Learning ZIL

- or -

Everything You Always Wanted to Know
About Writing Interactive Fiction
But Couldn't Find Anyone Still Working Here to Ask

Comments to SEM

Conversion to Microsoft Word -- SEM -- 8/1/95

Table of Contents

Chapter 1: The Basics

1.1 The Basic Ingredients

When you write interactive fiction (hereafter IF), you will mostly be dealing with two kinds of things: objects and routines. There are two major categories of objects: rooms (such as the Living Room in *Zork I* or Joe's Bar in *Leather Goddesses*), and objects you can refer to, such as the brass lantern in *Zork I* or the dressing gown in *Hitchhiker's Guide*.

Routines are little sub-programs which perform a whole variety of functions. There are many kinds of routines, but the kind you'll be concerned with first are called action routines. These are directly linked to a room or object.

Much more detail on objects and routines in subsequent chapters.

1.2 The Handler

In IF, the player types an input, and the game must produce a response:

```
>HIT UNCLE OTTO WITH THE HAMMER
You knock some sense back into Uncle Otto, and he stops
insisting that he's Napoleon Bonaparte.
```

Somewhere, the game decided to print that response. The part of the game that printed that response is said to have handled the input. The input must be handled at some point; a non-response is always a no-no:

```
>UNCLE OTTO, REMOVE YOUR HAND FROM YOUR SHIRT
>
```

1.3 The Parser's Role

There's a notorious part of every IF program called the parser. It gets the first look at the input. If it decides that the input is indecipherable, for any of several reasons, it handles the input. For example:

```
>EXAMINE THE FLEECY CLOUD
[I don't know the word "fleecy."]

>ASK UNCLE OTTO ABOUT MOSCOW AND WATERLOO
[You can't use multiple objects with the verb "ask."]
```

Cases like these are called parser failures. (This is not to be confused with those times when the parser fails, which are called parser bugs.)

If the parser succeeds in digesting the input, it passes three pieces of information on to the rest of the program: the verb, the direct object, and the indirect object. Internally, these are called PRSA, PRSO, and PRSI. In the first example in section 1.1, PRSA is HIT, PRSO is the UNCLE OTTO object, and PRSI is the HAMMER object.

Not every input has a PRSI. For example, in:

```
>CALL THE FUNNY FARM
Men in white coats arrive and hustle Uncle Otto into
the wagon.
```

the verb is CALL and the PRSO is the FUNNY FARM object. In such a case, when there is no PRSI, the parser sets the value of PRSI to false. Furthermore,

not every input has a PRSO. Some examples of inputs where PRSO and PRSI are both false:

```
>YELL
>PANIC
>INVENTORY
```

Note that you cannot have a PRSI without also having a PRSO. Also note that every input has a PRSA.

1.4 The Basic Handling Sequence

After the parser identifies PRSA, PRSO, and PRSI, the game gets to decide who will handle the input. By convention, PRSI gets the first crack (providing there is a PRSI, of course).

What this means, is that if PRSI has an associated object action routine, that action routine tries to handle the input. If it does so, the current turn is considered to be complete. If not, then the PRSO's action routine is given the opportunity next. The PRSO and PRSI routines can give very specific responses to the input.

If the PRSO also fails to handle the input, the task falls to the routine associated with the verb. Because such a routine is the "last resort," and since it usually gives a very general response, it is called the default response.

Here's an example of how the response to an input might look depending on who handled it:

```
>HIT THE OAK CHEST WITH THE CROWBAR
The crowbar bends! It appears to be made of rubber, not
iron!
      → (handled by PRSI's action routine)
The sound of the impact reverberates inside the chest.
      → (handled by the PRSO's action routine)
Hitting the oak chest accomplishes nothing.
      → (handled by the verb default)
```

As you can see, the verb default is the least interesting and "colorful" of the responses. Of course, there's not enough space on the disk or time in the schedule to give every possible input its own special response; a good implementor must find the proper balance, deciding when to write a special colorful response, and when to let the relatively drab verb default do the handling.

There are other places where the game gets a chance to handle the input, but we'll get to that later on.

Chapter 2: Creating Rooms

2.1 What a Typical Room Definition Looks Like

Here's what the definition of the Living Room from Zork I looks like:

```
     <ROOM LIVING-ROOM
          (LOC ROOMS)
          (DESC "Living Room")
          (EAST TO KITCHEN)
          (WEST TO STRANGE-PASSAGE IF CYCLOPS-FLED ELSE
               "The wooden door is nailed shut.")
          (DOWN PER TRAP-DOOR-EXIT)
          (ACTION LIVING ROOM-F)
          (FLAGS RLANDBIT ONBIT SACREDBIT)
          (GLOBAL STAIRS)
          (THINGS <> NAILS NAILS-PSEUDO)>
```

Note that, as with everything you will ever write in ZIL, the parentheses and angle brackets are balanced.

Let's go over this room definition line by line. The first thing in a room definition is the word ROOM followed by the internal name of the room. This name, like all names of objects or routines in ZIL, must be one word (or, if more than one word, connected by hyphens) and must be all capital letters.

The second line, LOC, gives its internal location. All rooms are located in a special object called the ROOMS object.

The third line is the external name of the room, its DESC. This is what will appear in the output each time a player enters that room. Note: The internal and external names of a room (or object) do not need to be identical. For example, there's no reason the internal name couldn't be LIV-ROOM. It's up to the author. Usually, it's a trade-off: using the same name makes it easier to remember the internal name, but it means more typing.

2.2 Exits

The next several lines are all the exits from the Living Room. In general, the bulk of a room definition is its exits. The fourth line, (EAST TO KITCHEN), is an example of the simplest kind of exit, called a UEXIT (for unconditional exit). This means that when the player is in the Living Room and types EAST, the player will go to the Kitchen—in all cases.

The fifth line is a CEXIT (for conditional exit). If involves a global called CYCLOPS-FLED. A global is the simplest way that you store a piece of information in ZIL. (See section 5.2 and section 7.2.) In this case, CYCLOPS-FLED is either true or false depending on whether the cyclops has made a hole in the oak door. What this CEXIT means is that when the player types WEST, the player will be sent to Strange Passage if the global CYCLOPS-FLED is true. If CYCLOPS-FLED is false, the player will be told "The door is nailed shut." This piece of text inside quotes is called a string. The string is not required for a CEXIT; if it is omitted, the game will supply a default string like "You can't go that way."

The sixth line is an example of an FEXIT (for function exit). (Function is another word for routine.) The game will recognize this line as an FEXIT because of the "PER." In this case, if the player types DOWN, the routine called TRAP-DOOR-EXIT decides if the player can move, and if so, to where, and if not, what the response should be. In this case, it will say "You can't go that way." if you haven't moved the rug, or "The trap door is closed." if you have moved the rug but haven't opened the trap door.

There are two other types of exits, which don't happen to be used by the Living Room in Zork I. The NEXIT (for non-exit) is simply a direction in which you can never go, but for which you want something more interesting than the default "You can't go that way." response. The game will recognize it as an NEXIT because of the use of "SORRY." It might look something like this:

```
(NW SORRY "The soldier at Uncle Otto's front door
informs you that only Emperor Bonaparte is allowed
through.")
```

The other is the DEXIT (for door exit). This is similar to the CEXIT, substituting the condition of a door object for the global. It might look something like this. Note the "IS OPEN" which isn't found in a CEXIT:

```
(SOUTH TO GARAGE IF GARAGE-DOOR IS OPEN ELSE
"You ought to use the garage door opener.")
```

If the GARAGE-DOOR object is open, and the player types SOUTH, you'll end up in the GARAGE. Else, the player will be told the string. Once again, the string is optional. If no string is supplied, the response will be something like "You'll have to open the garage door, first."

2.3 Other Parts of a Room Definition

Getting back to the Living Room example, the next line defines the room's action routine, LIVING-ROOM-F. (The F at the end is short for "function.") You'll hear (a lot) more about a room's action routine in a while.

The next line contains those FLAGS which are applicable to this room. RLANDBIT (the R is for "room") means that the room is on dry land, rather than water (such as the Reservoir) or in the air (such as the volcano rooms in Zork II). ONBIT means that the room is always lit. Some flag names appear in every game; but you can make up special ones to fit the requirements of your own game. For example, SACREDBIT is special to Zork I, and means that the thief never visits this room. By convention, all flag names end with "BIT." For a complete list of commonly used flags, see Appendix B.

Forget about the GLOBAL and THINGS lines for now. You'll learn about the GLOBAL property in section 7.4, and can read about THINGS in Appendix A.

All these things—EAST, UP, FLAGS, ACTION, etc.—are called properties. As you'll see in a moment, objects have properties as well; some are the same as the properties that rooms have, but some are different. Although most properties are the same from game to game, you may occasionally want to create your own. For a complete list of commonly used properties, see the Appendix A.

Chapter 3: Creating Objects

3.1 What an Object Definition Looks Like

Here's what the definition of Zork I's brass lantern looks like:

```
<OBJECT LANTERN
        (LOC LIVING-ROOM)
        (SYNONYM LAMP LANTERN LIGHT)
        (ADJECTIVE BRASS)
        (DESC "brass lantern")
        (FLAGS TAKEBIT LIGHTBIT)
        (ACTION LANTERN-F)
        (FDESC "A battery-powered lantern is on the trophy
        case.")
        (LDESC "There is a brass lantern (battery-powered)
        here.")
        (SIZE 15)>
```

As you can see, there are some properties which appeared in the room example, but some new ones as well.

3.2 Object properties

LOC refers to the location of the object at the start of the game. In this case, the location of the lamp is in the Living Room room. Over the course of the game, the location of objects may change as the player does stuff. For example, if the player picked up the lamp, the LOC of the lamp would then be the PLAYER (sometimes called PROTAGONIST) object. If the player then dropped the lamp in the Kitchen, the Kitchen room would be the lamp's LOC.

The SYNONYM property is a list of all those nouns which can be used to refer to the lamp. The ADJECTIVE property is a list of those adjectives which can be used to refer to the lamp. An object, to be referred to, must have at least one synonym; the ADJECTIVE property is optional. In the case of the lamp, the player could refer to it using any of six combinations: lamp, lantern, light, brass lamp, brass lantern, brass light.

The DESC property, as with rooms, is the name of the object for external consumption. It will appear whenever a routine needs to "plug in" the name of the object. For example, the EAT verb default would use it to form the output: "I doubt the brass lantern would agree with you."

The lamp has two flags: the TAKEBIT means that the lamp can be picked up by the player; the LIGHTBIT means that the lamp can be lit. The lamp is not currently on; once it gets turned on, it will have the ONBIT, meaning that it is giving off light. The flags in the object definition are only those attributes which the object has at the start of the game.

The ACTION property identifies LANTERN-F as the action routine which tries to handle inputs relating to the lantern. For example, if the player typed THROW THE NERF BALL AT THE BRASS LANTERN, the LAMP object would be the PRSI, and the routine LANTERN-F would get the first crack at handling the input. If the player typed THROW THE BRASS LANTERN AT THE NERF BALL, then

the LAMP object would be the PRSO, and LANTERN-F would get a crack at handling the input provided that the nerf ball's action routine failed to do so.
The FDESC property is a string which is used to describe the brass lantern until the player picks it up for the first time; in other words, it describes its original state. The LDESC property is a string which subsequently describes the lantern when it's on the ground. These strings are used when a room description is given, which occurs when you enter a room or when you do a LOOK. If there are no FDESC or LDESC properties, an object will be described by plugging its DESC into a default: "There is a brass lantern here."
The SIZE property defines the size/weight of the object. This helps the game decide things like whether you can pick something up, or whether you're holding too much already. If no SIZE is given to a takeable object, the default is usually 5. A very light object, like a key or a credit card, might have a SIZE of 1 or 2. A very heavy object, like a trunk or anvil, might have a SIZE of 25 or 50. Don't worry too much when you're creating the object; you can always go back and tweak the sizes during the testing phase.

Chapter 4: Routines in ZIL

4.1 The Basic Parts

A routine is the most common item that makes up ZIL code. If you think of rooms and objects as the skeletal structure of a game, then routines are the blood and muscle that make the skeleton dance.

Like all things in ZIL, a routine must have balanced sets of parentheses and angle brackets. The basic parts of a routine look like this. Note how the angle brackets balance out:

```
<ROUTINE ROUTINE-NAME (argument-list)
     <guts of the routine>>
```

There are various conventions for naming routines; object and room action routines are usually the name of the object or room with "-F" appended. As usual, there's a trade-off between shorter, easier to type names and longer, easier to remember and understand names.

The argument list appears within parentheses after the routine name. Arguments are variables used only within the specific routine—unlike global variables, which are used by any routine in the game. In many cases, a routine will have no arguments; in that case, the argument list must still appear, but as an empty set of parentheses: ().

Here's an example of a couple of simple routines, just to show you what they look like. You don't have to understand them fully, just yet:

```
<ROUTINE TURN-OFF-HOUSE-LIGHTS ()
     <FCLEAR ,LIVING-ROOM ,ONBIT>
     <FCLEAR ,DINING-ROOM ,ONBIT>
     <FCLEAR ,KITCHEN ,ONBIT>>
<ROUTINE INCREMENT-SCORE (NUM)
     <SETG SCORE <+ ,SCORE .NUM>>
     <COND (,SCORE-NOTIFICATION-ON
          <TELL "[Your score has just gone up by "
               N .NUM ".]" CR>)>>
```

The first routine, called TURN-OFF-HOUSE-LIGHTS, makes the three rooms in the house dark. Note the empty argument list.

The second routine, INCREMENT-SCORE, has one local argument, NUM. This routine adds the value of NUM to the player's score; if the player has the notification feature turned on, this routine will inform the player about the increase.

4.2 Calling a Routine

Every routine in a game is activated by being called by some other routine. The PRSO and PRSI action routines, and the verb default routine, are called by a routine called PERFORM, which runs through the potential handlers and stops when someone has handled the input. In turn, an action routine can call other routines, as you will see.

A routine calls another routine by putting the name of the called routine at the appropriate place, inside brackets:

```
<ROUTINE ROUTINE-A ()
```

```
          <TELL "Routine-B is about to be called by
               Routine-A." CR>
          <ROUTINE-B>
          <TELL "Routine-B just called by Routine-A." CR>>
```

Sometimes, the caller may want to call the callee with an argument, in order to pass information to the callee. In that case, the argument list of the called routine must be set up to receive the passed arguments:

```
     <ROUTINE RHYME ("AUX" ARG1 ARG2)
          <SET ARG1 30>
          <SET ARG2 "September">
          <LINE-IN-RHYME .ARG1 .ARG2>
          <SET ARG1 28>
          <SET ARG2 "February">
          <LINE-IN-RHYME .ARG1 .ARG2>
          etc.>
     <ROUTINE LINE-IN-RHYME (ARG-A ARG-B)
          <TELL N .ARG-A " days hath " .ARG-B "." CR>>
```

Note that in the example above, routine RHYME has the notation "AUX" in its argument list before the two arguments. This means that the two arguments are auxiliary arguments, used within the routine RHYME, but are not passed to RHYME by whatever routine calls RHYME. No such notation appears in the LINE-IN-RHYME argument list, because ARG1 and ARG2 are being passed to LINE-IN-RHYME. Note that LINE-IN-RHYME calls the variables ARG-A and ARG-B instead of ARG1 and ARG2; this is completely arbitrary. The writer of LINE-IN-RHYME could have called them ARG1 and ARG2 if he/she wished. Remember, even though the routine LINE-IN-RHYME only exists in one place in your game code, it can be called any number of times by other routines throughout your game code. In the case of the routine LINE-IN-RHYME above, it must be passed two arguments every time it is called.

There is a third type of argument, the optional argument. When an argument list contains such an argument, denoted by "OPT" in the argument list, it means that the routine accepts that passed argument, but it doesn't require it.

