name <object-ref>)`. For example, asking for `(name rock)` would give me back '`A very small rock (that floats).`' `(Nickname rock)` will just give me back '`rock`' (this seems kind of useless right now, but wait till we get to locks and key descriptors).

We mentioned before that you can change your pretty name if you so desire. Here is an example of how to use the `change-name` command:

```
(change-name jack "Jack Vanilla")
```

5.2 Descriptions

You already saw The Lobby's description, since it is printed out automatically when you look around. Every object has a description. You can see any object's description (i.e., look at it) by using the `describe` command, for example:

```
(describe zig)
```

This returns

```
(; Value: (a crazy hacker whose motto is: sleep is for the weak!))
```

If you look at yourself, however, you will see:

```
(; Value: (yet another random non-descript person))
```

You will probably want to change your description to something cool. You do that by saying, for example (note that any description is a list which can, but doesn't have to, be quoted),

```
(change-description jack '(all work and no play makes jack a dull boy))
```

Now, whenever anyone looks at you, they will see the new description.

5.3 Uttering Your First Words

By now, you are probably anxious to notify the world of your existence. There are several utterance commands; they share a common structure: the only requirement is that the first (and sometimes the only) argument given to any utterance command is a list.

If you want all the people in the room to hear you, use `say`.

```
;; Either of these is fine.  The second one is case sensitive and will
;;  look nicer.  However, you may find, once you start creating
;;  listeners (see the chapter on them), that the first version is
;;  much more useful, since you can look for the occurence of a
;;  specified symbol in a list of symbols but you cannot perform
;;  substring searches.  We will be using symbols just to be consistent.
(say '(hey hackers!))
(say '("Hey hackers!"))
```

This causes your message to be displayed on the screens of zig and dml, as well as yourself. Specifically, this is what you will see:

```
;-----------------------------------------------------------------------------
; [jack (a.k.a. Jack Random) hears]
;
; At The Lobby : Jack Random says --
;  `` hey hackers! ''
;-----------------------------------------------------------------------------
```

(What actually happens, when you do any kind of utterance, is that you send '`listen`' messages to a specified group of people—in this case, to everybody in the room. We will talk more about messages and the '`listen`' message in particular when we get to locks and listeners.) Zig or dml may choose to reply, in which case soon something interesting will pop up on your screen as well.

`Think` will only display the message on your screen, no one else's. `Announce` will display it on the screen of every logged in user:

```
(announce '(hello, world!))
```

You can `page` any single user as well, by specifying a character to page, for example:

```
(page '(what was that about sleep?) zig)
```

`Whisper` is like `page`, except you can only whisper to someone in the same room as you. Finally, `emote` is a more flexible version of `say`. If you issue the following emote request

```
(emote '(yells -- ``i am getting restless.'')),
```

all three characters in the room will hear this:

```
At the Lobby: Jack Vanilla yells -- ``i am getting restless.''
```

5.4 Taking Your First Steps

When you have been in the lobby enough that you have explored everything there and are going out of your mind trying to keep yourself occupied, you might want to simply leave the place. You can always find out what are the possible exits at your current place by typing `(exits)`. This is what we get when we issue an `exits` command at the lobby:

```
(; Value: (door-no-1 door-no-2 door-no-3))
```

Each returned symbol (such as `door-no-1`) is a direction of travel. You can always find out the exit's destination with the `destination` command,

```
(destination door-no-1),
```

which returns the object ticket for the place this exit leads to. You can then find out that place's name if you so desire.

One of the exits out of the lobby is hal-office (a.k.a. "Hal's office"), which is an exciting place indeed, and we decide to go there to check out the old hacker's dictionary which is sitting on one of the shelves. All you have to do is type `(go door-no-1)`, and you will magically move to Hal's office. The people at the Lobby are notified of your departure, and, as you move to a new place, you automatically look around and say hi to everybody in the room.

5.5 Issues of Ownership and Possessions

Hal's office turns out to contain a fascinating collection of objects. In particular, there are, among other things, an HP (the nickname of which is snake), a hacker's dictionary mentioned earier (dictionary), and even an empty bottle of Idaho dry (bottle). After checking out the descriptions of all of these, you decide that possessing a good old hacker's dictionary wouldn't be so bad. Any object of type '`thing`' is ownable, but some things are not owned by anyone. You wonder if somebody already owns the dictionary. In order to determine that, you type

```
(owner dictionary)
```

and get back an object ticket. Turns out Hal already owns that. So you go for the wine bottle instead, and that returns

```
(; Value: nobody)
```

which sounds pretty convincing. Wondering all the while what exactly you will do with an empty bottle of Idaho dry, you attempt to take it:

```
(take bottle)
```

and oh joy! you are now the proud owner of the bottle.

You can find out what your possessions are by typing `(possessions jack)`. Before you were the rightful owner of the bottle, your possessions were nil—e.g., the command returned `(; Value: ())`. Right now, however, it returns the list with one object ticket in it.

You can try taking other things the place has in it. If you get bored with one of them, you can just `drop` it right back—the `drop` command is remarkably similar to `take`. You do decide to keep the bottle, though—it is simply too intriguing a thing to leave it standing at Hal's desk.

5.6 Creator

Typing `(creator bottle)` will return the object ticket of the user who created the thing (or a person, or a place) of interest. You ask for that user's name and find out it is `zig`. You also determine that this command is pretty useful, for it can tell you much about why some things are where they are—especially if you know the user in person or you have encountered him or her on the SchMUSE before.

5.7 Object Type

As we have already seen, there are several types of objects on the SchMUSE. The basic ones are: places, things, people, portals, locks, and listeners. There may also be more "advanced" object types, such as trolls, that can do things people can't (trolls would be able to eat people, for example), but you shouldn't worry about these for now. You can always find out the object's type by typing

```
(object-type <object-ref>)
```

For example, `(object-type jack)` will return the symbol `person`, whereas `(object-type bottle)` will return `thing`.

5.8 Place

At this point, dml pages you from the lobby warning you that zig just found out you took the bottle of Idaho dry and is prepared to chase after you. You start getting worried and want to determine where exactly zig is. You can find out any thing's or person's place by issuing a `place` command, such as

```
(place zig)
```

Again, you get back an object ticket and from it can determine the place's name. It seems that zig changed his mind and is staying where he has been the entire time, back in the lobby. You get sick of worrying about other people stealing the things they have created back from you, however, and you want to learn to create your own things. Good timing, pal. This is your chance.

6 Creating Objects of Your Own

As mentioned earlier, every user is given a set amount of money to be spent on creating objects. You can type `(balance)` to determine how much money you have left. If you do that now, you will get back 205, which is exactly how much you were given by Mr. Potatoehead [sic]. It is now time to spend some of it. Don't be afraid to go wild—there are always ways to get more money, and we will talk about those later.

6.1 Places

You figure you want to start by creating your own place, where you can experiment with creating new things and people. The `create-place` command takes a short name (nickname) and a pretty name to be given to the place. You can be as inventive as you want with the pretty name, but you probably want the nickname to be short and easy to remember, since you will have to type it in for every command requiring this place reference. Here is how you would create MIT:

```
(create-place MIT "The Main Campus")
```

Now try typing `(balance)` again. It is only down to 199. Pretty cheap way to build a university, isn't it?

You will probably want to give your place a description, something like

```
(change-description MIT '(hell would be a better name))
```

Now you are all set to furnish it with some things and people.