The three types of arguments must appear in the argument list in the following order: passed arguments, optional arguments, and auxiliary arguments (which are also known as local variables). A routine which uses all three kinds might look something like this:

```
     <ROUTINE CALLEE (X "OPT" Y "AUX" Z)
          <some-stuff>>
```

It could be called in either of two ways:

```
     <CALLEE .FOO>
```

or

```
     <CALLEE .FOO .BAR>
```

Here's an example of a routine that takes an optional argument:

```
     <ROUTINE OPEN-DOOR (DOOR "OPT" KEY)
          <TELL "You open the " D .DOOR>
          <COND (.KEY
               <TELL " with the " D .KEY>)>
          <TELL "." CR>>
```

If this routine is called with <OPEN-DOOR ,OAK-DOOR> it will print "You open the oak door." If it is called with <OPEN-DOOR ,OAK-DOOR ,LARGE-RUSTY-KEY> it will instead print "You open the oak door with the large rusty key."

4.3 Conditionals

The guts of a routine are composed of things called CONDs, short for conditionals. A COND is sort of an if-then statement. In its simplest form, it looks something like this:

```
<COND (<if-this-is-true>
       <then-do-this>)>
```

The "if-this-is-true" section of the COND is called the predicate. A predicate is basically anything in ZIL whose value can be true or false. A bit later, we will look at the most common types of predicates.

A COND may have more than one predicate clause; if so, the COND will continue until one of the predicates proves to be true, and then skip any remaining predicate clauses:

```
<COND (<predicate-1>
       <do-stuff-1>)
      (<predicate-2>
       <do-stuff-2>)
      (<predicate-3>
       <do-stuff-3>)>
```

If <predicate-1> is true, then do-stuff-1 will occur, and the second and third clauses will be skipped. If <predicate-1> is false, the COND will then look at <predicate-2>, etc.

Often, a routine will have more than one COND:

```
<COND (<predicate-1>
       <do-stuff-1>)>
<COND (<predicate-2>
       <do-stuff-2>)>
```

In this construction, the second predicate clause will happen even if the first predicate clause turned out to be true.

4.4 Exiting a Routine

All routines return a value to the routine that called them. That value is the value of the last thing that the routine did. If that value is false, the routine is said to have returned false; if it returns any other value, the routine is said to have returned true:

```
<ROUTINE FIND-FOOD ("AUX" FOOD)
     <COND (<IN? ,HAM-SANDWICH ,HERE>
            <SET FOOD ,HAM-SANDWICH>)
           (<IN? ,CANDY-BAR ,HERE>
            <SET FOOD ,CANDY-BAR>)
           (<IN? ,BELGIAN-ENDIVE ,HERE>
            <SET FOOD ,BELGIAN-ENDIVE>)
           (T
            <SET FOOD <>)>
     .FOOD>
```

Remember, FOOD is an auxiliary local variable; its value is not passed to FIND-FOOD by the calling routine. However, FIND-FOOD does return the value of FOOD back to the calling routine. If any of the three predicates are true, FIND-FOOD will return the appropriate object, which means it has returned true. If the call to FIND-FOOD were a predicate, it would return true, as in:

```
<COND (<SET FOOD <FIND-FOOD>>
       <REMOVE .FOOD>
       <TELL "You eat the " D .FOOD
          ", and your hunger abates." CR>)>
```

If none of the three predicates in FIND-FOOD were true, then the value of .FOOD would be false, and the routine FIND-FOOD would return false. You can force a routine to return before reaching its end, by inserting <RTRUE> for "return true" or <RFALSE> for "return false" at any point in the routine. Note, however, that once the routine gets to this point, it will immediately stop executing, even if you haven't reached the bottom of the routine:

```
<ROUTINE STUPID-ROUTINE ()
     <TELL "This is a stupidly-designed routine...">
     <RTRUE>
     <TELL "...because the RTRUE prevents it from
getting to this second string!" CR>>
```

You can also force a routine to return a specific value, be it a string, an object, the value of a global variable, or the value of a local variable. For example, looking at two variations of FIND-FOOD:

```
<COND (<IN? ,CANDY-BAR ,HERE>
       <RETURN ,CANDY-BAR>)>

<COND (<IN? ,CANDY-BAR ,HERE>
       <RETURN "candy bar">)>
```

The first case returns the object named CANDY-BAR, the second returns the text string "candy bar." If you're confused about the significance of a routine returning a value, re-read this section after reading Chapter 5.

4.5 ZIL Instructions

There are a number of things in ZIL code which, at first glance, look like calls to a routine. However, if you look for the associated routine in your game code, you won't find it anywhere. These are ZIL instructions, the method by which the game communicates with the interpreter that runs the game on each individual micro. ZIL instructions are sometimes referred to as op-codes.

You've already seen a few of the most common instructions, such as FCLEAR and SET. Some instructions, such as those, take one or more arguments:

```
<FCLEAR ,SEARCHLIGHT ,ONBIT>
<SET .LAP-COUNTER 12>
<RANDOM 100>
```

Others take no argument:

```
<SAVE>
<QUIT>
```

And, like routines, some instructions accept an optional argument. A complete list of the current ZIL instructions can be found in Appendix D.

Chapter 5: Simple Action Routines

5.1 Simple Object Action Routines

Let's say you have an object AVOCADO with the property:

```
(ACTION AVOCADO-F)
```

The routine called AVOCADO-F is the action routine for the object. It might look something like this:

```
<ROUTINE AVOCADO-F ()
     <COND (<VERB? EAT>
            <REMOVE ,AVOCADO>
            <TELL "The avocado is so delicious that you
               eat it all." CR>)
           (<VERB? CUT OPEN>
            <FSET ,AVOCADO ,OPENBIT>
            <MOVE ,AVOCADO-PIT ,AVOCADO>
            <TELL "You halve the avocado, revealing a
               gnarly pit." CR>)>>
```

AVOCADO-F, like most action routines, is not passed any arguments, and it uses none itself, so it has an empty argument list.

This routine demonstrates one of the simplest and commonest types of predicates. <VERB? EAT> is true if EAT is the PRSA identified by the parser. If the input was something like EAT THE AVOCADO or DEVOUR AVOCADO, the parser would set PRSA to the verb EAT, and <VERB? EAT> would be true. If so, the next two things would happen. The first is to REMOVE the avocado; that is, to set its LOC to false, since once it is eaten, it is now nowhere as far as the game is concerned. The second thing is the TELL. TELL is a kind of routine called a macro which is in charge of printing everything the user sees as output to his input. In its simplest use, as in this example, TELL simply takes the given string and prints it. The CR after the string means "carriage return," and causes one to occur after the string. CR will sometimes appear as the instruction <CRLF>, meaning "carriage return line feed." CR and CRLF have the same effect.

If EAT was the verb, this string is printed, the input has been handled, and the turn is essentially over. However, if EAT wasn't the verb, then the COND continues to the next predicate. This second predicate is true if the verb is either CUT or OPEN. In fact, the VERB? predicate can take any number of verbs to check.

If the second predicate is true, the first thing that happens is the FSET, which means "flag set." In this case, it gives the AVOCADO the OPENBIT flag. Note that an object either has the OPENBIT or it doesn't; there is no such thing as a CLOSEDBIT; the absence of the OPENBIT is enough to tell that the object is closed. If you wanted to get rid of the OPENBIT, or any other flag, you would use the FCLEAR operation. For example, if the player closed the iron gate, you'd probably have something like:

```
<FCLEAR ,IRON-GATE ,OPENBIT>
```

Getting back to AVOCADO-F, the next thing that happens is to move the AVOCADO-PIT object to the AVOCADO. This will set the AVOCADO pit's LOC

to AVOCADO. Until this point, the pit probably had no LOC; that is, its LOC was false, meaning that as far as the game was concerned, it wasn't anywhere. Finally, the appropriate TELL is done. The input has now been handled.
If neither of these predicates proves to be true, because the verb is something other than EAT, CUT, or OPEN, then AVOCADO-F has failed to handle the input. If AVOCADO was the PRSI, the PRSO routine will then be given the chance. If AVOCADO was the PRSO, the appropriate verb default will handle the input.

5.2 Other Common Predicates

In addition to the VERB? predicate, let's look at some other common predicates. The most common one is EQUAL? as in:

```
<EQUAL? ,HERE ,DRAGONS-LAIR>
```

HERE is a global variable which is always kept set to the current room. So, if the player was in the Dragon's Lair room, this predicate would be true; otherwise it would be false.
Many global variables (such as the CYCLOPS-FLED example in section 2.1) are just true-false variables. Such a global can be a predicate all by itself, without angle brackets:

```
,CYCLOPS-FLED
```

If CYCLOPS-FLED were currently true, the predicate would be true. Note the comma before the global. When writing ZIL code, all global variables, room names, and object names must be preceded by a comma. However, local variables, those defined in the argument list and appearing only in the current routine, must be preceded by a period. Such a local variable could similarly be used as a predicate:

```
<ROUTINE OTTO-F ("AUX" OTTO-TO-SANITORIUM)
     <stuff-affecting-the-value-of-the-local-variable>
     <COND (.OTTO-TO-SANITORIUM
            <TELL "The van drives away, turning left
               toward Happy Dale." CR>)>>
```

Another common predicate is the FSET? predicate:

```
<FSET? ,AVOCADO ,OPENBIT>
```

will be true if the AVOCADO object has the OPENBIT flag set. Note the vastly important difference between FSET? and FSET—the first is a predicate, used to check whether a flag is set; the second is used to actually set the flag.
Another common predicate is IN? which checks the LOC of an object:

```
<IN? ,EGGS ,BASKET>
```

would be true if the LOC of the EGGS object was the BASKET object.
A routine can also be used as a predicate:

```
<COND (<IS-OTTO-ON-ELBA?>
     <TELL "A telegram arrives from Uncle Otto,
          complaining about his exile." CR>)>
```

If the routine IS-OTTO-ON-ELBA returned true, the predicate would be true, and vice versa. Here's what the routine might look like:

```
<ROUTINE IS-OTTO-ON-ELBA? ()
     <COND (<EQUAL? <LOC ,UNCLE-OTTO> ,PARIS ,WATERLOO>
```

```
                 <RFALSE>)
                (T
                 <RTRUE>)>>
```

Notice that EQUAL? can take any number of arguments, which will be compared to the first thing after the EQUAL? The predicate will be true if any of the subsequent arguments have the same value as the second argument. Thus, the predicate in IS-OTTO-ON-ELBA? would be true if the LOC of the UNCLE-OTTO object was either PARIS or WATERLOO.

[Put in something here about PDL's new P? predicate.]

5.3 More Complicated Predicates

The predicates you've already learned can be combined with several fundamental ZIL operations to be more useful. One of these is NOT. NOT simply changes the sense of the predicate. If

```
<FSET? ,TROPHY-CASE ,OPENBIT>
```

is false, because the trophy case is closed, then

```
<NOT <FSET? ,TROPHY-CASE ,OPENBIT>>
```

will be true. Once again, note the absence of a CLOSEDBIT. You check to see if the trophy case is closed by checking to make sure it doesn't have the OPENBIT.

Two or more simple predicates can be combined together using AND. In such a case, all of them must be true for the entire predicate to be true:

```
<AND <EQUAL? ,HERE ,LAUNDRY-ROOM>
     <FSET? ,DRYER ,ONBIT>>
```

This predicate will only be true if the player is in the laundry room and the dryer is on. If either part is false, the entire predicate will be false.

Similarly, two or more simple predicates can be combined using OR. In that case, the entire predicate is true if any of the parts are true:

```
<OR <IN? ,SABOTEUR ,ZEPELLIN>
    ,BOMB-DEFUSED
    <EQUAL? ,BLIMP-DESTINATION ,NEW-JERSEY>>
```

This predicate will be true if the SABOTEUR object is in the ZEPELLIN object, or if the global variable BOMB-DEFUSED is set to true, or if the global variable called BLIMP-DESTINATION is set to NEW-JERSEY. Of course, if two or more of the parts are true, the entire predicate remains true. Only if all three parts are false will this predicate be false.

As you can surmise, all sorts of hair-raisingly complicated predicates can be strung together using AND, OR and NOT. Here's what a more complicated version of AVOCADO-F might look like:

```
<ROUTINE AVOCADO-F ()
     <COND (<VERB? EAT>
            <COND (,AVOCADO-POISONED
                   <SETG PLAYER-POISONED T>
                   <REMOVE ,AVOCADO>
                   <TELL "You begin to feel sick." CR>)
                  (<AND <EQUAL? ,HERE ,GARDEN-OF-EDEN>
                        <IN? ,SNAKE ,TREE-OF-KNOWLEDGE>>
```

```
                    <MOVE ,APPLE ,HERE>
                    <TELL "The avocado is so
unappetizing. Suddenly, a seductive voice beckons from
the tree. A moment later, a succulent apple lands at
your feet." CR>)
                   (T
                    <REMOVE ,AVOCADO>
                    <MOVE ,AVOCADO-PIT ,PLAYER>
                    <TELL "You eat the entire avocado.
It was filling, if not tasty." CR>)>)
              (<AND <VERB? THROW>
                    <EQUAL? ,HERE ,MIDWAY>
                    <NOT ,BALLOON-POPPED>
                    <HAWKER-AT-CIRCUS>>
               <MOVE ,AVOCADO ,MIDWAY-BOOTH>
               <TELL "The avocado bounces off the balloon.
The hawker sneers. \"You'd have more luck with a dart,
kiddo! Only two bits!\"" CR>)
              (<AND <VERB? COOK>
                    <OR <NOT <EQUAL? ,HERE ,KITCHEN>>
                        <NOT <IN? ,COOKPOT ,OVEN>>>>
               <TELL "Even a master chef couldn't cook an
avocado with what you've got!" CR>)>>
```

Note some things about the above routine that are new to you. The first is the inner COND, which appears within the <VERB? EAT> clause of the main COND. Such nesting can continue more and more deeply without limit (other than the ability for people looking at the code to understand the resulting routine).

The second thing to note is the last predicate in that inner COND: it is simply a T. This is simply a predicate that is always true. The result in this case is that AVOCADO-F should always handle the input EAT AVOCADO. If the verb is EAT, the inner COND will check the first and second predicates. If neither is true, then the third clause, the one with the T predicate, will occur in all cases. Sometimes you'll see ELSE instead of T; these mean the same thing.

The third thing is the string in the second TELL, where the hawker speaks. Note the backslash before the hawker's quotes. These are necessary anytime you have quotes within a string; without the backslash, the compiler will think the quote means the end of the string rather than a character within the string, and get very confused. A confused compiler is not a pretty sight.

5.4 A Room's Action Routine

A room's action routine is (generally) not used for the purpose of handling the player's input directly, as with an object's action routine. These routines perform a wide range of tasks. All room action routines are passed an argument, which tell the routine for what purpose it is being called. By convention, this argument is called RARG (for room argument).

The most common use of a room's action routine is for the purpose of describing the room (due to a LOOK or due to entering the room for the first time, or due to entering the room in verbose mode). If a room's description never changes, it

can have an LDESC property, a string which is the room's unchanging description. More often than not, however, the room's description changes (a door closes, a roof collapses, a window gets broken, a tree gets chopped down...) and in these cases the room's action routine must handle the description. In this case, the action routine is called with an argument called M-LOOK:

```
<ROUTINE CAFETERIA-F (RARG)
<COND (<EQUAL? .RARG ,M-LOOK>
       <TELL
"This is a lunch room, with windows overlooking a
loading dock. ">
       <COND (<IN? LUNCH-CROWD ,CAFETERIA>
              <TELL
"Every table is jammed with patrons. ">)>
       <TELL "The only exit is north." CR>)>>
```

Notice how the room description changes, depending on whether or not the LUNCH-CROWD object is in the cafeteria. Also, notice the argument list with RARG. Finally, notice that the RARG in the predicate has a period in front of it; this is because it is a local variable in CAFETERIA-F. (It's okay for any number of routines to have a local variable with the same name; but every global variable, object, etc. must have a unique name.)

Another reason for calling a room's action routine is to print a message or perform some operation upon entering it. For example:

```
>NORTH
Upon entering the crypt, a cold icy wind cuts through
you, and you realize that the dreaded poltergeist is
near!

Crypt
This is a spooky room...
```

The phrase "Upon entering the crypt..." is probably being TELLed by an action routine that looks something like this:

```
<ROUTINE CRYPT-F (RARG)
     <COND (<EQUAL? .RARG ,M-ENTER>
            <MOVE ,POLTERGEIST ,HERE>
            <TELL
"Upon entering the crypt, a cold icy wind cuts through
you, and you realize that the dreaded poltergeist is
near!" CR CR>)
           (<EQUAL? .RARG ,M-LOOK>
            <TELL "This is a spooky room..." CR>)>>
```

There are two CRs after the M-ENTER TELL in order to create a blank line between it and the room name.

Very often, an M-ENTER clause just does some stuff invisibly:

```
<ROUTINE HAUNTED-PANTRY (RARG)
     <COND (<EQUAL? .RARG ,M-ENTER>
            <MOVE ,SKELETON ,HERE>
            <SETG SKELETON-SCARE T>)>>
```

This M-ENTER moves the SKELETON object into the pantry, and sets a global variable called SKELETON-SCARE to T. (The SETG command, for set global, is used to change the value of any global variable. Local variables are set with SET instead.)
There are several more uses of a room's action routine, but you're still too young and tender to hear about them. Besides, it's time for...

EXERCISE ONE
Using what you've learned so far, write a room definition, an object definition, and an action routine for that object. Then find a friendly imp (if possible) and have him or her critique it.

Chapter 6: Events

6.1 Definition

Not all text that appears in an IF game is in response to the player's input. Some text is the result of an event—the lamp burning out, an earthquake, the arrival of a murder suspect, etc. Such events are called interrupts.
The convention for naming an interrupt routine is to prefix it with "I-" as in I-GUNSHOT. Here's an example of what a simple interrupt might look like:

```
<ROUTINE I-OTTO-GOES-NUTS ()
     <FSET ,UNCLE-OTTO ,LOONEYBIT>
     <COND (<IN? ,UNCLE-OTTO ,HERE>
            <TELL
"Sigh; it appears that Uncle Otto's delusion has
returned; he has begun shouting orders to unseen
troops." CR>)>>
```

6.2 How The Interrupt System Works

During most turns, time passes within the story. The exceptions are parser failures, and certain game commands such as SCRIPT or VERBOSE. At the conclusion of each turn in which time has advanced, after the player's input has been handled, a routine called CLOCKER runs. This routine "counts down" all the interrupt routines that are queued to run; if a given routine's time has come, CLOCKER calls that routine.
Any interrupt which does a TELL should return true; any interrupt that doesn't should return false. This is for the benefit of the verb WAIT, which causes several turns to pass. Interrupts must terminate a WAIT in order to prevent this kind of thing:

```
>WAIT
Time passes...
    A truck begins speeding toward you.
    The truck loudly honks its horn.
    Since you refuse to move out of the way, the truck
merges you into the pavement.
```

The only way for CLOCKER to tell V-WAIT to stop running is for the interrupt routine to RTRUE, meaning, "I've done a TELL." In the reverse case, if an interrupt runs and RTRUEs but fails to TELL anything, the WAIT will terminate early, but for no apparent reason.

6.3 Queuing an Interrupt Routine

An interrupt routine is queued by you, the writer, at some point. It might be at the beginning of the game, or it might be in response to something the player did. One interrupt routine might queue a subsequent interrupt routine.
It is done using the routine QUEUE, which takes the name of the interrupt routine, and a number to indicate the number of turns from now you'd like the interrupt to run:

```
<QUEUE I-SHOOTING-STAR 10>
```

Ten turns from now, the interrupt routine called I-SHOOTING-STAR will be called by CLOCKER, and will do whatever it is set up to do.
An interrupt routine which is queued to 1 will run on the same turn, before the very next prompt. An interrupt which is queued to 2 will run after the following turn, and so forth.
An interrupt runs only once, and then isn't heard from again unless you requeue it. However, there is one important exception: if you queue an interrupt to -1, the interrupt will run every turn from then on, until you dequeue it.
A routine is dequeued simply by saying <DEQUEUE I-WHATEVER>.
An example of a routine queued to -1 would be the truck interrupt which was used as an example in section 4.2. In conjunction with a global variable called TRUCK-COUNTER, initially set to a value of zero, the routine might look like this:

```
<ROUTINE I-TRUCK ()
     <SETG TRUCK-COUNTER <+ ,TRUCK-COUNTER 1>>
     <COND (<EQUAL? ,TRUCK-COUNTER 1>
            <MOVE ,TRUCK ,STREET>
            <TELL
"A truck begins speeding toward you." CR>)
           (<EQUAL? ,TRUCK-COUNTER 2>
            <TELL
"The truck loudly honks its horn." CR>)
           (<EQUAL? ,TRUCK-COUNTER 3>
            <COND (<EQUAL? ,HERE ,STREET>
                   <JIGS-UP
"Since you refuse to move out of the way, the truck
merges you into the pavement.">)
                  (T ;"you've gotten out of the way"
                   <TELL
"The truck blasts you with hot exhaust fumes as it
rumbles past." CR>)
           (T ;"counter is 4"
            <DEQUEUE I-TRUCK>
            <SETG TRUCK-COUNTER 0>
            <TELL
"The truck vanishes in the direction of Hoboken."
CR>)>>
```

Notice the first item, where 1 is added to the value of TRUCK-COUNTER. This enables I-TRUCK to "know" how far through the truck sequence it is.
Also, notice the routine called JIGS-UP, which is being passed a string as an argument. This notorious routine is the routine which "kills" the player; it prints the string which gets passed to it, followed by something like "*** You have died ***." The JIGS-UP occurs only if the player remains in the street. Otherwise, the second part of that COND occurs, with the text about the truck rumbling (safely) past.
Finally, notice that the last predicate in the main COND is simply T. This is a predicate which is always true; anytime the COND reaches that point, the stuff in that clause will be executed. In this case, that will happen whenever TRUCK-COUNTER is 4. In addition to turning itself off using DEQUEUE, notice that

TRUCK-COUNTER is also set to 0. This is so that if and when I-TRUCK runs at some future point in the game, it will act properly.
Note the semi-coloned messages next to the two "T" predicates. These are called comments. Anything preceded by a semi-colon is ignored by the compiler when it comes time to compile your ZIL code. Therefore, comments like these can be placed anywhere to annotate your code but without interfering with it. You can also put a semi-colon in front of a routine or part of a routine, if you wish to remove it for now, but keep it around in case you decide to use it again in the future. This is called "commenting out" code.

6.4 Room M-ENDs

A simple way for an event to occur is by having an M-END clause in the room's action routine. At the end of every turn (but before CLOCKER is called) the current room's action routine is automatically called with the argument M-END. An M-END differs from an interrupt in that it cannot be queued, it can only be based on current conditions; further, it is limited to one room. Here's an example of an action routine with an M-END which is basically just for atmosphere:

```
<ROUTINE AIRPORT-F (RARG)
     <COND (<EQUAL? .RARG ,M-ENTER>
            <QUEUE I-STRAFING 5>)
           (<EQUAL? .RARG ,M-LOOK>
            <TELL
"You are on the tarmac of an airport runway..." CR>)
           (<EQUAL? .RARG ,M-END>
            <TELL "A plane zooms low overhead." CR>)>>
```

Alternatively, an M-END can be more of an event:

```
<ROUTINE AIRPORT-F (RARG)
     <COND (<AND <EQUAL? .RARG ,M-END>
                 ,BATTLE-BEGUN>
            <JIGS-UP
"The shadow of a bomber sweeps across the airfield. The
tarmac where you were standing is now a giant hole, and
you are now tiny bits of exploded flesh.">)>>
```

Chapter 7: Let's Learn a Whole Lot More About ZIL Code

7.1 Mathematical Expressions in ZIL

ZIL uses an arithmetic notation called prefix notation. In this form of notation, the operator (+ or - or * or /) comes before the operands, rather than between them. For example if you wanted to add 4 and 7, it would look like this:

```
<+ 4 7>
```

And if you wanted to subtract 10 from 50, it would look like this:

```
<- 50 10>
```

Multiplication and division, though rarely used in game code, look thus:

```
<* 5 12>
</ 20 4>
```

For a more complicated example, let's look at (10 + 5) * (26/2 -5) in prefix notation:

```
<* <+ 10 5> <- </ 26 2> 5>>
```

Pay close atttention to the balancing of the angle brackets.

Most of the time, in game code, you'll be performing arithmetic operations on variables and routines, rather than numbers. For example, to subtract 10 from a global variable called SPACESHIP-TEMP, which is keeping track of how cold it's getting in your leaking spacecraft:

```
<SETG SPACESHIP-TEMP <- ,SPACESHIP-TEMP 10>>
```

If there were a routine called POPULATION which took a room and returned the number of people there, you might see an expression like this:

```
<COND (<EQUAL? <+ <POPULATION ,BAR>
                  <POPULATION ,RESTAURANT>
                  <POPULATION ,COFFEE-SHOP>> 101>
       <MOVE ,POLICEMAN ,RESTAURANT>
       <TELL
"A policeman shows up, loudly blowing his whistle.
\"Occupancy by more than 100 persons is unlawful and
hazardous!\" he shouts." CR>)>
```

Actually, in this last example, <EQUAL? ... 101> would probably want to be <G? ... 100> since if the three populations added up to 103 or 154, you'd want the policeman to show up. G? is a predicate meaning "greater than," and takes two arguments. The predicate will be true if the first thing is greater than the second thing. There is, of course, a L? predicate for "less than."

A final note: ZIL can only deal with integers (anywhere in the range from -32767 to 32767). You cannot use a non-integer value, such as 7.5 or 3.625.

There is a useful instruction in ZIL called MOD. It takes two values, and returns the remainder of dividing the first value by the second value. Example:

```
<MOD ,FAT-LADYS-WEIGHT 10>
```

If the global variable FAT-LADYS-WEIGHT was currently 425, then this MOD would return 5 (425 divided by 10 is 42 with a remainder of 5). If FAT-LADYS-WEIGHT were instead 420, then this MOD would have a value of 0. MOD is useful because ZIL can only do integers. Normally, if you divide 425 by 10, you get 42, and the remainder is essentially thrown away; if you need to save the remainder of a division for some reason, the only way to do so is with MOD.

7.2 Global Variables

We have already seen global variables and a number of their uses—for example, CYCLOPS-FLED from back in section 2.2. Recapping, globals are the primary method of storing information about the state of the game's universe. Their value is changed by the game code whenever appropriate, using SETG (for "set global"). If there were a global called PIZZAS-EATEN, then it might be changed in your code as follows:

```
<TELL "You swallow another pizza." CR>
<SETG PIZZA-EATEN <+ ,PIZZA-EATEN 1>>
```

In other words, add 1 to the current value of the global; if it's currently 5, make it 6, for example. The values of globals stay the same until your code changes them—or until the player restarts the game, at which point they would resume their initial values.

When you decide to create a global variable, first you must define it. This definition should look something like this:

```
<GLOBAL GLOBAL-NAME INITIAL-VALUE>
```

This definition should not have a comma before the global name. The definition can go anywhere at top level; that is, it can go anywhere at all in your code, but not inside a routine or object definition. (With very few exceptions, any routine, global definition, or object definition can go anywhere at top level; however, it's good form to keep related items grouped together.) Here are some examples of typical global definitions.

```
<GLOBAL SECRET-PASSAGE-OPENED <>>
<GLOBAL SLEEPY T>
<GLOBAL FUSE-COUNTER 0>
<GLOBAL NUMBER-OF-MATCHES 5>
```

The first two globals are used simply as true-false indicator. (Such a global, which is either true or false, is sometimes referred to as a "flag." This shouldn't be confused with FLAGS such as TAKEBIT and RLANDBIT.) The first one, SECRET-PASSAGE-OPENED, is being defined with an initial value of false (<> means false) because, at the beginning of the game, the secret passage is closed. The second one, SLEEPY, is being defined with an initial value of T, presumably because the player begins the game in a tired state. Globals can also be used to hold a numerical value, like the third and fourth examples. Global variables can also be set to objects, rooms, strings, and even routines. For example, the global HERE is always set to the player's current room. However, it sometimes causes the compiler confusion to define a global's value as an object; it's safer to say: *[is this still true?]*

```
<GLOBAL HERE <>>
```

and then early in the game, such as in the startup routine (which is always called GO), you can say:

```
<SETG HERE ,FRONT-PORCH>
```

where the FRONT-PORCH room is the opening room of the game.

Globals are one of the few things in the ZIL environment that you may run short of; a maximum of only 240 are allowed in the entire game (although, if you do run short, there are some tricks...)

7.3 The Containment System

This has nothing to do with nuclear power plants or with halting the spread of Communism. It is the inter-related system of object locations which is one of the pillars of ZIL.

As you'll recall, every object in ZIL has a LOC. All rooms are located in a special object called ROOMS. Objects are located "on the ground" of rooms, or inside other objects. Some objects might not have any LOC at a given point in time; their LOC is false.

The containment system determines a number of very important things. One of these is whether a given object is referenceable. Remember, the parser identifies a PRSO and PRSI; to be identified by the parser, an object must be present and visible (unless the verb is a special one, like FOLLOW, where the PRSO isn't required to be present).

For example, if an object isn't in the same room as the player, or the object is present but inside a closed container, or if the object has its INVISIBLE flag set, the object won't be currently referenceable. If the player refers to it in his or her input, the parser will fail, responding "You can't see [that] here!"

To produce the location of an object, you simply say:

```
<LOC ,OBJECT-NAME>
```

For example, you might have a predicate like:

```
<EQUAL? <LOC ,HORSE> ,STABLE ,BARN ,LARRYS-BEDROOM>
```

which would be true if the LOC of the horse was any of those three rooms. You could also use LOC like this:

```
<SETG GLOWING-ROOM <LOC ,LEAKY-NUCLEAR-WASTE-DRUM>
```

You can also use IN? as a predicate to check the LOC of an object:

```
<IN? ,PICKLE ,BARREL>
```

would be true if the BARREL object was the LOC of the PICKLE object. Note that IN? takes only two arguments, an object and a possible location. You cannot give several possible locations, as in <IN? ,PICKLE ,BARREL ,JAR>; you would have to use an EQUAL? construct like the HORSE example a few lines back.

You change the LOC of an object using MOVE:

```
<MOVE ,HORSE ,STABLE>
<TELL "The horse gallops off toward the stable." CR>
```

Whatever the LOC of the horse was before, it will now be the stable. There's no harm done if the horse's LOC was already the stable. If you want to "get rid" of an object, you REMOVE it:

```
<REMOVE ,HORSE>
<MOVE ,GLUE ,OUTPUT-HOPPER>
<TELL
"The horse obediently limps into the glue machine,
which shakes and rattles for a minute. Suddenly, the
machine becomes still, and a bottle of glue appears in
the output hopper." CR>
```

Working the other way, to find the contents of a given object, you need two commands, FIRST? and NEXT?. Let's say you had an object called KITCHEN-CABINET, which contained a pitcher, a serving spoon, and a severed head:

```
<FIRST? ,KITCHEN-CABINET>
```

would return the object PITCHER. Then

```
<NEXT? ,PITCHER>
```

would be the serving spoon, whose NEXT? would be the severed head. Since the severed head is the last object contained by the cabinet, then

```
<NEXT? ,SEVERED-HEAD>
```

would, by definition, be false. FIRST? or NEXT? can be used as a predicate:

```
<COND (<AND <VERB? SHAKE>
            <FIRST? ,PRSO>>
       <TELL
"You hear something rattling around inside " D ,PRSO
"." CR>)>>
```

7.4 Globals and Local-Globals

Now we're going to discuss global objects. Don't confuse this with global variables, discussed earlier in section 7.2. A global variable is a variable whose value can be used anywhere in your ZIL code; a global object is an object which can be referenced by the player anywhere in the game.

Some objects can be referenced at all times, in all locations. For example, an AIR or GROUND object, or body parts such as the HANDS object. Such objects are called global objects. There is a special object, like the ROOMS object, called GLOBAL-OBJECTS; in order to make an object referencable at any time, define it with (LOC GLOBAL-OBJECTS).

Similarly, there is an object called GENERIC-OBJECTS. Concept-objects, such as the MURDER or NEW-WILL objects in Deadline, belong there. These objects can be talked about, but not seen or touched.