6.2 Things

`Create-thing` takes another argument in addition to a short name and a pretty name, a place reference (this makes sense, since things are mobile and can move from place to place). Let us create a coffee mug (those are always good to have around):

```
(create-thing mug "A handy dandy coffee mug" MIT)
```

Now everybody walking into MIT will see your mug there, as they rightly should. If you don't trust us, go see for yourself. But first, let's connect MIT to one of the already existing rooms.

6.3 Portals

A portal is another type of object. (The terms '`portal`' and '`exit`' are used almost interchangeably—the only difference between them is that an exit is a portal associated with a specific direction.) Portals are, essentially, passageways into places. Every time you create a place, you have to create exits leading to and from that place if you expect to ever visit it (or let others explore it). `Create-exit` takes as arguments a direction (which is essentially the new portal's nickname), a pretty name for the portal, and a reference to the place to which you want the portal to lead. Here is how we would create a passageway to MIT from hal-office (if you want MIT to be connected to the lobby instead, you have to go there first, for the exit will only be created from the current place of your character to the place specified by you):

```
(create-exit out "Down the street" MIT)
```

Now try `(exits)` again. The one you have just created should be among them. You can also `(go out)` and you will find yourself at MIT. Sure enough, the coffee mug is in its place. There is nothing else there, so you might want to play around with creating more objects.

The direction you pick when creating exits can be any symbol (like `out`, `downstairs`, or `up-the-wall`). However, if you want your exits to show up on the SchMUSE world map—and yes, it will be provided some day—the directions have to be either valid campus directions, such as `east` or `southeast`, or `up`/`down`.

You will notice that, although you can get into MIT, you cannot get out unless you create an exit leading back. In other words, portals are one way gateways. You certainly don't have to create an exit leading right back to hal-office. Instead, we can create an exit to the lobby:

```
(create-exit home "Back to The Lobby" lobby)
```

Or then again, you didn't even have to do that. Creating an entrance to a room with no exits is a great way to trap people. Who knows, maybe somebody will wander in and like it there so much he won't want to leave—and then you can share your bottle of Idaho dry with him. Or something like that.

6.4 People

Finally, you can create objects of type '`person`'. This is much more exciting than any other kind of creating you will be doing, because any person you create is a potential character. We will discuss character commands in more detail in just a moment, in the chapter on characters. For now, let us create a hacker person:

```
(create-person hacker "Alyssa P. Hacker" MIT 42)
```

The last argument to `create-person` is the threshold. This has to do with the internal system '`clock`'. Every person created on the SchMUSE is put on the clock list. Initially, a person's restlessness is zero. With every clock tick, everybody on the clock list gets more restless—i.e., their restlessness gets incremented by one. When hacker's restlessness reaches her threshold (i.e., 42), she is forced to `act`—if the room has exits, one of them is randomly chosen and she is moved to a new place, at which point her restlessness is set back to 0. Thus, you can create people, leave them behind and move on to the different areas of the SchMUSE, and the people you created will be wandering through random rooms without your supervision. This allows for a kind of "robot" behavior of your inactive characters. If, for some reason (which, to us, appears to be absolutely, totally, and in every other way, inconceivable), this idea does not appeal to you, you can avoid the issue by making your threshold factors negative—this essentially sets your threshold to infinity, signaling your wish to never be influenced by the clock.

Threshold can be changed with a `change-threshold` command—i.e.,

```
(change-threshold hacker 24)
```

6.5 Locks and Listeners

Creating locks and listeners can be hairy and complicated, and we would rather leave this issue for later, since it is not essential to understanding the basic commands. See the chapters on locks and listeners for much more information.

7 Characters

As a user, you have the power to become any person you have created. You can get a list of object tickets representing all such people by typing `(characters)`. Right now, jack and hacker are the only two people on your character list.

Furthermore, you can type `(current-character)` to find out which of your characters you are at the moment (this also returns an object ticket). Your present `current-character` is `jack`, as you might have suspected. But not for long.

7.1 Changing characters

Being Jack Random (or Jack Vanilla, if you wish) may get kind of boring, so let us become Alyssa P. Hacker for a while and see what a hacker's life is all about. You do that by issuing a simple `change-character` command, giving it a person reference as an argument:

```
(change-character hacker)
```

As the above command is executed, you see two things happening: Jack Random goes to sleep and Alyssa P. Hacker wakes up. This brings us to the topic of sleeping and waking up, further discussed below.

7.2 Sleep

Axiom 1: `sleep is good.`

Axiom 2: `sleep is always an option.`

And as for the certain person's motto reading '`Sleep is for the weak!`', that may be regarded as one man's opinion and shouldn't be taken seriously.

Every character is created asleep. All being asleep really means is that the person won't be able to do certain things—for example, hear anything that is being said in the room. If you try to page a sleeping person, your message will bounce—you will get a message back denying you your page request. The sleeping person's creator will not see the messages sent to that person. This is nice, because you want to be able to concentrate on your current character without having to worry about your other characters being harassed, and because you usually don't care to hear everything said in the room where your character just happens to be asleep.

You can still issue most of the commands involving the sleeping person. We can ask for jack's name, place, creator, or possessions, for example, while he is asleep. We can still both see and change his description. In fact, let us make sure people know jack is asleep when they look at him:

```
(change-description jack '(a very tired SchMUSEer who has been diligently
                            exploring the SchMUSE world all day and fell
                            asleep in the process))
```

As you have witnessed, when you `change-character` from one character to another, the old character is put to sleep and the new character is awakened. Thus, by default, only one of your characters is awake (i.e., active) at any given time. If you would like for several of your characters to be awake at once use the command `awake-character`, in the following manner:

```
(awake-character jack)
```

Let's say you want to make jack a "robot," for example—if his threshold is low enough, he can run around from room to room, hearing people say all kinds of interesting things, which then will be displayed on your screen. You will probably find, when you do that, however, that (this being MIT, especially) there is way too much flamage going on. If so, you can easily reverse the `awake-character` command with the `sleep-character` command (original, isn't it?):

```
(sleep-character jack)
```

Now, once again, the world is relatively quiet and you can go off doing wonderful things as Alyssa P. Hacker.

8 "I'm flat broke."—Ways to Get More Money

As you get into creating objects, you may find your balance approaching zero faster than you expected (who said an adventure game couldn't be just like real life?). When you try to create an object and discover that you are flat broke (you will get one of those cute messages we are sure you came to love and appreciate by now), don't panic. There are two remedies to this situation. We will discuss them in turn.

8.1 Allowance Requests

Remember how, when you first registered, Mr. Potatoehead [sic] sought you out and gave you $205, out of the goodness of his heart? Well, he's got a lot more where that came from. The easiest thing to do when you run out of cash is to type `(allowance-request)`. This is almost guaranteed to work the first time you try it. Mr. Potatoehead [sic] will automagically find you and, as you acknowledge getting your allowance, you balance will again go up to $205. (Incidentally, you don't have to wait for your balance to be zero in order to request an allowance—if you are the kind of person who wants that money, and wants it now.)

There is a catch, though. We cannot allow you to create infinitely many objects and get away with it, for the server will easily get overloaded. Therefore, Mr. Potatoehead [sic], being the smart guy that he is, keeps a payroll with a record for every user, where he records the amount of money given to each one. If a certain threshold has been reached (this threshold depends on the system's load and things like that, and varies with time), you are toast. No allowance requests can do you any good—you are not getting any more money until you get rid of some of the objects you have created—i.e.., '`de-install`' them.