There's another class of objects, which are referenceable in more than one location, but not in all locations. A classic example is a door, which is in the two rooms on either side of the door, but not in any other rooms. Other examples are things like WATER, TREES, or STAIRS. Such objects are called local-globals and are "stored" in another one of those special objects, called LOCAL-GLOBALS. The definition of a local-global might look like this:

```
<OBJECT RIVER-BANK
     (LOC LOCAL-GLOBALS)
     (DESC "river bank")
     etc...>
```

But that's not enough. You also need to tell the game what subset of rooms this local-global is referenceable in. You do this using the GLOBAL property. (Remember it from the LIVING-ROOM definition way back in section 2.1?) The GLOBAL property tells the parser all the local-globals that are referenceable in each room. In the RIVER-BANK example, the room definitions for the rooms called LEFT-BANK and RIGHT-BANK would each need a line:

```
(GLOBAL RIVER-BANK)
```

A GLOBAL property can contain any number of objects (well, up to 31, anyway):

```
(GLOBAL RIVER-BANK RIVER TREE EIFFEL-TOWER KIOSK)
```

There are some semi-obvious restrictions on globals and local-globals. They can never be takeable. They cannot be containers; if you had a wastebasket local-global on a number of street corners, and the player threw a Big Mac wrapper in it at one location, the wrapper would be sitting in the wastebasket at every corner! Similar problem with having a global or local-global which you can turn on to produce light, etc.

7.5 The Full Glory of TELL

It's time to learn the full power of the TELL Remember how important TELL is: almost every single character of text output that appears on the player's screen does so through the use of TELL.

TELL prints text that is given to it in a number of different forms. The most common form is a string:

```
<TELL "This is just a plain old string.">
```

You can also give TELL the name of a global whose value is a string. For example if you defined a global thusly:

```
<GLOBAL  CANT-SEE-ANY "You can't see any ">
```

or if you set a GLOBAL thusly:

```
<SETG CANT-SEE-ANY "You can't see any ">
```

then you could use it in a TELL thusly:

```
<TELL ,CANT-SEE-ANY "rainbow here." CR>
```

which would appear in the game's output as "You can't see any rainbow here." You can also give TELL a call to a routine which returns a string. For example:

```
<ROUTINE PICK-STRING (OBJ)
<COND (<EQUAL? .OBJ ,DRAGON>
       <RETURN "nasty">)
      (<EQUAL? .OBJ ,FOX>
       <RETURN "sly">)
      (T
       <RETURN "hungry">)>
       <TELL
"The animal looks very " <PICK-STRING ,PRSO> "." CR>
```

Very often you want TELL to print the DESC of a room or object. To do this, you would include D ,OBJECT-NAME. For example:

```
<TELL "The " D ,LARGE-KEY " doesn't fit the tiny lock."
CR>
```

If the DESC of the LARGE-KEY object was "large key" then this TELL would produce "The large key doesn't fit the tiny lock." You can also use D ,GLOBAL-VARIABLE when the global variable in question is set to a room or object. This is very commonly done with the global variables PRSI and PRSO. For example, the V-EAT verb default often looks like this:

```
<ROUTINE V-EAT ()
     <TELL
"I doubt the " D ,PRSO " would agree with you." CR>>
```

If the direct object of the input was the SWORD object, the parser would set PRSO to SWORD. If the SWORD had (DESC "elvish sword") then V-EAT would print "I doubt the elvish sword would agree with you."

In these examples, D is called a tell token. Most games have a number of tell tokens in addition to D; the writer can add them, but don't worry now about how to do so. Two of the most common tell tokens are A and T. These are used in conjunction with two flags called the VOWELBIT and the NARTICLEBIT.
T ,OBJECT-NAME means that TELL should print a space followed by the word "the" followed by another space followed by the object's DESC. Using the T tell token, the V-EAT from above would look like this:

```
<ROUTINE V-EAT ()
     <TELL
"I doubt" T ,PRSO " would agree with you." CR>>
```

However, if an object has a FLAG called the NARTICLEBIT, the T tell token knows not to print "the" before the DESC. For example, if an object called CROWDS had a DESC "lots of people" then you'd want V-EAT to print "I doubt lots of people would agree with you." rather than "I doubt the lots of people would agree with you."
The tell token A does the same thing as the tell token T except, of course, printing the indefinite article ("a") rather than the definite article ("the"). The A tell token has one additional wrinkle, though; it checks whether the object in question has a FLAG called the VOWELBIT to decide whether to print "a" or "an" before the DESC. Here's a TELL and how it would handle three different object DESC's:

```
<TELL "It looks just like" A ,PRSO "." CR>

<OBJECT TEA-BAG
(DESC "tea bag")
(FLAGS TAKEBIT)
etc...>

>EXAMINE TEA BAG
It looks just like a tea bag.

<OBJECT APPLE
(DESC "apple")
(FLAGS TAKEBIT VOWELBIT)
etc...>

>EXAMINE APPLE
It looks just like an apple.

<OBJECT VERMICELLI
(DESC "vermicelli")
(FLAGS TAKEBIT NARTICLEBIT)
etc...>

>EXAMINE VERMICELLI
It looks just like vermicelli.
```

If you want TELL to print something whose value is a number, use the tell token N. For example if a global variable called DIAL-SETTING was currently set to 94, then

```
<TELL "The dial is currently set to " N ,DIAL-SETTING
"." CR>
```

would print "The dial is currently set to 94."

Finally, TELLs can do carriage returns. A carriage return puts the text output point at the beginning of the next line. You can have TELL do a carriage return by putting CR outside of a string, or a vertical bar (upper case backslash key) inside a string:

```
<TELL "You are knocked unconscious." CR
"Later, you come to.">
<TELL "You are knocked unconscious.|
Later, you come to.">
```

would both print out as:

```
You are knocked unconscious.
Later, you come to.
```

The CR or vertical bar prevents "Later..." from appearing on the same line, immediately after "...unconscious." If you wanted a blank line between the two lines, you'd simply use two CRs (or two vertical bars).

Note: When a TELL displays a bunch of text which is longer than the width of the player's screen you do not have to worry about putting in a carriage return when the text reaches the right hand margin. The game will do this automically for you.

When you are typing a string in a TELL, you can hit the RETURN/ENTER key on your keyboard as much as you want; it won't affect where CRs occur when the game is played. Example:

```
<TELL "You walk down the hall
for a long time. Suddenly,
a trap door opens
under you!" CR "You fall into darkness." CR>
```

would appear on the screen like this:

```
You walk down the hall for a long time. Suddenly, a
trap door opens under you!
You fall into darkness.
```

A TELL should end in a carriage return if it concludes the handling of the input, as most TELLs do. This ensures the blank line before the next input prompt. The game will automatically put in a CR before the prompt (unless the player is in superbrief mode), and you'll end up with the proper-looking format:

```
>WALK AROUND THE TREE
You circle the trunk. Fun, huh?

>
```

If you leave the CR off the end of this TELL, you'd get:

```
>WALK AROUND THE TREE
You circle the trunk. Fun, huh?
>
```

Even worse, if you leave the CR off, and the player is in superbrief mode:

```
>WALK AROUND THE TREE
You circle the trunk. Fun, huh?>
```

Note how the prompt appears on the same line as the text output.

7.6 Vehicles

Most of the time, the LOC of the player is a room. However, there are cases where the PLAYER object is moved into another object; this type of object is called a vehicle.

The term vehicle is somewhat of a misnomer. The earliest examples of vehicles, such as the raft in Zork I or the balloon in Zork II, did move the player around from room to room. However, a vehicle has come to mean any non-room object which the player can enter, which includes such stationary things as a chair or bed.

To create a vehicle, give the object in question the VEHBIT flag. In addition, vehicles should all have the CONTBIT, OPENBIT, and SEARCHBIT as well.

When a player is in a vehicle, it will be mentioned in the room description thusly:

```
     Living Room, on the couch
          You are in a starkly-modern living room with...
```

or

```
     Bulb-Changing Room, on the stepladder
          This is a room whose ceiling is covered with
     sockets...
```

The purpose of vehicles are several-fold. The first is that they allow a player to be in both an object and a room at the same time. For example, if you sit on the bar stool, the STOOL object becomes the LOC of the PLAYER object. However, the stool is in the PUB room, so the player is (indirectly) still in the PUB room, and can see it as well as anything else in it.

Another purpose is that it gives the vehicle an opportunity to handle the input, via M-BEG, as you'll read about in the section called The Bigger Picture.

Finally, some vehicles do actually move the player around from room to room, often in interesting ways.

Chapter 8: The Bigger Picture

8.1 The Main Loop

When a player boots the game, the first thing the interpreter does (as far as you're concerned) is to call a routine called GO. This routine should include things like the opening text, the title screen graphic, a call to V-VERSION (to print all the copyright and release number info), and a call to V-LOOK (to describe the opening location). It should also queue any interrupts whose runtime is based on the start of the game, such as the interrupt that causes the earthquake in *Zork III*, or the interrupt that causes the Feinstein to blow up in *Planetfall*.

The last thing that GO should do is call the routine called MAIN-LOOP. MAIN-LOOP is sort of the king of ZIL routines; other than GO, every routine in the game is called by MAIN-LOOP, or by a routine that is called by MAIN-LOOP, or by a routine that is called by a routine that is called by MAIN-LOOP, etc. MAIN-LOOP is basically a giant REPEAT which loops once for each turn of the game. Simplified, it works something like this:

```
<ROUTINE MAIN-LOOP (argument-list-from-hell)
     <REPEAT ()
          <PARSER>
          <COND (<did-the-parser-fail?>
                 <AGAIN>)>
          <PERFORM ,PRSA ,PRSO ,PRSI>
          <COND (<did-this-input-cause-time-to-pass?>
                 <call-room-function-with-M-END>
                 <CLOCKER>)>>>
```

First, the parser is called. If the parser fails, it prints some failure message, and the turn is considered complete. AGAIN sends you to the top of the REPEAT, and the new turn begins. If the parser succeeds, it identifies the PRSA, PRSO, and PRSI. PERFORM is called, and uses that information to offer the PRSI's and PRSO's action routine a chance to handle the input. If not, it lets the verb default handle the input. Finally, unless the input was a non-time-running one, such as SUPERBRIEF or UNSCRIPT, the MAIN-LOOP causes events to occur via M-END and CLOCKER.

The MAIN-LOOP runs over and over until something, such as JIGS-UP or V-QUIT, tells the interpreter to end the game. This is done using the <QUIT> instruction.

The MAIN-LOOP does lots of other stuff as well. For example, if the player used a pronoun, such as IT in his/her input, in place of an object, the MAIN-LOOP decides what the word refers to, and substitutes the object. Therefore, it would convert an input like GIVE IT TO RIFLEMAN to GIVE BULLET TO RIFLEMAN.

8.2 More About PERFORM

Up to now, you've been told that PERFORM gives three routines the opportunity to handle the input: the PRSI's action routine, the PRSO's action routine, and the verb default routine. It can now be revealed, for the first time ever on nationwide

TV, that this was a gross simplification; there are actually many more places where the input can be handled.

Before anything else, PERFORM gives the WINNER action routine an opportunity. WINNER is a global variable which is usually set to the PLAYER object. This is the object that represents the player and gets moved around from location to location as the player moves, and which contains all of the player's inventory. Note that this is not the same as the ME object, which is used when you use ME in an input, as in KILL ME.

If you give the PLAYER object an action routine, such as PLAYER-F, then PLAYER-F will get the first opportunity to handle the input. An example of how this is used is when the player is a ghost in *Zork I*. This allows all the things which are handled differently when the player is a ghost to be handled in one spot. (See DEAD-FUNCTION in the *Zork I* ZIL files.)

Sometimes, the PLAYER object is not the WINNER. This is most often the case when you are talking to a character in the game; for that turn, that character is the WINNER, and the character's action routine gets the opportunity to handle the input. You'll hear more about this in the section on Actors.

Next, PERFORM gives the room's action routine an opportunity by calling it with an argument called M-BEG. (Actually, if the player's LOC is not a room, PERFORM first calls the vehicle's action routine.) Here's an example of an room action routine with an M-BEG clause:

```
<ROUTINE TORTURE-CHAMBER-F (RARG)
     <COND (<AND <EQUAL? .RARG ,M-BEG>
                 ,PLAYER-STRAPPED
                 <VERB? TAKE WALK LEAP ATTACK>>
            <TELL
"You're strapped to the wall and can't move!" CR>)>>
```

After the M-BEG, PERFORM sees if there is a pre-action, and if so, gives that routine an opportunity to handle the input. A pre-action is a routine associated with a particular verb, and which gets this early handling opportunity, to compensate for the fact that the verb default gets such a late opportunity. You'll find out in the section on the Syntax File how you define a pre-action. By convention the pre-action for a verb called V-FOO is called PRE-FOO.

Here's a common example of how a pre-action is useful. Let's say you have a verb SHOOT. Obviously, if the player doesn't have the GUN object, he can't shoot anyone or anything. You'd want the handler for such an input to say, "You don't have a gun!" or some such. If you wait until the V-SHOOT verb default to handle it, then an action routine may incorrectly handle the input before then. For example, TIN-CAN-F might simply check <VERB? SHOOT> and print "You hit a bull's-eye, knocking the can off the fence." On the other hand, you don't want to have every action routine that handles SHOOT have to check whether the player has a gun; that's very inefficient. The answer is to have a pre-action, called PRE-SHOOT, which looks like this:

```
<ROUTINE PRE-SHOOT ()
     <COND (<IN? ,GUN ,PLAYER>
            <RFALSE>)
           (T
```

```
                <TELL "You don't have a gun!" CR>)>>
```

Finally, the PRSI's action routine is given the opportunity, etc. There's one last complication: something called a CONTFCN. If the LOC of the PRSI has a property called CONTFCN, than the routine named in the CONTFCN property is first given the opportunity to handle. The same thing happens before the PRSO's action routine. I've never used a CONTFCN myself, so don't worry if you don't understand the concept.

[Stu—please write something about BE-verbs here.]

8.3 Calling PERFORM Directly

As you've already learned, PERFORM is called by the MAIN-LOOP for the purposes of handling the input. It is called using the PRSA, PRSO, and PRSI identified by the parser. However, it is very common and very useful to call PERFORM yourself, with a different PRSA or PRSO or PRSI. The most common reason for calling PERFORM yourself is to avoid duplicating code.

PERFORM is called with one required argument, the PRSA, and two optional arguments, the PRSO and the PRSI. One complication: when you use PERFORM, you must refer to the PRSA by its internal name, which is in the form ,V?VERB. Example: whereas you might say <VERB? SMELL>, you would say <PERFORM ,V?SMELL>.

Here's what a few potential calls to PERFORM might look like:

```
     <PERFORM ,V?LEAP>
     <PERFORM ,V?EAT ,PIZZA>
     <PERFORM ,V?TAKE-FROM ,CANDY ,BABY>
     <PERFORM ,PRSA ,ELVISH-SWORD>
     <PERFORM ,V?GIVE ,PRSI ,PRSO>
```

In the first example, PRSO and PRSI will both be false in the new PERFORM. In the second example, PRSI will be false. Notice the use of PRSA in the fourth example; in this case, you are leaving PRSA the same, and are presumably just changing the PRSO. In the last example, notice that PRSI comes before PRSO; in this case, the PRSI will become the PRSO and vice versa. (This is a very common case, which you'll hear more about in the chapter on syntaxes.)

Here's an example of a case where you would call PERFORM. Let's say you had a verb default that looked like this:

```
     <ROUTINE V-SHOOT ()
          <COND (<FSET? ,PRSO ,ACTORBIT>
                 <TELL "An expert shot fells" T ,PRSO>
                 <JIGS-UP ". The police arrive, and after a
     long and lurid trial, you get the chair.">)
                (T
                 <TELL "The shot ricochets off" T ,PRSO ",
     almost hitting you." CR>)>>
```

Then you might have an action routine for the PISTOL object which looked like this:

```
     <ROUTINE PISTOL-F ()
          <COND (<AND <VERB? FIRE>
                      <PRSO? ,PISTOL>>
```

```
                <PERFORM ,V?SHOOT ,PRSI>
            <RTRUE>)>>
```

The effect of this PERFORM would be to take an input like FIRE THE PISTOL AT UNCLE OTTO and have it executed as though the player had actually typed SHOOT UNCLE OTTO. This obviates the need to handle the shooting in PISTOL-F; instead, V-SHOOT, which is already set up to handle a shooting, gets to handle it.