8.2 De-installing Objects

You can de-install, or eliminate from the world, any object you have created (but not any other). De-installing an object causes it to be removed from the database: it is removed from the place if it is mobile, it is removed from its creator's (i.e., the user's) `characters` list if it's a person, it is removed from its owner's possessions if it's a thing, the access to it is cut off if it's a place, and so on and so forth. More importantly, its object ticket gets "cancelled", so that you cannot get a handle on the de-installed object.

Let's imagine that you created a person, Ben Bitdiddle (ben for short) at MIT, with whom you suddenly got bored. You are running low on cash (your balance is $5, as usual), and you decide to de-install him. (By the way, you can only do that if you are someone else at the time—i.e., you cannot de-install your current character. Since you are currently acting as Alyssa P. Hacker, however, you are safe.) Here is the way to do it:

```
(de-install ben)
```

Now, if you try typing `ben` (recall that this used to return an object ticket for a person with nickname ben), you will get an error. The server does not seem to know of ben's existence (it claims that the symbol ben is an unbound variable). Ben is no longer on your character list, nor is he at MIT—you can't see him when you look around. He seems to have vanished. In other words, "He is dead, Jim."

If you look at your balance now, it has gone up to $11. This means, by the way, that you originally spent $6 in order to create ben—it is an interest-free enterprise! Pretty good for an old moldy object you weren't playing with any more anyway, eh?

9 Locks and Keys

We have mentioned locks earlier, when we talked about creating different kinds of objects. A lock is just another object type, like a place or a thing. Locks are used as a powerful safety mechanism, to control other users' access to your objects—in particular, any of the '`messages`' your objects accept. As a simple example, you might want to put a lock on the `possessions` message of your character jack, which would only allow certain people to see his possessions. You would specify the conditions other characters must satisfy in order to see jack's possessions in what we call a `<`key-descriptor`>` for your lock (the character might have to be called anything but `foo`, for example). This way, you can save yourself a lot of time and frustration by not having to deal with `foo`, a generally annoying person who constantly looks up your possessions and, finding some of them interesting, tries to take them away from you.

Don't worry if you have no clue as to what we are talking about at this point. Locks are a rather complicated subject, and we will go through the commands very carefully, explaining everything in gory detail. There are a lot of things to be cleared up, however, before we can show you the actual examples of the locks you can create—so bear with us.

9.1 What We Mean By "Messages"

`Possessions` is just one of the messages you can put a lock on. There are many more. Essentially, what we mean by a "message" is a command that can be applied to an object of a specific type. For example, among the messages any object will accept are name, nickname, describe, change-description, and creator. We have encountered all of these before. Usually, you won't run into a problem when you type something like `(describe zig)`. If zig's creator, however, decided to be totally sadistic and put a lock on zig's `describe` message, you would have to satisfy some condition in order to see zig's description.

There are messages that can only be sent to an object of a specific type, just like some commands can only be given certain object types as arguments. A thing's messages, for example, include `owner` and `place`; a person's messages, `place`, `possessions`, and `threshold`; a user's messages, `balance` and `characters`. These lists are in no way complete: for a complete list of all available messages for different types refer to Appendix B. You probably understand by now, though, that all those commands you can type in are messages sent to different objects, and thus, you can put a lock on any one of them.

9.2 Deadbolts and Combos

There are two uses for locks: as '`deadbolts`' and as '`combos`'; any number of each may be added to a message. A deadbolt lock functions as a logical AND, while a combo, or a combination lock, is a logical OR. Thus, a message can have a complicated mix of deadbolts and combos guarding it. Being able to put an arbitrary number of locks on the same message is a nice feature, because, as the time goes by, you may want to either relax (by adding more combos) or toughen (by adding more deadbolts) the requirements players must satisfy in order to ask that message of the object.

For example, you should be able to put several locks on the portal you created leading from the lobby to MIT, specifying the conditions players must satisfy in order to bypass the portal—you would be adding locks to the portal's `pass-through` message. If your portal has three deadbolts and two combos, it can be bypassed only if all three of the deadbolts and at least one of the combos are "satisfied."

9.3 First Shot At Creating Locks

Here is the `create-lock` command, looking as innocent as ever:

```
(create-lock <nickname> <pretty-name> <key-descriptor>)
```

Creating locks can be a rather complicated task, depending upon how tough you want your key descriptor to be. We will start with simple key descriptors and then move on to frisk descriptors, which is where all the evil lies.

After we create some sample locks, we will try adding them to some objects' messages.

9.4 Key Descriptors

Once again, a key descriptor is a set of rules that a lock uses to see if a player should be allowed access to a particular message of an object. Key descriptors range from basic keys you will probably use most of the time to frisk keys, which are described separately. Below is a full list of possible key descriptors.

`(trivial-key)`

the basic key; lets anyone through

`(sealed-key)`
: the facist key; lets no one through but the creator of the lock

```
(combinations <key-descriptor-1>
                <key-descriptor-2>
                ...
                <key-descriptor-N>)
```

adds a series of combo (OR) keys to a lock

```
(deadbolts    <key-descriptor-1>
                <key-descriptor-2>
                ...
                <key-descriptor-N>)
```

adds a series of deadbolt (AND) keys to a lock

`(not <key-descriptor>)`
: adds a logical NOT key to a lock

`(frisk-key <frisk-descriptor>)`
: adds a frisk key to a lock (we will discuss frisk descriptors separately)

You can see the key descriptor of any lock you've created by issuing the command

```
(key-descriptor <lock-ref>)
```

where <lock-ref> is a reference to an object of type lock.

9.5 Really Creating Some Simple Locks

Here are two basic locks we can create using the simplest key descriptors possible:

```
(create-lock trivial "Trivial Lock" (trivial-key))
(create-lock sealed  "Sealed Lock"  (sealed-key))
```

Now you can type `(key-descriptor trivial)` to see the trivial lock's descriptor. You will get back the symbol `true`. The sealed lock's descriptor, on the other hand, returns `false`. You will still be able to bypass `sealed`, however, since you are its creator.

Our trivial lock will have no effect on any message that is currently unlocked. What's nice, though, is that you can replace any existing locks with a new lock—so, if you decide to get rid of all your locks on some message, you can just replace them with a trivial lock.

The sealed lock is rather effective if you want absolutely everybody—except yourself—to be excluded from sending a specific message to your object. You probably don't realize this, but some sealed locks are already installed for you. Try doing this, for example:

```
(change-name zig "Zigafoo")
```

You will find that, since zig isn't someone you created, you can't change his name, and neither can anyone but his creator himself. The reason is that the `change-name` message has a sealed lock on it. So do `change-description`, `un-register`, and some other commands. As you find out more about which messages do and do not have sealed locks on them, you might choose to take off some of them and/or put some new ones on.

9.6 Frisk Descriptors and Creating Non-trivial Locks

A frisk key gives you control, as creator, over what a lock requires of the player who is anything to bypass it. It can be any combination of most Scheme expressions that will tell the lock the appropriate required conditions. For example, the lock could ask if the bypasser has an odd number of things with a nickname `pizza` in his or her inventory (by using `odd?` in a lambda expression).

The conditions you give to the frisk descriptor can be arbitrarily complex. They may use the `object-type` of the character being probed or of his possessions (for example, you might want all the trolls to bypass your lock, or all the characters that have a possession of type '`food`', if there were such a type). The frisk descriptor may also probe for a specific "property" of an object, such as its pretty-name, possessions, balance, or any other <property-name>. Here is the command you would use to get a value for some property of an object:

```
(property <property-name> <object-ref-of-appropriate-type>)
```

For example, `(property pretty-name jack)` will give you back the string "Jack Vanilla," and `(property owner dictionary)` will return the object ticket referring to Hal. Property names, just like messages, may only apply to an object of the specific type (you can only ask for the '`owner`' of a '`thing`'). For a complete list of property names, refer to Appendix B.

Lastly, the frisk descriptors make use of the two magic variables, namely, `(current-user)`, which specifies the user being probed and `(current-character)`, which refers to his or her current character.

If you are still not exactly sure as to what is going on, hold on. We will look at examples of a few frisk descriptors, and life will become much easier.

9.6.1 Example: No Nosers Allowed

Suppose you don't mind things being just the way they are, except for one minor detail: there is a certain person you dislike that keeps hanging around you sending you (i.e., hacker, your character) messages you don't want him to be able to access. You know that his nickname is `noser`, and want to create a lock that would keep him away from anything the lock is put on. Here is how the frisk descriptor for the lock would look:

```
(not (eq? (nickname (current-user))
          'noser))
```

The `nickname` command inside the descriptor returns a symbol, and you are testing if it is `eq?` to the symbol `'noser`. If so, the lock will refuse the user his request. Remember that `(current-user)` specifies the user being probed.

Note that you did not have to say `(property nickname (current-user))`, although you could have chosen to do it that way. The reason is that `nickname` is on the list of the provided commands. Other <property-name>s, such as somebody else's balance or characters, can only be accessed by using the `property` command.

Now, you can create a lock with the above frisk descriptor:

```
(create-lock noser-lock
             "No nosers allowed"
             (frisk-key (not (eq? (nickname (current-user)) 'noser))))
```

9.6.2 Example: Rich or Hacking

The following lock allows through only the users with balance greater than or equal to 200, or those whose current character has the word '`hacker`' in their description (these are usually mutually exclusive):

```
(create-lock rich-hacker-lock
             "Rich or hacking"
             (combinations (frisk-key (>= (property balance
                                                    (current-user))
                                          200))
                           (frisk-key (memq 'hacker
                                            (describe (current-character))))))
```

`Combinations` (see the section on key descriptors) is, really, just an alternative way of saying OR.

9.6.3 Example: All in Favor of Trolls...

Suppose we want to see if a character trying to get past the lock has not only an odd number of things named `pizza` in his or her possessions, but also a threshold of 37 (since everybody knows that is the most random number)—unless the character is of type troll, in which case no restrictions are imposed. This is the lock we would create:

```
(create-lock troll-lock
             "Trolls favored"
             (frisk-key
              (or (and (odd? (length
                              (filter
                                (lambda (x) (eq? (nickname x) 'pizza))
                                (possessions (current-character)))))
                       (= (property threshold (current-character)) 37))
                  (eq? (object-type (current-character)) 'troll))))
```

We used OR and AND here in order to show you all the different ways of creating keys and locks. Using combinations and deadbolts would have been equivalent, although probably more verbose and bothersome.

9.7 Adding and Removing Locks

Now that (hopefully) you are feeling more comfortable about creating locks with long and nifty key descriptors, it's time to learn how to add your locks to a message.

Locks can be added to an object's message and removed from it, but only by the object's creator.[1] New locks can be added to a message using one of the `add-` commands, while old locks can be removed using one of the `del-` commands. `Set-` commands override the current locks on a message by replacing all existing locks with a new lock.

Here are the `add-`, `del-`, and `set-` commands:

```
(add-deadbolt  <lock-ref> <message> <object-ref>)
(del-deadbolt  <lock-ref> <message> <object-ref>)
(set-deadbolts <lock-ref> <message> <object-ref>)

(add-combo     <lock-ref> <message> <object-ref>)
(del-combo     <lock-ref> <message> <object-ref>)
(set-combos    <lock-ref> <message> <object-ref>)
```

`<Lock-ref>` is a reference to the lock itself. `<Object-ref>` and `<message>` refer to the object you are playing with and the message you want to put your lock on. For instance, the `add-deadbolt` reads as: add the deadbolt lock to the message of the object given (in addition to any other locks already there).

Again, here are some examples of manipulating the locks.

9.7.1 Example: Noser-lock Enforcement

Now that we've got our nifty `noser-lock`, we can put it on any message we want. Let's start with our character's possessions:

```
(add-deadbolt noser-lock possessions jack)
```

---

[1] Well, actually, anyone who passes the lock on the object's `add-deadbolt` or whatever message can modify the object's locks, but by default these are all sealed locks.

This adds a deadbolt to the `possessions` message of jack. Now, whenever noser tries `(possessions jack)`, his request will be refused, for the lock won't let him through. (In this case, we chose to issue an `add-deadbolt` command, as opposed to `add-combo`, since you really don't like `noser` and will probably never want to allow him to bypass your lock. Adding a combo would give him a chance in case another combo were added later, the key descriptor for which he would satisfy.)

Noser, of course, gets annoyed at this restriction. He starts constantly paging you, whining and complaining about this and that. You get really tired of him and decide to put your `noser-lock` on your `listen` message:

```
(add-deadbolt noser-lock listen jack)
```

Now, anything noser utters will escape your ears—you will simply not see it on your screen. Aren't you glad you came up with the idea of creating this lock?

9.7.2 Example: Locking Portals

One of the more interesting things you can do with locks is locking the portals' `pass-through` messages. Let us put a couple of locks on the portal associated with the direction `out`, for instance—the portal leading from The Lobby to The Main Campus:

```
(add-combo rich-hacker-lock pass-through out)
(add-combo       troll-lock pass-through out)
```

Now you have to be either a rich user, a hacker character, a type troll, a person with an odd number of pizzas in your possession or a person with a threshold of 37 in order to pass through this portal. What an exquisite crowd this will bring to MIT!

9.8 Final Word on Locks and Keys

As with any gratuitously general feature, you should find yourself settling into a fairly simple use of locks with fairly simple keys. Also, you will probably only ever lock a few messages of any given object. Nevertheless, some amount of trial and error may be necessary to see that you haven't goofed. Your closest ally in debugging locks is first to look at the `key-descriptor` of any lock you created to see what Scheme expression the SchMUSE server would actually evaluate to produce true or false. Second, it is not a bad idea to have a friend cooperate with you in testing your locks to see that your key descriptors are working the way you think they are. Finally, a `test-lock` command (i.e., `(test-lock <lock-ref>)`) is provided which evaluate the lock's key descriptor with you as the current user and your current character as the current character, without letting you trivially pass the lock by virtue of being its creator. This at least lets you frob your own state to see if you can make yourself pass, then fail the lock in the appropriate ways.

10 Building “Machines” With Listeners

Having digested all the information we have given you on locks and key descriptors, you are probably wondering how you can put it to a good use. This chapter will make you aware of some reasons why you would need to put locks on messages—specifically, on the `listen` message of your characters. Also, it will present frisk descriptors in a new light, as we use them to create “listeners.”

10.1 What is a Listener?

So, let us talk about listeners. Listeners are very similar to locks in design. They are objects (yes, another kind of objects, just like persons, places, or locks) that you can put on any object, just like you can put locks on any object’s messages. Unlike locks, however, listeners are associated with an object itself, not one of its messages.