Note the RTRUE after the PERFORM. In virtually all cases, a PERFORM must have an RTRUE after it. Reason: PERFORM itself usually returns false. If the PERFORM were the last thing in the action routine, which it usually is, then the action routine would in turn RFALSE to the original PERFORM—which is still going on! The original PERFORM would then think that the action routine hadn't handled the input, when in fact it had. If the RTRUE were missing from PISTOL-F, something like this would happen:

```
>FIRE THE GUN AT THE FIRE HYDRANT
The shot ricochets off the fire hydrant, almost hitting
you.
You can't fire the gun -- it's not gainfully employed!
```

8.4 Flushing inputs

The player is permitted to type multiple commands on the same input line:

```
>NORTH. NORTH. OPEN TRAP DOOR. DOWN. TAKE EGG
```

However, sometimes something will happen in the middle of executing this string of inputs that will possibly make the player want to rethink the subsequent moves: something like a character appearing or attacking, the player tripping and dropping something or everything, and so on. Example:

```
>NORTH. OPEN TRAP DOOR. DOWN. EAST
Dungeon
Your torch slips out of your hands as you open the
heavy trap door.

It is pitch black. You are likely to be eaten by a
grue.

Oh no! You have wandered into the slavering fangs...
```

Clearly, it is only fair to flush the inputs after the OPEN TRAP DOOR, and give the player a chance to reconsider his or her descent into darkness.

Until recently, this was done by putting RFATAL (for return fatal) in the object's action routine:

```
<ROUTINE TRAP-DOOR-F ()
     <COND (<AND <VERB? OPEN>
                 <NOT <FSET? ,TRAP-DOOR ,OPENBIT>>>
            <FSET ,TRAP-DOOR ,OPENBIT>
            <COND (<IN? ,TORCH ,PLAYER>
                   <MOVE ,TORCH ,HERE>
            <TELL "Your torch slips out of your hands
as you open the heavy trap door." CR>
            <RFATAL> ;"flush any additional inputs")
```

```
               (T
                <TELL "Oomph! You open the trap door. It
     sure is heavy!" CR>)>)>>
```

The RFATAL returns a value called M-FATAL to PERFORM, which in turn returns M-FATAL to the MAIN-LOOP. The MAIN-LOOP then knows to ignore any subsequent commands that were on the input line.

The problem with this scheme was that if the action routine called another routine, which then called another routine, and so on, then each of these deeper and deeper routine calls would all have to carefully pass the M-FATAL upwards to PERFORM. Thus, lots of extra code and lots of chances to screw up.

Therefore, a new and simpler method has been developed. They use a global variable called P-CONT, which the parser used to keep track of whether there are additional inputs on the input line. If you want to flush any subsequent inputs, just <SETG P-CONT -1>. This can be done at any level; it doesn't have to be passed up to MAIN-LOOP like a bucket brigade. When the cycle gets back to MAIN-LOOP, MAIN-LOOP will check the value of P-CONT; if the value is -1, MAIN-LOOP will flush any additional inputs, and go straight back to the ">" prompt.

Under this new and improved theory, TRAP-DOOR-F would look like this:

```
     <ROUTINE TRAP-DOOR-F ()
          <COND (<AND <VERB? OPEN>
                      <NOT <FSET? ,TRAP-DOOR ,OPENBIT>>>
                 <FSET ,TRAP-DOOR ,OPENBIT>
                 <COND (<IN? ,TORCH ,PLAYER>
                        <MOVE ,TORCH ,HERE>
                        <SETG P-CONT -1> ;"flush inputs"
                        <TELL "Your torch slips out of your
     hands as you open the heavy trap door." CR>)
                (T
                 <TELL "Oomph! You open the trap door. It
     sure is heavy!" CR>)>)>>
```

EXERCISE TWO

Using what you've learned so far, design and implement a mini-game, with a small geography of about five interconnected rooms, and about five objects scattered about those rooms. Write action routines for all of the objects, and as many of the rooms as are applicable. Include at least one event. When you're done, get someone to look over your code, and to help you compile it. Then spend some time playing with what you've created. Congratulations! You've taken a big step on the road to Imphood!

Chapter 9: The Syntax File

9.1 Basic Syntaxes

Syntaxes are the writer's way of telling the parser what the legal sentence structures are, and what PRSA a given sentence structure produces. Sentence structures are composed of verbs, noun phrases, and prepositions. Every syntax is associated with an internal-verb, which is what you check for in the predicate VERB? and which will ultimately lead to an associated verb default in the VERBS file if not handled earlier.

Syntaxes are defined in a file called, appropriately, the syntax file. Each syntax is a line in that file that looks something like this:

```
<SYNTAX EAT OBJECT = V-EAT>
```

In this case, EAT is a verb, OBJECT is the point where a noun phrase occurs, and EAT is the resulting PRSA or action. So, if the player typed in any of the following

```
>EAT BIRD
>EAT THE PHEASANT
>EAT THE LIGHTLY SEASONED PHEASANT-UNDER-GLASS
>EAT THE BIRD IN MY HAND
```

the parser would match the input up with that syntax, and return EAT as the PRSA, the PHEASANT object as the PRSO, and <> as the PRSI.

Note that the verb does not have to be the same as the resulting PRSA. For example, you could have a syntax like this:

```
<SYNTAX SLICE OBJECT = V-CUT>
```

The verb is slice, but the PRSA is CUT. The distinction is that "SLICE" is the word from the player's input, but CUT is the internal name for the verb when the input matches this syntax. A few of the most common examples of this:

Input word(s)	Internal name
ATTACK	V-KILL
BREAK	V-MUNG
BLOW OUT	V-EXTINGUISH
ENTER	V-BOARD
GET	V-TAKE
GET IN	V-ENTER
GET OUT	V-EXIT
JUMP	V-LEAP
LEAVE	V-DROP or V-EXIT
LOOK AT	V-EXAMINE
PICK UP	V-TAKE
WALK UP	V-CLIMB

9.2 Prepositions in Syntaxes

Prepositions appear in a syntax definition exactly as they would appear in an input:

```
<SYNTAX LOOK THROUGH OBJECT = V-LOOK-INSIDE>
```

Now, if the player types

```
>LOOK THROUGH THE TELESCOPE
```

the game will set PRSA to LOOK-INSIDE. Now we can see the importance of the syntax definition; if the game had syntaxes like:

```
<SYNTAX LOOK UNDER OBJECT = V-LOOK-UNDER>
<SYNTAX WALK THROUGH OBJECT = V-WALK-THROUGH>
```

but not

```
<SYNTAX LOOK THROUGH OBJECT = V-LOOK-INSIDE>
```

and the player typed

```
>LOOK THROUGH THE TELESCOPE
```

the game would know all the words in the input, but it would not be able to match the sentence structure with any known syntax. The parser would fail, producing a message like "Sorry, but I don't undertand that sentence."

9.3 Syntaxes with Indirect Objects

All the syntax definitions you've looked at so far are for inputs with a PRSO but no PRSI. To do so, simply have two OBJECT spots in the syntax definition:

```
<SYNTAX GIVE OBJECT TO OBJECT = V-GIVE>
```

Now, the input

```
>GIVE THE MAP OF ELBA ISLAND TO UNCLE OTTO
```

would get parsed as PRSA equals GIVE, PRSO equals ELBA-MAP object, and PRSI equals UNCLE-OTTO.

Furthermore, you can have a syntax definition with no OBJECT spot, just a verb:

```
<SYNTAX INVENTORY = V-INVENTORY>
```

Note that you can have many syntaxes which use the same verb, and which can sometimes produce the same PRSA, but which can sometimes produce different PRSAs:

```
<SYNTAX GET OBJECT = V-TAKE>
<SYNTAX GET IN OBJECT = V-ENTER>
<SYNTAX GET ON OBJECT = V-ENTER>
<SYNTAX GET OFF OBJECT = V-EXIT>
<SYNTAX GET OBJECT WITH OBJECT = V-TAKE-WITH>
```

Generally, when you have such a "family" of syntax definitions, you put them in order from the simplest to the most complicated.

9.4 Pre-actions

If you recall from the section 8.2, a pre-action is a routine associated with a verb, and which is given a pretty early opportunity to handle the input. The way to define that a verb has a pre-action is in the syntax file, thusly:

```
<SYNTAX SHOOT OBJECT = V-SHOOT PRE-SHOOT>
```

In the verbs file, there must be two routines for this verb, the verb default routine (called V-SHOOT) and the pre-action routine (called PRE-SHOOT).

Remember that a PRSA can appear in more than one syntax. If it has a pre-action in one of those syntaxes, it must have that same pre-action in all of those syntaxes. For example, this would be illegal:

```
<SYNTAX SHOOT OBJECT = V-SHOOT PRE-SHOOT>
<SYNTAX FIRE AT OBJECT = V-SHOOT>
```

because the first V-SHOOT syntax has a pre-action and the second one doesn't. Likewise, this would be illegal:

```
<SYNTAX SHOOT-OBJECT = V-SHOOT PRE-SHOOT>
```

```
<SYNTAX FIRE AT OBJECT = V-SHOOT PRE-FIRE>
```

because you have two different pre-actions defined for the same PRSA.

9.5 The FIND Feature

You are, of course, familiar with this parser occurence (if you're not, go back to Testing and Lose One Turn):

```
>TAKE
[the jeweled eggplant]
Taken.

>ATTACK MIKE DORNBROOK
[with the marketing budget]
You thrust the marketing budget at Mikey, who turns
pale and dashes away, leaving a Mikey-sized hole in the
conference room wall.
```

Since you're already beginning to think like an Implementor, you're asking yourself "Gee! I wonder how the parser knew to pick the eggplant in the first case and the budget in the second case?" The answer is FIND in the syntax definition. Here's what these two syntaxes might look like:

```
<SYNTAX TAKE OBJECT (FIND TAKEBIT) = V-TAKE>
<SYNTAX ATTACK OBJECT WITH OBJECT (FIND WEAPONBIT)
     = V-ATTACK>
```

If you put FIND in a syntax definition, along with the name of the flag, and the player uses that syntax, but gives incomplete information, the parser will look to see if there is one (and only one) object present which has that flag; if so, it assumes that object! (This is sometimes referred to as GWIM, for "get what I mean.")

If the parser finds zero, or more than one, object present with that flag, it will ask "What do you want to take?" or "What do you want to attack Mike Dornbrook with?" (This feature, where the parser is asking for a little more information to complete an input, is called orphaning.)

There is one unusual use of the FIND feature. Normally, the parser cannot handle a syntax which is just a verb and a preposition, such as LOOK UP or FUCK OFF. However, a kludge has been installed using a flag called the ROOMSBIT:

```
<SYNTAX LOOK UP OBJECT (FIND ROOMSBIT) = V-LOOK-UP>
<SYNTAX FUCK OFF OBJECT (FIND ROOMSBIT) = V-FUCK-OFF>
```

Thanks to this kludge, when the parser gets to this syntax, it will not complain about the lack of an object. Instead, it will set PRSO to the ROOMS object (that special object which is the LOC of all rooms). You can then have V-LOOK-UP or V-FUCK-OFF check whether PRSO is ROOMS. Since there is no way the player can ever refer to the ROOMS object, if ROOMS is the PRSO you can be sure that the player typed LOOK UP without an object. Here's what such a V-LOOK-UP might look like:

```
<ROUTINE V-LOOK-UP ()
     <COND (<PRSO? ,ROOMS>
            <TELL "You stare up at the sky until you
get a stiff neck." CR>)
```

```
            (T
             <TELL "You can look up a chimney; you can
look up a dress; you can look up your uncle in the
phone book; but you can't look up" A ,PRSO "!" CR>)>>
```

9.6 Syntax Tokens

There are several tokens which can appear in parentheses within a syntax definition: HAVE, TAKE, MANY, EVERYWHERE, ADJACENT, HELD, CARRIED, ON-GROUND, and IN-ROOM. This parenthetical list appears after either or both OBJECTs:

```
<SYNTAX GIVE OBJECT (HAVE)
     TO OBJECT (ON-GROUND IN-ROOM) = V-GIVE>
```

HAVE informs the parser that the object in question must be in the player's inventory before the input can be successfully parsed. If the player uses a syntax with a HAVE, and the object in question is not in the player's inventory, the parser will fail with a message like, "You aren't holding the object." For instance, given the GIVE syntax defined above, if the player wasn't holding the pineapple, and said GIVE THE PINEAPPLE TO THE HULA DANCER, the response would be "You aren't holding the pineapple." Note that HAVE is after the first OBJECT; the player doesn't have to have the hula dancer in his or her inventory!

TAKE tells the parser that if the object referred to is not in the player's inventory, but it is takeable, to take it (by moving it to the player's inventory) before returning to let the game handle the input. When the parser does this, it will TELL something like, "[taking the FOO first]". This is referred to as an implicit take, because the player is getting the object without having asked for it. A verb which commonly does an implicit take is READ.

MANY tells the parser that it is okay to allow multiple direct objects with this verb (or multiple indirect objects, if the MANY is placed after the second OBJECT in the syntax). Normally, if the player said EXAMINE SALT AND PEPPER, the parser would fail, saying "[You can't use multiple direct objects with 'examine.']" However, if the syntax definition was:

```
<SYNTAX EXAMINE OBJECT (MANY) = V-EXAMINE>
```

then the parser would allow it. All the items listed in the input will then be examine, preceded by their DESC and a colon, thusly:

```
>EXAMINE SALT AND PEPPER
salt: It is white and crystalline.
pepper: It is powdery, in various shades of brown.
```

EVERYWHERE tells the parser that the object referred to doesn't have to be visible. In these cases, you are telling the parser to look everywhere in the game to match the player's input. For example, you can ASK GEORGE BAILEY ABOUT MR. POTTER even when the MR-POTTER object isn't present; you can FOLLOW SANTA CLAUS after he leaves the room and is no longer referenceable. Therefore these syntaxes would look like this:

```
<SYNTAX FOLLOW OBJECT (EVERYWHERE) = V-FOLLOW>
<SYNTAX ASK OBJECT ABOUT OBJECT (EVERYWHERE)
     = V-ASK-ABOUT>
```

I guess there's another token called ADJACENT, but I have no idea what it does. *[Stu?]*
[The other four tokens—ON-GROUND, IN-ROOM, HELD, and CARRIED—are incredibly confusing, and no one really understands them except Stu, so he should probably write this bit.]

9.7 Verb Synonyms

In much the same way that objects can have synonyms for the nouns and adjectives that are used to refer to them, using the SYNONYM and ADJECTIVE properties, a verb can also have synonyms.
In the syntax file, you would put something like this:

```
<VERB-SYNONYM WORRY FRET AGONIZE>
```

Nearby, you would probably have a syntax definition like:

```
<SYNTAX WORRY ABOUT OBJECT = V-WORRY>
```

With the VERB-SYNONYM list, the player could then type WORRY ABOUT UNCLE OTTO, or FRET ABOUT UNCLE OTTO, or AGONIZE ABOUT UNCLE OTTO, and all would mean the same thing. Note that you could accomplish the same result by having three syntax definitions:

```
<SYNTAX WORRY ABOUT OBJECT = V-WORRY>
<SYNTAX FRET ABOUT OBJECT = V-WORRY>
<SYNTAX AGONIZE ABOUT OBJECT = V-WORRY>
```

However, using the VERB-SYNONYM is more efficient, especially if there are more than one WORRY syntax (WORRY ABOUT OBJECT, WORRY WITH OBJECT, etc.)
Actually, this synonym list can be generalized to all parts of speech, although it is most commonly done with verbs. For example:

```
<PREP-SYNONYM UNDER UNDERNEATH BENEATH BELOW>
<ADJ-SYNONYM LARGE BIG GREAT HUGE>
```

9.8 "Switch" Syntaxes

Frequently, the same action can be worded in two different ways such that the PRSO and PRSI are in the oppposite order in the two constructions. The commonest example: GIVE THE RAZOR BLADE TO THE DUCK and GIVE THE DUCK THE RAZOR BLADE have the same meaning. However, in the first example, the PRSO is the RAZOR-BLADE object and the PRSI is the DUCK object; in the second, PRSO is the duck and PRSI is the razor blade.
This is handled by creating a special second verb for the second syntax. This special verb usually does nothing except switch the order of the PRSO and the PRSI. The naming convention for this "switching" verb is the regular verb with an "S" tacked on the front. Thus:

```
<SYNTAX GIVE OBJECT TO OBJECT = V-GIVE>
<SYNTAX GIVE OBJECT OBJECT = V-SGIVE>
<ROUTINE V-SGIVE ()
     <PERFORM ,V?GIVE ,PRSI ,PRSO>
     <RTRUE>>
```

Chapter 10: Actors

10.1 Definition of an Actor

An actor is simply a character in an interactive fiction story. The term does not include the main character—the player, that is. Some examples of actors are the thief in *Zork I*, Mr. Baxter in *Deadline*, Floyd in *Planetfall*, or Ford Prefect in *Hitchhiker's Guide*.

Creating an actor is very similar to creating any other object. They should have a flag called the PERSONBIT. Also, since actors frequently have items in their possession, they should have the OPENBIT, CONTBIT, and SEARCHBIT so that those possessions will be visible to the player.

10.2 Talking to an Actor

A player talks to an actor by typing something like:

```
>SERGEANT DUFFY, PUT THE HANDCUFFS ON MOE
```

The parser turns an input like this into two inputs; the first one is PRSA equal to the verb TELL and PRSO equal to the DUFFY object; the second one is more conventional, with PRSA equal to PUT-ON, PRSO equal to the HANDCUFFS object, and PRSI equal to MOE.

Eventually, the first PERFORM will probably get to the V-TELL verb default, because of the PRSA in the first input. V-TELL will see that the PRSO is an actor, and that therefore the player is attempting to talk to this actor. It will then do three unusual things: it will set the global variable WINNER to this actor, it will return without outputting anything, and it will tell CLOCKER not to run this turn. Remember, WINNER is usually set to the PROTAGONIST/PLAYER object. However, when you speak to an actor, that actor becomes the WINNER for that turn (or as many turns as you're speaking to them). V-TELL doesn't output anything in this case, and stops time from passing, in order to preserve the appearance that this input was really one input, rather than the two inputs that the parser converted it to.

Remember also, the first opportunity to handle any input is the WINNER's action routine! On the second time through, for the second input, the actor's action routine will get called right away—this is the time and place to handle speaking to the actor!

The first clause in any actor's action routine should check to see whether the actor is the WINNER. For example:

```
<EQUAL? .ARG ,M-WINNER>
```

If true, you know that the player was speaking to DUFFY. Now, within this clause, just handle what was said to the actor:

```
<ROUTINE DUFFY-F ("OPT" ARG)
     <COND (<EQUAL? .ARG ,M-WINNER>
            <COND (<AND <VERB? ,PUT-ON>
                        <PRSO? ,HANDCUFFS>>
            <TELL "Sgt. Duffy arrests " T ,PRSI "." CR>)
           (<VERB? ANALYZE>
            <REMOVE ,DUFFY>
            <QUEUE I-DUFFY-RETURNS 20>
```

```
                <TELL "Duffy exits, saying \"I'll have to
     take this to the lab.\"" CR>)
               (T
                <TELL "\"Sorry, detective. My expertise
     extends only to police work.\"" CR>)>)>>
```

In most games, the actor's WINNER clause must handle everything said to that actor, thus the "Sorry, detective..." line at the end. Without such a catch-all, the output would be handled as though the input were typed to the game, rather than said to a character. For example, the DUFFY-F above has special responses for only two things that might be said to Duffy:

```
>DUFFY, PUT THE HANDCUFFS ON [someone]
```

and

```
>DUFFY, ANALYZE [something]
```

Suppose that the "Sorry, detective..." clause weren't there to catch anything else said to Duffy; the following could happen:

```
>DUFFY, HIT MRS. ROBNER
You slap Mrs. Robner. Furious, she throws you out of
the house.
```

Notice how the response matches an input of HIT MRS. ROBNER rather than DUFFY, HIT MRS. ROBNER.

10.3 The Transit System

Chapter 11: The Describers

11.1 Definition

There's a small package of programs which handle the descriptions for the player's environment: current room and visible objects. These routines are called the describers.

The describers are used every time a room description is needed, either because the player has entered a new location or because the player has typed LOOK. The describers are also called by a number of other verbs, such as INVENTORY or LOOK-INSIDE.

The exact details of how the describers do their job vary from game to game. For instance, back in the old days, objects were always printed out in a "laundry list" style:

```
>OPEN THE BOX
Opening the box reveals:
     a Monopoly board
     a red hotel
     a green house
     a dog token
     a fifty-dollar bill
     a Park Place deed
```

More recently, many games have chosen to list objects in paragraph form:

```
>OPEN THE BOX
Opening the box reveals a Monopoly board, a red hotel,
a green house, a dog token, a fifty-dollar bill, and a
Park Place deed.
```

11.2 What goes on during a LOOK

When the player types LOOK, the game should give him a full description of his surroundings. This should also happen the first time the player enters a room in brief mode, or any time the player enters a room in verbose mode.

In the case of LOOK, the V-LOOK verb default handles the input by calling two routines, one which describes the room, and another which describes the objects in the room. These routines are called DESCRIBE-ROOM and DESCRIBE-OBJECTS (in some games, D-ROOM and D-OBJECTS). In the case of the player entering a room, these two routines are called by the GOTO routine which handles player movement.

11.3 DESCRIBE-ROOM

There are two parts to a room description: the name of the room and the sentence(s) describing it. The room name (the room object's DESC property) gets printed every time the player enters a room. The descriptive text should get printed only if the player is in verbose mode, or if the player is in brief mode and is entering the room for the first time. Both parts should be given when the player types LOOK.

The first thing the DESCRIBE-ROOM does is to figure out whether the area is lit or dark. If it's dark, it TELLs a string such as "It's too dark to see." and returns without doing anything else.
If the player's location is lit, DESCRIBE-ROOM then TELLs the room's DESC:

```
Shark Tank
```

Some games put the room name in bold type (see the graphics section about the HLIGHT command):

Shark Tank

Some games tack on an addendum to the room name if the player is in a vehicle:

```
Shark Tank, in the shark cage
```

Next, DESCRIBE-ROOM decides whether the second part of the room description, the descriptive text, is warranted—depending on the briefness mode, whether the room has been visited before (is the TOUCHBIT of the room set?), and whether the player is doing a LOOK.
If the full description is warranted, DESCRIBE-ROOM must then figure out where to get this description. The first place it checks is to see if the room has an LDESC property. If it does, DESCRIBE-ROOM then TELLs this string.
If a room has a description that changes over the course of game play, such as a door which opens or closes or a wall which collapses into rubble, then it must be described by the room's action routine. Therefore, when DESCRIBE-ROOM finds no LDESC property, it next calls the room's action routine with the argument M-LOOK. The action routine should have an <EQUAL? .RARG ,M-LOOK> predicate, and within this clause it should TELL the room description. Except in very unusual cases, all rooms must have either an LDESC or an M-LOOK clause. DESCRIBE-ROOM has now completed its work.

11.4 DESCRIBE-OBJECTS

The objects in a room should get described anytime the player does a LOOK and anytime the player enters a room, except in SUPERBRIEF mode. The exception is if DESCRIBE-ROOM has decided that it is dark, and TELLed an appropriate string. It then returns false, informing V-LOOK or GOTO not to bother calling DESCRIBE-OBJECTS.
DESCRIBE-OBJECTS is quite a bit more complicated than DESCRIBE-ROOM, because of all the different ways that objects can be described: FDESC, LDESC, DESCFCN, or default description. And that's not even including containment issues.
Objects may be described several ways. One way is the default; don't do anything special to describe the object. Instead, the object's DESC is used to fill in a default description. For example, if you created a bicycle horn with no special description, and with a DESC of "bicycle horn," then the player would enter a room with the horn in it and see:

```
Bicycle Store
     This huge shop is filled with bicycles of every
description. One wall is covered with a pegboard of
parts and accessories.
     You can see a bicycle horn here.
```

Let's say you wanted to give the horn a more interesting description. You could give the horn object an LDESC property:

```
(LDESC "A shiny brass horn is lying on the ground.")
```

The LDESC will be used by DESCRIBE-OBJECTS whenever the horn is on the ground in the player's room. In other words, instead of the above vanilla description, the player would see:

```
Bicycle Store
    This huge shop is filled with bicycles of every
description. One wall is covered with a pegboard of
parts and accessories.
     A shiny brass horn is lying on the ground.
```

The horn's DESC would still be used for verb defaults, such as "Shaking the bicycle horn isn't very helpful."

Another option is to give the horn an FDESC (as in First DESCription):

```
(FDESC "One of the items on the pegboard wall is a
shiny brass horn. It almost seems to be calling to you,
begging to be mounted on a handlebar.")
```

The FDESC will be used by DESCRIBE-OBJECTS as long as the horn is in its original state. If the player takes the horn, its TOUCHBIT will be set, and the FDESC will no longer be used. Instead, the LDESC will be used, if the horn has one, else the DESC/default. (Note that an object can have both an FDESC and an LDESC.) The player would now see:

```
Bicycle Store
    This huge shop is filled with bicycles of every
description. One wall is covered with a pegboard of
parts and accessories.
     One of the items on the pegboard wall is a shiny
brass horn. It almost seems to be calling to you,
begging to be mounted on a handlebar.
```

The reason for the FDESC should be apparent. If the player picks up the horn, then drops it in the VEGETABLE-STAND room, you don't want a LOOK to give the "pegboard wall" description—the pegboard wall isn't even here (in VEGETABLE-STAND)!

Note that there's no reason for giving an FDESC to an untakeable object; its TOUCHBIT will never get set, and the FDESC will be used throughout the play of the game. Instead, use an LDESC to give an untakeable object a non-vanilla description.

The last describer option is the DESCFCN. It will be described in detail in the next section.

DESCRIBE-OBJECTS makes three passes through all the objects in a room, in order to describe them in a certain order. First, objects with DESCFCNs and FDESCs are described. Then, any object with an LDESC is described. Finally, all the remaining objects are described using their DESCs:

```
Amazing Describer Room
     This is a dimly-lit room filled with arcane
objects and complex routines.
     Sitting on a shelf is an object with an FDESC. It
hasn't been touched yet.
```

```
     An object with an LDESC is lying in a discarded
heap on the ground.
     You can see a drab object, a boring object, and a
dismally uninteresting object here.
```

The contents of objects are described immediately after the object itself:

```
Amazing Describer Room
     A box with an LDESC has been discarded here. It
seems that the box contains a foo and a bar.
     You can see a drab object, a boring object, and a
dismally uninteresting object here. It seems that the
boring object contains a bletch.
```

11.5 DESCFCNs

A DESCFCN is the most complex, but most powerful, way to describe an object. Basically, it means creating a function whose purpose is to describe the object in any given situation.

The first step is to give the object the DESCFCN property, indicating the name of the routine which will handing the describing:

```
(DESCFCN HORN-DESC-F)
```

Next, write the routine:

```
<ROUTINE HORN-DESC-F (ARG)
     <COND (<EQUAL? .RARG ,M-OBJDESC?>
            <RTRUE>)
;"subsequent clauses must be due to an M-OBJDESC call"
           (,HORN-MOUNTED
            <TELL " A brass bike horn is mounted on the
bicycle handlebars.">
            <COND (<EQUAL? ,HERE ,MAGIC-BIKEPATH>
                   <TELL " The horn is glowing with a
gentle yellow light.">)>
            <CRLF>)
           (T
            <TELL " A brass bicycle horn is lying here.
You can almost here it saying, /"Mount me on a pair of
handlebars!/"" CR>)>>
```

Notice how this routine handles several different cases: whether the horn is mounted on the handlebars or not, and if so, whether it is glowing due to the presence of the Magic Bikepath.

Also note the spaces at the beginning of the TELLs in the DESCFCN. If your describers are of the indentation flavor, your DESCFCN must supply its own indentation.

Finally, notice the first clause in the COND. The describers call the DESCFCN twice. The first time is just to ask whether the DESCFCN will be describing the object at the present time. At this point, the DESCFCN is called with the argument M-OBJDESC?. The second time is to tell the DESCFCN to go ahead and do the describing. In this case, the DESCFCN is called with M-OBJDESC. (The difference in argument names, for maximal confusion, is simply the terminating question mark.) In the case of HORN-DESC-F, the DESCFCN

describes the horn in all cases. However, if you wrote HORN-DESC-F to describe the horn only when mounted on the handlebars, the M-OBJDESC? clause would have to RFALSE whenever the horn wasn't mounted, telling the describers, "Go ahead and describe the horn; I'm not planning on describing the horn under current conditions." The routine would then look like this:

```
<ROUTINE HORN-DESC-F (ARG)
     <COND (<EQUAL? .RARG ,M-OBJDESC?>
            <COND (,HORN-MOUNTED
                   <RTRUE> ;"I'll describe the horn")
                  (T
                   <RFALSE> ;"you describe the horn")>)
;"subsequent clauses must be due to an M-OBJDESC call"
           (,HORN-MOUNTED
            <TELL " A brass bike horn is mounted on the
bicycle handlebars.">
            <COND (<EQUAL? ,HERE ,MAGIC-BIKEPATH>
                   <TELL " The horn is glowing with a
gentle yellow light.">)>
            <CRLF>)
           (T
            <RFALSE>)>>
```

11.6 The Useful but Dangerous NDESCBIT

Normally, all visible objects will appear when the describers provide a room description. If you don't want a particular object described, you must give it a flag called the NDESCBIT. This flag simply tells the describers to skip over that object.

The most common use of an NDESCBIT is for an object which is already described in the room description. For example, if you had a room with a water fountain, and the water fountain was mentioned in the room's LDESC, or its M-LOOK clause, but the water fountain object didn't have the NDESCBIT, this would happen:

```
Hallway
    You are in a short hallway between the kindergarten
classrooms to the north and the principal's office to
the south. A water fountain is nestled in a shallow
alcove, its cooling system humming quietly.
    You can see a water fountain here.
```

You might also give the NDESCBIT to a takeable object (as opposed to a permanent feature of a room) provided that the takeable object is described by the room at first. If you do this, however, you must make sure that the room stops describing the object once it is gone, and that the NDESCBIT is cleared any time the object is moved. (For example, you should give such an object the TRYTAKEBIT to prevent it from being acquired by an implicit take.)

Chapter 12: Some Complicated Stuff That You Don't Want to Learn But Have To

12.1 Loops

The way that ZIL code loops back on itself is through a device called a REPEAT. A REPEAT lives inside a routine, and when the routine reaches and enters the REPEAT, it continues to run through all the expressions in the REPEAT until ordered to leave it using RETURN. A REPEAT must always have its own argument list. The word "AUX" is implicitly at the beginning of a REPEAT's argument list; any variable in it is an auxiliary, but you don't need to explicitly include the "AUX." Here's an example of a simple repeat:

```
<SET CNT 0>
<REPEAT ()
     <TELL "Ha">
     <SET CNT <+ .CNT 1>>
     <COND (<EQUAL? .CNT 5>
            <TELL "!">
            <RETURN>)
           (T
            <TELL " ">)>>
```

This repeat would cause the game to print "HA HA HA HA HA!" Note the RETURN, and where it occurs logically. Also note the empty argument list. You could substitute for the <SET CNT 0> by having an argument list that says ((CNT 0)), but don't worry if you don't understand that.

FIRST? and NEXT? are often used along with REPEAT. For instance, here's a call to a routine, and the routine itself, that would remove everything from a given container:

```
<EMPTY-CONTAINER ,BASKET>
<ROUTINE EMPTY-CONTAINER (OBJ "AUX" X N)
     <SET X <FIRST? .OBJ>>
     <REPEAT ()
          <COND (.X
                 <SET N <NEXT? .X>>
                 <MOVE .X ,HERE>
                 <SET X .N>)
                (T
                 <RETURN>)>>
     <TELL "You completely empty" T .OBJ "." CR>>
```

Let's say that the BASKET contained TOTO and a pair of RUBY-SLIPPERS. In the first line of the routine, the local variable X is set to the FIRST? object in the basket, TOTO. (If there was nothing in the basket, X would be set to <> at this point.)