Listeners have two unique properties—“triggers” and “actions.” When a listener is put on an object, it frobs everything that object is made to listen to—i.e., the value of every `listen` message sent to the object (this is a list somebody types in as an argument whenever they issue one of the utterance commands), and also the object making the utterance. For example, if we are in the same room as dml when we issue the command

```
(say '(I hear the new issue of Sam & Max is coming out soon))
```

the `listen` message will be sent to dml (as well as everybody else in the room). The listener (or listeners, if there is more than one) on dml will then grab the list `'(I hear the new issue of Sam & Max is coming out soon)` and you, the utterer, and process both of these. If it “triggers” the listener, it will then activate its “actions” on the utterer.

Let us suppose, for example, that dml has a listener on him that looks for occurences of either of the symbols `sam` and `max`. When an utterer mentions either of these (thus, triggering the listener), the listener “takes action”—it asks dml to page the utterer in return. As a result, both of you get back this message:

```
;---------------------------------------------------------------------------
; [hacker (a.k.a. Alyssa P. Hacker) hears]
;
; At The Main Campus : Daveman pages Allyssa P. Hacker in The Main Campus --
;  ``Yo! sam and max win! ''
;---------------------------------------------------------------------------
```

Note that dml the user did not physically type in

```
(page '(sam and max win!) hacker)
```

—the paging happened automagically, with the help of the listener.

As you can see, listeners are a device used to produce "robot" behavior. By putting listeners on objects, you essentially create "machines" which will produce interesting behavior when the listeners are triggered. Thus, you can create objects that, even if not under your direct supervision, are able to interact with each other. These objects, probably contrary to your expectation, do not have to be people—any object in our world is able to `listen` to utterances.[1] You can create echo chambers that echo everything uttered in them; phones ringing in reaction to certain utterances or people; at last, Sam and Max running around, having their twisted conversations.

We will come back to the examples of some Sam & Max listeners you can create in a moment. Right now, let's look at the syntax of creating listeners, with their triggers and actions.

10.2 Triggers and Actions.

Here is the command:

```
(create-listener <nickname> <pretty-name> <triggers-list>
                                           <actions-list>)
```

You will notice that the listener expects a list of triggers and a list of actions. When the listener does its frobbing, any one of the satisfied triggers will activate everything on the actions list. Thus, a listener can be triggered by more than one specific kind of behavior, and the result can be a series of actions, not just one. We will see later how you can add and remove triggers and actions from a listener.

Triggers and actions are essentially our old friends frisk descriptors. Again, they contain Scheme expressions and `property` commands. The only real difference is that, instead of the magic variables `(current-user)` and `(current-character)`, there are now `(current-utterer)`,

[1] Well, actually, not yet. We are working on it. Right now, you should only put listeners on people, or you will lose.

which specifies the object being probed—i.e., the object doing the uttering, `(current-listenee)`, which refers to the object which is in possession of the listener, and `(current-list-to-listen-to)`—the list of stuff uttered. In our dml example, hacker is the `(current-utterer)`, dml is the `(current-listenee)`, and `'(I hear the new issue of Sam & Max is coming out soon)` is the `(current-list-to-listen-to)`.

After you've created a listener, you can look at its triggers and actions (this is not unlike looking at the lock's key descriptor) by issuing the following commands:

```
(triggers <listener-ref>)
(actions  <listener-ref>)
```

where <listener-ref> is, of course, the reference to a listener object.

10.3 Creating Some Sample Listeners

As before, examples are in order. So here goes, from everybody's favorite comic book.

```
(create-person sam "Sam" MIT  4)
(create-person max "Max" MIT -4)
```

10.3.1 Example: Where do you keep that thing?

Here is a random conversation Sam and Max might have on occasion. In fact, the question has come up more than once.

```
(create-listener tell-sam-off-listener "Stop asking me that!"
  ;; Triggered by someone's curiosity (this is probably Sam again, but
  ;;  nothing prevents others from asking Max that same question---and
  ;;  they should all know better)
  ((equal? (current-list-to-listen-to)
           '(Hey, where do you keep that futuristic weapon,
             Max?)))
  ;; Proper response
  ((say '(None of your damn business!))))
```

`Tell-sam-off-listener` only has one trigger and one action. We could certainly add more triggers to it—the response it gives is a nice one to any inquiry of a personal nature. In fact, we will do that when we discuss how triggers and actions can be added and deleted.

Of course, `tell-sam-off-listener` gets triggered if Sam (or someone else) says the right words. What's neat is that we can make him say those words by creating another listener, one that listens for Max shooting, for example:

```
(create-listener shooting-max-listener "Patience, little buddy..."
  ;; Here goes the one element list of triggers...
        ;; Is it just Max shooting, or is the world simply too violent
        ;;  nowadays?
  ((and (eq? (nickname (current-utterer)) 'max)
        (equal? (current-list-to-listen-to)
                '(Bang! Bang! Kerpow! Kak Kak Kak! At-at-at-at!
                  Kpshhh!))))
  ;; This is how the question comes up every time...
  ((say '(Hey, where do you keep that futuristic weapon,
          Max?))))
```

Now, whenever Max pulls out a gun and uses it (one implies the other, of course), Sam gets curious and gets the slap in the face he deserves. We can expand this conversation by creating more and more listeners, so that the two buddies simply never shut up. Now do you see why you might want to put a lock on your `listen` message?

10.3.2 Eternal Companions

Sam and Max are people, and they `act` from time to time—that is, they move from place to place with a frequency which depends on their threshold. We intentionally made Max's threshold negative (remember that the clock ignores people with negative thresholds), because we don't want our characters to move off in different directions (recall that, when you `act`, direction is randomly picked by the system). Instead, we want Max to follow Sam wherever he goes, so that the odd couple can have their weird conversations. We create a listener that detects Sam's intention to leave the place he is currently in and forces Max to go there too.

```
(create-listener sam-follower "No, I won't leave you alone, you nut!"
        ;; Is it Sam?
  ((and (eq? (nickname (current-utterer)) 'sam)
        ;; Is he about to escape? ("exit, stage" is in the list which
        ;;  everyone emotes as they are about to leave the room)
        (member "exit, stage" (current-list-to-listen-to))))
```

```
 ;; Get 'im, get 'im, boy!
  ;; First, get a handle on the direction of travel---that's the
  ;;  element in the (current-list-to-listen-to) that comes right after
  ;;  "exit, stage"
 ((let ((handle (member "exit, stage" (current-list-to-listen-to))))
    (let ((direction (cadr handle)))
      (go direction)))))
```

Now, we can make Sam say something interesting every time the couple moves to a new place, for example:

```
(create-listener max-arrival-listener "Here we are..."
  ;; Last thing Sam hears is Max being polite and saying "Hi" to
  ;;  everyone in the room
  ((and (eq? (nickname (current-utterer)) 'max)
        (member "Hi" (current-list-to-listen-to))))
   ;; So, what is this place?
  ((say (append '(Well, here we are... This is what they say about the
                  place:)
                (describe (place (current-listenee)))))
   ;; What to do, what to do...
   (say '(I am sure our reason for being here will soon become
          apparent))))
```

Now, when Max hears the last thing Sam says—namely, `'(I am sure our reason for being here will soon become apparent)`—he can see what about the place interests him and start a new conversation. This is demonstrated by the next example.