The routine then enters the REPEAT; the first thing the repeat does is a COND, which checks whether X is true; that is, whether it is set to an object rather than to <>. Since X is equal to TOTO, the predicate is true, and the local variable N is set to the NEXT? object in the basket, the RUBY-SLIPPERS. (Remember,

RUBY-SLIPPERS is the NEXT? of the FIRST? object, TOTO, not the NEXT? of the basket itself.) After that, TOTO is moved to HERE. This is why N is used; once TOTO is moved to HERE, it is no longer in the basket, and RUBY-SLIPPERS is no longer NEXT? of TOTO.
After the MOVE, X is set to N, the RUBY-SLIPPERS. The COND is now done, and since there's nothing else in this REPEAT, you are now at the bottom of the REPEAT, and jump back to the top of the REPEAT. X, now the RUBY-SLIPPERS, is true. But NEXT? of the RUBY-SLIPPERS is <>, since there's nothing else in the basket, so N is set to <>. The RUBY-SLIPPERS are moved to HERE, and X is set to N, meaning that it is set to <>.
Once again we return to the top of the REPEAT. This time, the predicate is false, so we go instead to the second clause of the COND. Since X was false, the basket has been completely emptied, and we RETURN from the REPEAT. Finally, the routine does a TELL before finishing its task.
At any point in a REPEAT, you can return to the top of the REPEAT by invoking <AGAIN>. <AGAIN>, when used outside of a REPEAT, simply sends you back to the top of the current routine.

12.2 Property Manipulation

Properties, such as LDESC and SIZE, aren't just static storehouses of information that you set in stone when you define a room or object. They can be changed on the fly, just like flags or global variables.
The way to get at the information contained in a particular object's property is with the GETP command (as in GET Property). You must supply the name of the object and the name of the property (prefaced by ,P?). Here are some examples:

```
<GETP ,HERE ,P?DOWN>
<GETP ,OMARS-TENT ,P?LDESC>
<GETP ,PRSO ,P?SIZE>
```

Here's how each of those GETPs might appear in context:

```
<COND (<GETP ,HERE ,P?DOWN>
       <TELL "A stair leads to a deeper part of the
maze." CR>)>

<TELL "The camel lumbers into the tent; you dismount."
CR CR>
<MOVE ,CAMEL ,OMARS-TENT>
<MOVE ,PLAYER ,OMARS-TENT>
<TELL "Omar's Tent" CR <GETP ,OMARS-TENT ,P?LDESC>>

<SET FODDER-SIZE <GETP ,PRSO ,P?SIZE>>
<COND (<G? .FODDER-SIZE 10>
       <TELL "The " D ,PRSO " doesn't fit into the
cannon." CR>)
      (T
       <MOVE ,PRSO ,CANNON>
       <TELL "The " D ,PRSO " falls into the cannon's
barrel." CR>)>
```

The way you change the information in a property is using the PUTP command (as in PUT Property). This is just like GETP, except that you also have to supply the value that you want to put into that property:

```
<PUT ,OMARS-TENT ,P?LDESC "This once-fine tent is now
ruined -- tent poles knocked down, canvas sagging, the
floor covered with camel excrement, and a foul camel
odor pervading every corner.">
<PUTP ,PRSO ,P?SIZE 1>
<PUTP ,PROTAGONIST ,P?ACTION ,HYPNO-CASE-F>
```

In practice, these PUTPs might look like this:

```
<MOVE ,PLAYER ,OMARS-TENT>
<MOVE ,CAMEL ,OMARS-TENT>
<PUT ,OMARS-TENT ,P?LDESC "This once-fine tent is now
ruined -- tent poles knocked down, canvas sagging, the
floor covered with camel excrement, and a foul camel
odor pervading every corner.">

<MOVE ,PRSO ,MINIATURIZER>
<PUTP ,PRSO ,P?SIZE 1>
<TELL "As you place" D ,PRSO " in the miniaturizer, it
shrinks to a fraction of its former size." CR>

<TELL "You stare at the doctor's swinging stopwatch,
and feel yourself falling into a trance...">
<PUTP ,PROTAGONIST ,P?ACTION ,HYPNO-CASE-F>
```

12.3 Tables

A table is just a tool for storing information, like a global variable. Unlike a global variable, it can store more than one piece of information. Here's what a simple table might look like:

```
<CONSTANT MAZE-TABLE
 <TABLE 12 18 24 0 0 0>>
```

This table has six elements. I'm sure you'll find it intuitively obvious that the first element is element number 0, and that the last element is therefore element number 5. In other words, MAZE-TABLE is currently storing the number 12 in its zeroth slot, the number 18 in its first slot, etc.

The way to get information from a table is with the GET command:

```
<GET ,MAZE-EXITS 0>
<GET ,MAZE-EXITS 4>
<GET ,MAZE-EXITS .MAZE-ROOM-NUM>
```

The first of these GETs will return element number 0 from the table, which is 12. The second GET returns element number 4, which is 0. The value returned by the last GET will depend on the value of the local variable MAZE-ROOM-NUM.

The way to put information into a table is with PUT:

```
<PUT ,MAZE-EXITS 3 99>
```

After executing this PUT, MAZE-EXITS will now look like:

```
<CONSTANT MAZE-TABLE
 <TABLE 12 18 24 99 0 0>>
```

Elements in a table can be almost anything: numbers, strings, names of objects, and so on.
There are several kinds of tables: PTABLEs are stored outside of the game's pre-load; LTABLEs tell the compiler to create a 0th element whose value is the number of elements in the table; PLTABLEs have both features. You probably won't need to understand any of this until you're well into your first game.

12.4 Generics

[Stu, I think that you should write this section.]

12.5 Other Information You Can Obtain from the Parser

[Stu, I think that you should write this section.]

Chapter 13: New Kids on the Block -- Graphics and Sound

13.1 The Basic Organization

Up to now, everything you've written will end up in a single file, usually referred to as the game file. In a game with graphics, however, all the graphics will live in a separate file, usually referred to as the picture file. The picture file can be thought of as a collection of graphics which the game can reach into and grab pictures from at any time, using the DISPLAY instruction.

The game file, at least in theory, will be identical from version to version. The Apple II, Amiga, and IBM versions should all have the same game file. The information in the game file is said to be machine independent. However, since the graphics capabilities of these machines vary so widely, they will all have a different version of the picture file. In fact, a given version might have more than one picture file. The Mac version of the game will probably have two: a color picture file for Mac IIs, and a black-and-white picture file for all other Macs. The IBM version might have three picture files for the various IBM graphics configurations.

In addition to pictures, a picture file also contains something called invisible pictures. These are basically just a pair of numbers, an X-value and a Y-value, and are used to decide where to display a picture. Since this information is machine-dependent, it must live in the picture file so that it can be tailored to each machine version.

13.2 The DISPLAY Command

To put a graphic up on the screen, you must use the DISPLAY command. DISPLAY is supplied with a picture and with a point on the screen; it then puts the picture up on the screen with its upper-left corner at the supplied point. DISPLAY takes three arguments: the picture to be displayed, and the Y and X co-ordinates of the point on the screen. To guarantee excrutiating confusion, the Y-value comes before the X-value. Example:

```
<DISPLAY ,P-LIV-RM 12 1>
```

This will display P-LIV-ROOM, which is presumably a picture of the Living Room, at Y=12 and X=1.

In almost every case, however, a picture is displayed at a different point depending on the machine version. This is the purpose of those invisible pictures. Before displaying the picture, you must get the X and Y info from the appropriated invisible picture using the PICINF command (for PICture INFormation). PICINF takes two arguments, the name of the invisible picture, and the name of a table to put the information in. There is a table called PICINF-TBL for this purpose:

```
<PICINF ,LR-LOC ,PICINF-TBL>
```

The Y value of LR-LOC is now stored in the 0 slot of PICINF-TBL, and the X value is now stored in the 1 slot.

Now, you must add 1 to each value. The reason is too complicated to go into here. (This is only the case if you are displaying a picture relative to the top left corner of the screen. If you are displaying a picture relative to some previously

displayed picture, or some previously determined point on the screen, then don't add 1 to each value.)
Therefore, the DISPLAY might look like this:

```
<PICINF ,LR-LOC ,PICINF-TBL>
<SET Y <+ <GET ,PICINF-TBL 0> 1>>
<SET X <+ <GET ,PICINF-TBL 1> 1>>
<DISPLAY ,P-LIV-ROOM .Y .X>
.c2.13.3
```

13.3 Sound and Music

[Beats the heck out of me! I've never done diddly-squat with sound!]

Chapter 14: Organizing Your ZIL Files

14.1 History and Theory

In the distant misty reaches of time, games were organized into two files. One was called ACTIONS.ZIL and contained all the action routines and other associated routines; the other was called DUNGEON.ZIL and contained all the room and object definitions. (As you can tell from the latter name, all the early games were some flavor of *Zork*).

Nowadays, files of game source code can be divided up almost any way you please. In theory, you could have all the code in one single giant humongo quivering file. However, there are many reasons to divide it up into smaller chunks. For example, to make a new version of the game, you need only compile those file which have changed; if your game is divided into multiple source files, less code will need to be compiled, and the compilation will take less time. It also makes it easier to tell someone where to find a particular piece of your code. Also, if you need to make a printout of a part of the source code, you can print a more reasonably-sized section. Etc.

There's no exact answer to the question of what's the right size for a ZIL file. You don't want them to be too large, because of the reasons already mentioned. On the other hand, you don't want to have too many little files, because your directory will start to look like Oscar Madison's bedroom. A good rule of thumb is around 10 or 12 files (not including the parser). If, as you're writing the game, a particular file starts getting too bloated, simply break it up into two smaller files.

Although periods in the middle of filenames have no special meaning to Spike, you should still name all your source code files [SOMETHING].ZIL. This will indicate to you, and anyone else looking through the game directory, exactly what files are the source code for the game.

There are three flavors of ZIL files. The following sections describe them, along with the conventions for dividing games and for naming ZIL files.

14.2 The Parser

The parser is a black box of ZIL files. They don't live in the directory with the other game files; everyone shares the same set of parser files, which live in a separate parser directory. And you can't touch the parser files. Ever.

14.3 The Substrate

The substrate refers to the basic core of objects, routines, and syntaxes which are common to almost every game.

When you begin writing a game, you (or your mentor) will take a recent game and "strip away" everything that is special to that game, leaving you with a vanilla shell. This shell is the substrate, upon which you begin writing your new game.

Here's a recap of the ZIL files which would commonly compose the substrate of a recent ZIL game:

The SYNTAX.ZIL file contains several hundred syntax definitions, along with verb and preposition synonym lists.

The VERBS.ZIL file contains all the verb default routines for the verbs defined in the syntax file. Also, it usually contains a bunch of utility routines, such as the describers, JIGS-UP, etc.
The GLOBALS.ZIL file contains the object definitions (and associated action routines) for those global and local-global objects which appear in almost every game: PROTAGONIST, GROUND, HANDS, STAIRS, etc.
The HINTS.ZIL file contains the vast tables of strings which make up your on-line hints. The code for the on-line hints is in a file called CLUES.ZIL which, like the parser files, doesn't live in the game's directory. It lives in a directory called ZILLIB (for ZIL LIBrary).
The INPUT.ZIL file is where all the code associated with reading the player's input lives. A lot of this file is devoted to the code for user-definable function keys.
The MISC.ZIL file contains a whole bunch of random substrate stuff that doesn't belong anywhere else. These include macros, which are like routines, only different. Few understand them. TELL is an example of a macro. The misc file is also the home of CLOCKER and other interrupt-related routines.

14.4 Your Game Files

The rest of the ZIL files are the stuff which makes your game, well, your game—and not *Zork III* or *Plundered Hearts*. How you organize these files is up to you, but here are some common ways:
You can divide the game up geographically. For example, *Planetfall* has two files, one for each of the two island complexes. (The stuff aboard the ship, at the beginning of the game, lives in the GLOBALS file.) For another example, *Leather Goddesses* divided the game up between files for Earth, Phobos, Mars, Venus, Outer Space, and Cleveland.
Most of the mysteries are divided up into files called People, Places, and Things. The first contains actors and their associated code; the second, rooms; and the last contains all the remaining objects.
If your game tends to divide into "scenes," that would be a good way to divide the code. Any one element of the game with a lot of code can be put in a separate file: for example, the file MAGIC.ZIL in *Enchanter* contains all the code for learning and casting spells.

Chapter 15: Fireworks Time -- Compiling Your Game

Chapter 16: Using ZIL for Other Types of Games

EXERCISE THREE
Design and implement a full-size game. Submit it to testing, fix all the resulting bugs, help marketing design a package, ship the game, and sell at least 250,000 units.

Appendix A: Properties

Properties are what make up the object definitions for objects and rooms. The information in a property can be gotten using GETP (or GETPT) and changed using PUTP (or PUTPT).
This is a list of those object properties which appear in many games. Additional properties can be created for your game if you need them. There is a limit of 64 properties in YZIP.
NORTH, **SOUTH**, **EAST**, **WEST**, **NE**, **SE**, **NW**, **SW**: These are the direction properties, generally used only in room definitions. For the various types of direction properties, see section 2.2. Note that the cardinal direction properties are not abbreviated, but that the non-cardinal ones are abbreviated. There is no direction property called NORTHEAST, for example.
UP, **DOWN**: These are just like the eight direction properties.
IN, **OUT**: These are just like the eight direction properties. If the player just types IN or OUT, this property will handle the movement. Generally, it's a good idea to give the OUT property to any room with only one exit.
SYNONYM: Contains a list of the nouns which can be used to refer to the object.
ADJECTIVE: Contains a list of the adjectives which can be used to refer to the object.
ACTION: Defines the action routine associated with the object. In the case of an object, the action routine is called when the object is the PRSO or the PRSI of the player's input. In the case of a room, the routine is called with M-BEG and M-END once each turn, with M-ENTER whenever the room is entered, and with M-LOOK whenever the describers need to describe the room.
DESCFCN: Defines the routine which the describers use to describe the object. This can be the same routine as the object's action routine, provided that the routine is set up to handle the optional variable (M-OBJDESC or M-OBJDESC?). See section 11.5.
CONTFCN: I never use this, why should you?
GENERIC: Defines the routine which handles cases where the parser determines an ambiguity about which object the player is referring to. In the absence of a generic property, the parser will simply ask "Which FOO do you mean..."
DESC: Technically, this isn't a property, but it looks just like one when you define an object. It contains the string which, in the case of objects,will be used in verb defaults, player's inventory, etc. In the case of rooms, it is the room name which appears before room description and on the status line.
SDESC: Using this property is the only way to give an object a changable DESC. You can't

```
<PUTP .OBJECT ,P?DESC "new desc">
```

but you can

```
<PUTP .OBJECT ,P?SDESC "new desc">
```

Be warned, however, that if your game "shell" isn't set up for SDESCs, you will have to change every verb default. Also, be warned that doing this will increase the size of your game by hundreds of bytes or more, since the verb defaults will

no longer simply TELL the desc of the object, but must instead call a little routine which decides whether the object in question has an SDESC or not.
LDESC: In the case of a room, this contains a string which the describers use for the long description of the room. In the case of an object, this contains a string which the describers use to describe the object if it is on the ground.
FDESC: This property, which isn't usually used in room definitions, contains a string which the describers use to describe the object before the first time it is moved.
LOC: Once again, technically not a property, but it looks just like one when you're creating an object. Simply, this property contains the name of the object which contains this object (in the case of a room, this is the object ROOMS).
SIZE: Contains a number which is the size/weight of the object. Generally, it is only meaningful for a takeable object. If a takeable object has no size property, the game usually gives it a default size of 5. The size of an object affects the number of object that a player can carry, how much of a container it takes up, and so on.
CAPACITY: Contains a number which is the capacity of the object. Generally, it is only meaningful for a container. If a container has no size property, the game usually gives it a default capacity of 5. The capacity of a container affects the number of objects which can be placed inside it.
VALUE: This property is used in many games that have scoring. The property contains a number; in the case of rooms, it is the number of points the player gets for entering the room for the first time; in the case of objects, it is the number of points the player gets for picking up the object for the first time.
GLOBAL: Generally found only in room definitions, this property contains a list of objects which are local-globals referencable in that room.
OWNER: Defines an object which is the owner of this object. For example, the SPORTS-CAR object might have the property

```
(OWNER CYBIL)
```

so that the player could refer to the car as "Cybil's car" even though Cybil isn't actually holding the car. When Cybil sells the car to the player, you would

```
<PUTP ,SPORTS-CAR ,P?OWNER ,PROTAGONIST>
```

so that the player could now refer to it as "my car."
TEXT: This property contains a string which is used when the player tries to read the object. It exists for those objects which would otherwise need an action routine to handle READ but nothing else.
THINGS: Formerly known as the PSEUDO property, this property allows you to create "pseudo-objects" with some of the properties of real objects. They have three parts: a list of adjectives, a list of nouns, and an action routine. Here's an example:

```
(THINGS  (RED CARMINE) (SCARF ASCOT) RED-SCARF-F)
```

Pseudo objects are very limited, however. They cannot have flags, and they cannot be moved. It is beneficial to use them whenever feasible, because (unlike real objects) they take up no pre-load space.
ADJACENT: Something to do with adjacent rooms and referencability. *Stu?*
PLURAL: *Stu?*

PICTURE: Contains the name of a graphic from the picture file associated with the room or object.
FLAGS: This is another fellow which looks just like a property but isn't actually a property. It contains a list of all the flags which are FSET in that object at the start of the game. A list of the common flags can be found in the next appendix.