10.3.3 Example: Life in the kitchen

So, there is this place called kitchen. And whoever heard of the kitchen with no roaches? So let us make sure one shows up every time Sam and Max enter the kitchen and Sam utters his routine "here we are". Watch what will happen to the poor thing, too.

```
(create-listener max-roach-activater "What else is there to do in a kitchen?"
  ;; One trigger but three conditions to be met
        ;; Is it Sam?
  ((and (eq? (nickname (current-utterer)) 'sam)
        ;; Sam just said the magic words
        (equal? (current-list-to-listen-to)
                '(I am sure our reason for being here will soon become
                  apparent))
        ;; Is it the kitchen?
```

```
        (eq? (nickname (place (current-utterer)))
             'kitchen)))
   ;; First, create the poor thing
  ((create-person roach
                  "itty bitty plain old earth roach"
                  (place (current-listenee))
                  33)
   ;; See below for definition of roach-listener.  The roach will need
   ;;  this to be able to participate in the conversation.
   (add-listener roach-listener roach)
   ;; Inform Sam of its existence
   (say '(Look, Sam, a roach!))))


(create-listener sam-roach-reacter "Why, this is just what we needed!"
  ;; Triggered by Max's discovery
  ((equal? (current-list-to-listen-to)
           '(Look, Sam, a roach!)))
  ;; We are sharp
  ((say '(Well whaddya know?  It's an itty bitty plain old earth roach,
          and he's packin' a rod.))))

(create-listener max-roach-insulter "He looks like trouble to me..."
  ;; Triggered by Sam's observation
  ((and (eq? (nickname (current-utterer)) 'sam)
        (equal? (current-list-to-listen-to)
                '(Well whaddya know?  It's an itty bitty plain old earth
                  roach, and he's packin' a rod.))))
  ;; ...and even if he doesn't, we'll make him look like trouble
  ((say '(I don't like the look on his face, Sam.  It's cold.
          Calculated.  Almost inhuman.))
   ;; Insult the sucker
   (say '(You deadly little bug!))))

(create-listener roach-listener "Oh, yeah?"
  ;; Triggered by Max's insult
  ((equal? (current-list-to-listen-to)
           '(You deadly little bug!)))
  ;; He is a brave one
  ((say '(Nobody calls me a bug!))))

(create-listener max-roach-killer "What an arrogant little sucker!"
  ;; Triggered by the roach's brave yet silly comment
  ((eq? (nickname (current-utterer)) 'roach)
   (equal? (current-list-to-listen-to)
           '(Nobody calls me a bug!)))
   ;; Dirty old trick
  ((say '(Hey, take it easy, fella'.  Don't go getting yourself in any
          deeper.  Just hand over the pistol and nobody will get hurt,
          right Max?))
   ;; Wake up.  Time to die.
```

```
        (de-install roach)
        ;; Satisfaction
        (say '(FWAP!))))

     (create-listener sam-philosophizer "I should've gone to junior college!"
       ;; Triggered by Max's unmanly deed
       ((equal? (current-list-to-listen-to)
                '(FWAP!)))
       ;; Bummer, I didn't get to kill it... let's make the most of it
       ((say '(Sometime we should sit down and have a spiritual discussion
               about the sanctity of life.))))
```

Cute, isn't it?

10.4 Adding and Removing Triggers and Actions

You can change your listeners by adding and/or removing some of their triggers and/or actions. Here are the commands:

```
(add-trigger     <trigger> <listener-ref>)
(del-ith-trigger  <number> <listener-ref>)
(set-triggers    <trigger> <listener-ref>)

(add-action-to-front <action> <listener-ref>)
(add-action-to-back  <action> <listener-ref>)
(del-ith-action      <number> <listener-ref>)
(set-actions         <action> <listener-ref>)
```

Let's say you want to add a trigger to your `tell-sam-off-listener`. This is how:

```
(add-trigger (equal? (current-list-to-listen-to)
                     '(Hey, why is it that everybody makes such a big
                       deal out of Sam & Max, anyway?))
             tell-sam-off-listener)
```

Next time, they'll know better than to ask such a thing.

Adding actions is similar, except you can either add an action to the front (or top) of the list or to the back (bottom) of it. Sorry, you can't stick it in the middle.

`Del-` commands refer to the index of a trigger or an action. Thus, to remove a trigger, you would issue a `triggers` command, figure out the index of the trigger you want to delete (the list is zero indexed, as usual), and type in, say,

```
(del-ith-trigger 0 tell-sam-off-listener)
```

This would delete the trigger we've just added (all triggers are added to front, or top, of the trigger list).[2]

`Set-` commands override the current triggers or actions by replacing all existing ones with a new trigger or action—just like the `set-` commands for the locks.

10.5 Adding and Removing Listeners

At the risk of becoming monotonous, we present you with a list of the commands you would issue to add and remove listeners themselves:

```
(add-listener  <listener-ref> <object-ref>)
(del-listener  <listener-ref> <object-ref>)
(set-listeners <listener-ref> <object-ref>)
```

10.5.1 Example: Sam & Max Come to Life

Here is how we would install all the listeners created in our examples in the right places:

```
(add-listener tell-sam-off-listener max)
(add-listener shooting-max-listener sam)
(add-listener sam-follower          max)
(add-listener max-arrival-listener  sam)
(add-listener max-roach-activater   max)
(add-listener sam-roach-reacter     sam)
(add-listener max-roach-insulter    max)
(add-listener max-roach-killer      max)
(add-listener sam-philosophizer     sam)
```

[2] One thing you cannot do is delete the listener's `only` trigger or action.

The `roach-listener` is already being added to every new roach created. `Del-` and `set-` commands are quite similar.

Don't forget to wake up our two characters, so that they can run around and talk to each other (remember, listeners are not activated when the listenee is asleep):

```
(awake-character sam)
(awake-character max)
```

10.6 Final Word on Listeners

One thing you should know is that, when an object listens, only the first of the list of listeners which are triggered proceeds to take action. You may treat that as either a bug or a feature, but it does make things easier.

Another good idea is not to write a listener like this:

```
(create-listener infinite-loop "Off into never never land..."
  ((memq 'stupid (current-list-to-listen-to)))
  ;; Some people just like to listen to themselves...
  ((say '(stupid me))))
```

A listener like this will keep the server busy for a while—it will basically keep triggering itself till hell freezes over, or till the server crashes, whichever comes first. Trust us, we know from experience.

Finally, you can test the listeners you create, just as you could test locks. We supply the following two commands:

```
(triggers?   <listener-ref> <object-ref> <list-to-listen-to>)
(take-action <listener-ref> <object-ref> <list-to-listen-to>)
```

`<Object-ref>` here is the object that would be listening—in other words, `(current-listenee)`. The `(current-utterer)` is presumed to be your current character. `Triggers?` tests if the listener has been triggered. For example, if your current character is hacker, and you try

```
(triggers? max-arrival-listener sam '("Hi" Sam))
```

the value returned will be false, because `max-arrival-listener` requires that the `(current-utterer)` is Max. All the other conditions are satisfied—Sam is the `(current-listenee)` and `(current-list-to-listen-to)` contains the string "Hi". If you now change characters to Max and issue the exact same command, like this:

```
(change-character max)
(triggers? max-arrival-listener sam '("Hi" Sam)),
```

it will return true. If you successfully trigger the listener, you may then try issuing `take-action` with the same set of arguments, and, if your listener isn't faulty, the actions of the listener should get executed.

11 Logging Out and Logging Back In

Obviously, you can't do all the SchMUSEing in one day. So if, at any point in time, you feel like getting some sleep (remember—sleep is good) or stuffing a burger down your throat, don't hesitate to log out for a while. We will miss you terribly, but we'll understand.

11.1 Good Bye, Cruel World

Suppose our Jack Random user is still logged in to the SchMUSE. His current character is Alyssa P. Hacker, and her place is MIT. Jack decides that he's had enough for today and wants to log out.

When you type `(logout)`, a few obvious things happen. Your current character goes to sleep (recall the rationale for putting characters to sleep given in the chapter on characters). Any other characters you have that are awake do the same. Everything else you have created, such as places, portals, and things that don't belong to your characters, stay pretty much the way they were while you were logged in. Somebody might still be carrying around a book you had created and dropped, going into places created by you, seeking access through your portals. Sad as it is, the world stays pretty much the same even as you are gone. Although the Artful Dodger (he seems to be doing most of the announcing) does make sure everybody knows of your leave.

11.2 Back to SchMUSEing

As you might have guessed, the `login` command requires your username and password. Forgetting your password is probably not a good idea.

```
(login jack "foobar")
```

It also probably comes as no surprise that, as you log in, it is Alyssa P. Hacker who gets woken up, not Jack Random or any of your other characters, since she was your current character when you logged out. Furthermore, you still find yourself at MIT, the place where you fell asleep while logging out (a rare occasion, we agree). In fact, you pretty much find yourself and your characters unchanged. The places created by you might have lost or acquired objects, for there are always things and people being tossed around, but that only makes life more interesting.

12 But Wait! There's More: More User Commands

By now, we have mastered registering, logging out and logging back in. We have seen how to get a list of all the users logged in or registered on the SchMUSE and of all the places in the SchMUSE world. We have discussed login times. We have reviewed the money policy and the issues of creating and de-installing objects. However, there are still a few commands you can issue as a user that we need to discuss.

12.1 Changing Your Password

So far, we have looked at how to change your name, description, threshold, character. You can also at any time choose to change your password, in case you get bored with it or somebody else finds out what it is. This is very simple:

```
(change-password <old-password> <new-password>)
```

For example, to change your password to "Byte me," you would type

```
(change-password "foobar" "Byte me")
```

Now, when you log in, you should type in your new password.

12.2 Autologout

By default, if you are idle for two continuous hours, you will be logged out automatically. You can verify this by typing

```
(autologout-time)
```

This gives you back 120, the time after which you will be logged out (in minutes). If you would like to increase (or decrease) your autologout time, use the command

```
(change-autologout <number-of-minutes>)
```

For example, issuing `(change-autologout 240)` will change your autologout time to four hours, as you can see if you type `(autologout-time)` once again. There is a limit on how high the <number-of-minutes> can be—it depends on the server's load, and is no higher than 300 minutes, or 5 hours.

12.3 Changing Users

Another interesting command is `change-user`. It gives you the ability to log in as a different user without giving an explicit `logout` command. Here is a hypothetical situation in which the `change-user` command would prove to be useful.

Let's say your best friend is also a SchMUSE hacker. His username is dave, and his password is "Abrakadabra" (he shared it with you). He is stranded on a desert island somewhere in Idaho, where (sigh) he has no access to a computer. So he calls you at home and asks you to log in as him and see if any of his characters appear to be in any kind of trouble. Of course, you happen to also be logged in at the time, SchMUSEing away. Here is what you do:

```
(change-user dave "Abrakadabra")
```

Everything happens automagically: jack is logged out, dave is logged in, and you can now issue commands as if you were dave himself. After you have made sure dave is doing just fine, just type

```
(change-user jack "foobar")
```

—and you are back as jack.

12.4 Whopr Users

The `whopr` command lets you be a super-user. Super-users are users, just like the rest of us, except they can bypass locks on any object's message regardless of their key descriptors. Issuing `whopr` does not actually create a new, super-user, object; it simply sets a bit in your object's state that lets you be treated as a special kind of user. Here is the magic command:

```
(whopr <magic-password>)
```

If you don't know the password, you probably shouldn't. Sorry, pal, life's tough in the city. And in case you are wondering, whopr users are there mostly for maintenance purposes. Sometimes we just gotta clean up the mess regular users make.

`(Exit-whopr)` returns you back to a normal user mode.

13 The Last Resort: Un-registering

Not that you will ever want to issue this particular command. We thought we'd put its description here, though, just so you know exactly why you don't want to do this.

Issuing the `un-register` command permanently removes you, the user, from the database. Think of it as de-installing the user. If you were to execute

```
(un-register jack "foobar"),
```

for example, the object by the name jack would no longer be accessible to other users. If you tried to log in after un-registering, your attempt would fail. All the characters you have ever created but have not yet de-installed (such as Alyssa P. Hacker and Jack Random himself), as well as all their possesssions, would now be de-installed with you.

Not everything you have created gets de-installed as a result of an `un-register` command. Places, portals, and locks on the portals, for example, all stay around, so that other users still have access to them. Things you have created which now belong to others (i.e., other user's characters) don't get de-installed either. Things that are not owned by anyone, on the other hand, do get de-installed (there might be too many of those, and we can't leave around lots of junk nobody wants, so if you really don't want some of your things to get trashed, you might consider giving them to your friends).

You are probably convinced by now that un-registering is not something you would want to do often, unless you are one of those people who starts a new life every Monday morning. True, you can easily `un-register` and `register` back as a new user. Just beware that all of your state will be lost. Moreover, since the SchMUSE provides multiple characters, there is really no point in un-registering—if you really want change, you can simply change your character.

Appendix A The Scheme Client Interface to the SchMUSE Server

When you load the problem set code for the SchMUSE problem set, at some point you will load the '`server-startup`' file to turn your Scheme into a client of the "00" lab SchMUSE server.

From that point on, you can send commands to the SchMUSE server and get results back by using the Scheme procedure `(schmuse <schmuse-command>)`. Since this is a procedure (and not a special form) it will evaluate the <schmuse-command> argument before passing it on to the SchMUSE server. This is an important feature since it allows you to compose new commands based on results computed by previous server commands. Here are a few interesting examples:

```
(schmuse '(login jack "foobar"))   ; First you must log in... assume
                                   ;  we already registered yesterday.

(let ((victims (schmuse '(who))))
  (for-each (lambda (victim)
              (schmuse (list 'page "I want a cookie, now!" victim)))
            victims))

;; We will annoy everyone described as a Hal (or whose description
;; mentions hal) every hour or so, so let's DEFINE them to keep a handle
;; on them...  of course, anyone who later tries to join the club (by
;; changing their description to mention hal) will not be included
;; unless we re-evaluate this define. Oh well.

(define hals (filter (lambda (luser)
                       (let ((descr
                              (schmuse (list 'describe luser))))
                         (memq 'hal descr)))
                     (schmuse '(all-users))))

;; For starters, let's bring them all together...
(for-each (lambda (hal)
            (schmuse (list 'page
                           "; Meet me in the Lobby for a meeting of the
Hal's at MIT [a.k.a. HAMIT]"
                           hal)))
          hals)

;; This frisk descriptor might come in handy for a lot of different
;; locks, so we define it separately...
(define hamit-member-frisk-descriptor
  '(memq 'hal (describe (current-user))))


;; the HAMIT office is for Members Only!!
(define hamit-lock
```

```
(schmuse (list 'create-lock 'hamit-lock
                            "HAMIT office lock"
                            (list 'frisk-key
                                  hamit-member-frisk-descriptor)))
```

You get the idea...

Appendix B Comprehensive List of Server Object Types and Their Messages

This is the promised complete list of all the messages different object types will accept. Some of these are what we call property names. These refer to the properties of an object—its name, description, owner, restlessness, possessions—everything that is a part of an object's `state`, everything that can be changed about an object. (Not all messages are property names, because asking an object to say or listen to something, for example, does not change the object's state.) All the property names are printed in **boldface**, so that they can be easily distinguished from the rest of the messages. All the messages are alphabetized.

The first section lists messages which belong to all the objects that follow, so we show them separately to save space.

B.1 Basic Messages For All Objects

add-combo add-deadbolt add-listener **asleep?** **birth-machloc** change-description change-name **combos** combos-for-message **creator** **deadbolts** deadbolts-for-message del-combo del-deadbolt del-listener **description** **globally-unique-id** install **installed?** listen **listeners** **listenspace?** lookup-listeners **machloc** **object-locks** object-locks-for-message **pretty-name** set-combos set-deadbolts set-listeners **short-name** sleep **unique-id-number** wake-up

B.2 User's Messages

alias alias! **autologout-time** **balance** become-character change-password **characters** **creations** creator-guard-lock **current-character** define exit-whopr **logged-in?** **logged-out?** **namespace** **namespace?** re-alias re-define set! change-autologout set-creator-guard-lock sleep-character un-alias un-define **user-last-process-time** **user?** **user-uptime-status** whopr **whopr-user-magic-cookie**

B.3 Place's Messages

add-exit direction-of-portal **exit-map** **exits** found-exit? lock-exit **place?** portal-towards **portals** set-exit-lock **things** unlock-exit

B.4 Thing's Messages

mobile? ownable? owned? owner place

B.5 Person's Messages

act alias alias! announce borgcast change-threshold define describe drop emote go have-fit list-possessions look-around **mobile?** move **namespace namespace?** page **person? place possessions** re-alias re-define **restlessness** say set! take think **threshold** un-alias un-define utter whisper

B.6 Portal's Messages

destination pass-through **portal?**

B.7 Lock's Messages

key-descriptor lock? open? set-key-descriptor test-lock

B.8 Listener's Messages

actions add-action-to-back add-action-to-front add-trigger del-action del-ith-action del-ith-trigger del-trigger set-actions set-triggers take-action **triggers** triggers?

Appendix C Object References and Object Aliasing

Referencing an object by something other than its object ticket (such as its short name or its index at a place) is essentially making use of the object's `aliases`. An alias is a short, easy to type in, list of characters that the SchMUSE server will recognize as an object reference and translate into an object ticket. The kind of object referencing you can do by looking around (as discussed in the next section) is possible because of place dependent aliases: places basically keep track of the objects in them (index them, for one) so that you can easily access any of them in a number of different ways. This is what the first section of this appendix attempts to clear up.

There are two other kinds of aliasing available to a SchMUSEer. One is user and character aliasing (this may especially be helpful to those who DON'T have a client running on Scheme). The other has to do with server-defined objects (we have encountered some of them, such as lobby and the-creator, in our short scenario). We will briefly discuss these in the sections on aliases, below.

C.1 Referencing Objects At a Place (Place Dependent Aliases)

Basically, an object reference (you will see a variable <object-ref> in many help menus) is anything that the server recognizes as the reference to an object (i.e., anything you can type in that will give you back an object ticket), such as:

1. the object's unique ID (the number associated with every object ticket, mentioned earlier)
2. the index of the object in a place (displayed when you `look-around`)
3. the object's short name (nickname)
4. the object's pretty name
5. a part of the object's description (for example, if the object is described as '`(a singing farting troll)`', you can refer to it as `farting`, as long as no one else in the room is described as `farting` as well)
6. a combination of 2 and either 4 or 5, used when there is more than one occurence of a pretty name or a part of a description (for example, if there are two '`farting`' objects in the room, you can refer to them as `(farting . 0)` and `(farting . 1)`, going from top to bottom of the list; note zero indexing)
7. the object's type (the format is similar to 6 if more than one representative of a type is present)

C.1.1 Example: Referencing DML

Here is an example of how a result of issuing our `look-around` request (refer back to the chapter on objects and object tickets) can be used to reference an object at a place. Let's say you are trying to reference dml, whose description happens to be: '`(a very twisted hacker who is very much into sam & max)`'. Zig's description, on the other hand, is: '`(a crazy hacker whose motto is: sleep is for the weak!)`'. Here are several ways to reference this object:

1. 19920727230205668
2. 2
3. dml
4. "Daveman"
5. twisted (or sam, or max – you get the idea)
6. (hacker . 1) [zig would be = (hacker . 0)]
7. (person . 1)

C.2 User and Character Aliases

Both a user and a character object have a concept of `namespace`—a list of all the aliases either the user or his/her character knows about. This is a list of pairs of alias names and object tickets. You can view your namespace by typing `(namespace)`. You can add new aliases to your namespace with an `alias` command and delete the old ones with the `un-alias` command. Moreover, any object reference can be `re-aliased` to a new alias name.

We provide two convenient aliases you will probably use most often: `me` and `here`. `Me` is a user alias; it changes every time the user changes his or her character. Thus, typing `me` will return an object ticket for your current character (for example, if your current character is jack, typing `me` is equivalent to typing `jack`).

`Here` is a character alias. Every person "knows" what his or her current place is; it gets updated as he/she moves from room to room. Again, typing `here` at The Lobby, for instance, will give you back an object ticket for the lobby—this is equivalent to typing `lobby`.

While `me` and `here` are provided by the SchMUSE, you may find it useful to alias a few things yourself. The reason is that, while you can frob the things you see at a particular place when you look around, you have no way of accessing these things once you move out of the room unless you create an alias for them in your namespace. For example, once you `go` out of the lobby, you will not get an object ticket back if you type `rock`. Hal's office won't know what the rock is. This means that, unless you alias the rock while you are in the lobby, you cannot ask it for its pretty name, for example, while running around in the SchMUSE world.

This is how you would `alias` the rock (add it to your, the user's, namespace):

```
(alias lobby-rock rock)
```

Lobby-rock is the alias name you give the object, while rock is the object reference. Now, even if you are outside the lobby, you can send the rock different messages (such as `(owner lobby-rock)`), since your namespace knows what object the name `lobby-rock` refers to.

`Re-alias` and `un-alias` commands are similar. `Un-alias` only takes one argument, an alias name:

```
(un-alias lobby-rock)
```

and removes the entry for the rock from your namespace. `Alias!` is just like `alias`, except it does not complain if there is already an entry with an alias name `lobby-rock`—it simply replaces such an entry. `Define`, `set!`, and `un-define` are synonyms for `alias`, `re-alias`, and `un-alias`, supplied for people who type with a slight lisp.

Character's namespace is remarkably similar to the user's namespace. The commands are identical, except for the prefix '`char-`' added in front. For example, you could have chosen to alias the rock in you current character's (hacker's, in this case) namespace instead:

```
(char-alias lobby-rock rock)
```

If you change your characters and become Jack Vanilla again, you will not be able to refer to the rock any more, since it will not be in jack's namespace. Also, character aliases supercede the user ones, so that, if both the user and his or her character have an alias entry with the same alias name but referring to two different objects, the character alias wins.

C.3 Server Defined Objects

We have seen these from the moment we registered on the SchMUSE and looked around. The lobby, for example, is an object that was defined on the server, to be used as a birthplace for all users. Other places defined on the server are hal-office and eric-office. The portals connecting these are also provided by the server. The-creator, the-terminator, and the-forger are some server-defined users (who also have characters). At last, Mr. Potatoehead [sic], your generous benefactor, is also an object that has been created on the server. All of these the SchMUSE server knows about and will recognize, even though their object tickets are not in your namespace. You can always refer to the-creator by his nickname, for example, even if you are not in the same room as him.

Concept Index

Table of Contents