Appendix B: Flags

Flags are the method for keeping track of the characteristics of an object or room. The starting characteristics of an object are defined in the object's FLAGS property. A flag can be set using FSET, cleared using FCLEAR, and checked using FSET?
This is a list of flags which appear in many games. Additional flags can be added to your game if you need them. There is a limit of 48 *[right?]* flags in YZIP.
TAKEBIT: One of the most basic bits, this means that the player can pick up and carry the object.
TRYTAKEBIT: This bit tells the parser not to let the player implicitly take an object, as in:

```
>READ DECREE
[taking the decree first]
```

This is important if the object has a value and must be scored, or if the object has an NDESCBIT which must be cleared, or if you want taking the object to set a flag or queue a routine, or...
CONTBIT: The object is a container; things can be put inside it, it can be opened and closed, etc.
DOORBIT: The object is a door and various routines, such as V-OPEN, should treat it as such.
OPENBIT: The object is a door or container, and is open.
SURFACEBIT: The object is a surface, such as a table, desk, countertop, etc. Any object with the surfacebit should also have the CONTBIT (since you can put things on the surface) and the OPENBIT (since you can't close a countertop as you can a box).
LOCKEDBIT: Tells routines like V-OPEN that an object or door is locked and can't be opened without proper equipment.
WEARBIT: The object can be worn. Given to garments and wearable equipment such as jewelry or a diving helmet. Only means that the object is wearable, not that it is actually being worn.
WORNBIT: This means that a wearable object is currently being worn.
READBIT: The object is readable. Any object with a TEXT property should have the READBIT.
LIGHTBIT: The object is capable of being turned on and off, like the old brass lantern from *Zork*. However, it doesn't mean that the object is actually on.
ONBIT: In the case of a room, this means that the room is lit. If your game takes place during the day, any outdoor room should have the ONBIT. In the case of an object, this means that the object is providing light. An object with the ONBIT should also have the LIGHTBIT.
FLAMEBIT: This means that the object is a source of fire. An object with the FLAMEBIT should also have the ONBIT (since it is providing light) and the LIGHTBIT (since it can be extinguished).
BURNBIT: The object is burnable. Generally, most takeable objects which are made out of wood or paper should have the BURNBIT.

TRANSBIT: The object is transparent; objects inside it can be seen even if it is closed.
NDESCBIT: The object shouldn't be described by the describers. This usually means that someone else, such as the room description, is describing the object. Any takeable object, once taken, should have its NDESCBIT cleared.
INVISIBLE: One of the few bits that doesn't end in "-BIT," INVISIBLE tells the parser not to find this object. Usually, the intention is to clear the invisible at some point. For example, you might clear the invisible bit on the BLOOD-STAIN object after the player examines the bludgeon. Until that point, referring to the blood stain would get a response like "You can't see any blood stain right here."
TOUCHBIT: In the case of a room, this means that the player has been to the room at least once. Obviously, no room should be defined with a TOUCHBIT, since at the beginning of the game, the player has not been in any room yet. In the case of an object, this means that the object has been taken or otherwise disturbed by the player; for example, once the TOUCHBIT of an object is set, if it has an FDESC, that FDESC will no longer be used to describe it.
SEARCHBIT: A very slippery concept. It tells the parser to look as deeply into a container as it can in order to find the referenced object. Without the SEARCHBIT, the parser will only look down two-levels. Example. There's a box on the ground; there's a bowl in the box; there's an apple in the bowl.
If the player says TAKE APPLE, and the box or the bowl have a SEARCHBIT, the apple will be found by the parser and then taken. If the player says TAKE APPLE, and the box and bowl don't have the SEARCHBIT, the parser will say "You can't see any apple right here." Frankly, I think the SEARCHBIT is a stupid concept, and I automatically give the SEARCHBIT to all containers.
VEHBIT: This means that the object is a vehicle, and can be entered or boarded by the player. All objects with the VEHBIT should usually have the CONTBIT and the OPENBIT.
PERSONBIT: This means that the object is a character in the game, and such act accordingly. For example, they can be spoken to. This flag is sometimes called the ACTORBIT.
FEMALEBIT: The object is an ACTOR who is a female. Informs various routines to say "she" instead of "he."
VOWELBIT: The object's DESC begins with a vowel; any verb default which prints an indefinite article before the DESC is warned to use "an" instead of "a."
NARTICLEBIT: The object's DESC doesn't not work with articles, and they should be omitted. An example is the ME object, which usually has the DESC "you." A verb default should say "It smells just like you." rather than "It smells just like a you."
PLURALBIT: The object's DESC is a plural noun or noun phrase, such as "barking dogs," and routines which use the DESC should act accordingly.
RLANDBIT: Usually used only for rooms, this bit lets any routine that cares know that the room is dry land (as most are).
RWATERBIT: The room is water rather than dry land, such as the River and Reservoir in *Zork I*. Some typical implications: The player can't go there without a boat; anyone dropped outside of the boat will sink and be lost, etc.

RAIRBIT: The room is in mid-air, for those games with some type of flying.
KLUDGEBIT: This bit is used only in the syntax file. It is used for those syntaxes which want to be simply VERB PREPOSITION with no object. Put (FIND KLUDGEBIT) after the object. The parser, rather than complaining about the missing noun, will see the FIND KLUDGEBIT and set the PRSO (or PRSI as the case may be) to the ROOMS object. Some games use RLANDBIT instead of the KLUDGEBIT; this saves a bit, since the parser won't "find" a room, and no objects have the RLANDBIT.
OUTSIDEBIT: Used in rooms to classify the room as an outdoors room.
INTEGRALBIT: This means that the object is an integral part of some other object, and can't be independently taken or dropped. An example might be a dial or button on a (takeable) piece of equipment.
PARTBIT: The object is a body part: the HANDS object, for example.
NALLBIT: This has something to do with telling a TAKE ALL not to take something, but I don't recall how it works. Help???
DROPBIT: Found in vehicles, this not-very-important flag means that if the player drops something while in that vehicle, the object should stay in the vehicle rather than falling to the floor of the room itself.
INBIT: Another not-too-important vehicle-related flag, it tells various routines to say "in the vehicle" rather than "on the vehicle."

Appendix C: Common Routines

This is a list of useful routines which you will find in many, and in some cases all, games. If you're not sure whether your substrate already has one of these routines, just do a hunt through all your ZIL files.

GOTO: This routine takes one argument, which should be a room:

```
<GOTO ,JAIL>
```

It sends the player to that room, and does all the appropriate things, such as call the room's action routine with M-ENTER, and call the describers. V-WALK, the routine which normally handles all movement, calls GOTO; however, there are many instances when you will want to call it yourself, such as when the player pushes the button in the teleportation booth. Some games allow GOTO to work with a vehicle as well as a room.

DO-WALK: Takes one argument, which is a direction:

```
<DO-WALK ,P?WEST>
```

The game will now attempt to walk the player in that direction. Notice the difference between GOTO and DO-WALK. DO-WALK is just an attempt, and the response might be something like "The door to the west is locked." GOTO overides all that, however, and positively sends the player to the given room.

JIGS-UP: Takes one argument, a string:

```
<JIGS-UP "The guillotine blade descends.">
```

This is the routine that "kills" the player. Most games follow the supplied string with a message like

```
*** You have died. ***
```

Some games allow for several "resurrections." All games should follow a death with an opportunity to RESTART or RESTORE.

THIS-IS-IT: Takes one argument, an object:

```
<THIS-IS-IT ,GOLDEN-ARROW>
```

Normally, IT is set to the most recent PRSO. For example, if the input was SHOOT THE BOW AT THE CENTER TARGET, then the current IT would be BOW—that is, DROP IT as the next input would be taken to mean DROP BOW. Calling THIS-IS-IT allows you to change IT. You might want to do this if some action or description in the output highlighted a particular object. The example above might occur after a bit of text like, "The Sheriff of Nottingham hands you the grand prize, a golden arrow."

INIT-STATUS-LINE: Takes an optional argument of T:

```
<INIT-STATUS-LINE T>
```

This sets up the status line, which fills the top of the screen in most IF games. For a fairly typical status line, this routine would draw a couple of lines in inverse video, and then print "Location:" and "Score:" and "Moves:" in the appropriate places. INIT-STATUS-LINE can be modified to your hearts content to make your own status line as austere or baroque as you please. The optional T tells INIT-STATUS-LINE not to clear the entire screen before going to work.

UPDATE-STATUS-LINE: Usually takes no arguments:

```
<UPDATE-STATUS-LINE>
```

This is the natural partner of INIT-STATUS-LINE. Whereas INIT-STATUS-LINE is usually called only at the beginning of the game, or when the screen is cleared

for some reason, UPDATE-STATUS-LINE gets called just about every turn. It changes the room name, if needed, and the score, if needed, and updates the number of moves, and anything else that's called for on your special little status line.

ITALICIZE: Takes one argument, a string:

```
<ITALICIZE "oy gevalt">
```

Causes the string to be appear in italics rather than in the normal font. On those machines which don't support italics (that is, most) the string will appear underlined.

GAME-VERB?: Takes no arguments:

```
<GAME-VERB?>
```

This routine, as you would expect by the trailing question mark in the title, is used as a predicate. It returns true if PRSA is one of a list of verbs that don't take a turn. Some examples of such verbs are VERBOSE, SCRIPT, and $VERIFY. You might create such a verb over the course of writing a game; if so, don't forget to add it to the GAME-VERB? list.

ROB: Takes one argument, an object, as well as an optional argument, which could be a room or and object:

```
<ROB ,MONTY-HALL ,BOX-BEHIND-CURTAIN-TWO>
```

ROB empties the supplied object—that is, it moves everything whose LOC is that object. If the optional argument is supplied, ROB moves the contents of the object to there. If no optional destination is supplied, ROB simply removes the contents (leaving the contents in limbo, without a LOC).

WEIGHT: Takes one argument, usually an object:

```
<WEIGHT ,SANTA-SACK>
```

Determines the total size of a container by adding its own SIZE to the SIZEs of any objects within the container, going recursively down as many levels as necesary.

PICK-ONE: Takes one argument, the name of a table:

```
<PICK-ONE ,SNIDE-COMEBACKS>
```

Randomly picks one element from the supplied table. There are two flavors of PICK-ONE around. The older version picks a random element each time it is called. The more modern version "remembers" which elements have been previous returned, and won't repeat an element until every element in the table has been returned once. Most commonly, this routine is used along with a table of strings in order to give variety to a common response. For example, since taking untakeable objects is so common, the V-TAKE default is a PICK-ONE from a table called YUKS which includes such classic responses as "Not bloody likely." and "What a concept!"

VISIBLE?: Takes one argument, an object:

```
<VISIBLE? ,SECRET-DOOR>
```

This routine, which is used as a predicate, returns true if the supplied object is visible to the player; that is, if it can be currently referred to.

ACCESSIBLE?: Takes one argument, an object:

```
<ACCESSIBLE? ,STAR-OF-SIAM>
```

Similar to VISIBLE?, except that it also checks for whether the object can be gotten. For example, an object inside a closed, transparent container would be visible but not accessible.

UNTOUCHABLE?: Takes one argument, an object:

```
<UNTOUCHABLE? ,GROUNDHOG>
```

Another in our exciting line of predicates, UNTOUCHABLE? returns true if the object supplied is out of reach of the player at the current time. It is usually used for the case where a player is inside a vehicle and is interacting with an object outside the vehicle.

WITHIN?: Takes four arguments, all integers:

```
<WITHIN? .LEFT .TOP .BOTTOM .RIGHT>
```

This routine determines whether the coordinates of a mouse click fall within a rectangle defined by the four data points supplied. The first two numbers are the X and Y of the top-left corner of the rectangle; the third and fourth numbers are the X and Y of the bottom-right corner of the rectangle.

META-LOC: Takes one argument, and object:

```
<META-LOC ,FUSE-17>
```

This routine take the supplied object and recurses until it determines what room the object is currently in. META-LOC then returns that room. META-LOC will return false if the ultimate location of the supplied object is not a room: for example, if the object has been removed (its LOC is false), or if the object is inside an object which has been removed, etc.

OTHER-SIDE: Takes one argument, an object which is a door:

```
<OTHER-SIDE ,FRONT-DOOR>
```

This routine returns the room on the other side of the supplied door from the player's current room. For example, if the player were on the Front Porch, OTHER-SIDE would return Foyer; if the player were in Foyer, OTHER-SIDE would return Front Porch.

NOW-DARK?: Takes no arguments:

```
<NOW-DARK?>
```

Despite the question mark, this is not a predicate. It is called when the player has just done something which might potentially leave him/her in the dark, such as extinguishing a light source, or closing a container which might contain the player's only light source. NOW-DARK? checks, and if it is now dark, it informs the player, and perhaps warns the player about grues, rats, boogey men, or whatever.

NOW-LIT?: Takes no arguments:

```
<NOW-LIT?>
```

This is the counterpart of NOW-DARK? It is called when the player may have done something to provide light to a previously dark condition. It tells the describers that the player can now see and that a room description is in order.

HELD?: Takes an argument, an object, as well as an optional argument, which can be either a room or an object:

```
<HELD? ,POISON ,DOCTOR>
```

If no optional argument is supplied to HELD? it assumes that the second argument is the PLAYER object. HELD? takes the first object and recurses to determine if it is ultimately within the second object. It differs from <IN? ,POISON

,DOCTOR> because that predicate will only be true if the POISON object has the DOCTOR object as its LOC; the HELD? predicate will be true even if the POISON is inside a bottle which is inside the black bag that the doctor is carrying. Some authors, who value accuracy above typing speed, call this routine ULTIMATELY-IN?

TOUCHING?: Takes one argument, an object, and decides, based on the current PRSA, whether the player must "touch" the object in order to perform his or her action. Verbs such as TOUCH, TAKE, SHAKE, PUSH, and many more, all require the player to "touch" the PRSO; there are a few which require the player to "touch" the PRSI. Here's an example. The INGOT has been in the bed of coals, and is red hot. The INGOT's action routine would have a clause like:

```
<COND (<TOUCHING? ,INGOT>
       <TELL "The ingot is red hot; you'd scorch your
fingers.">)>
```

This is obviously much better than having a long <VERB? ...> predicate each time you want to make a check like this.

CANT-SEE: Takes an object and prints "You can't see any ... here." thus imitating the parser failure with the same language. Rather than plugging in the object's DESC, it actually uses the player's noun phrase, as the parser would. This is useful if an object is "found" by the parser but shouldn't be referenceable in the current game situation:

```
<ROUTINE PUDDLE-F ()
     <COND (,AFTER-NOON
            <CANT-SEE ,PUDDLE>)
           (<VERB? EXAMINE>
            <TELL "The warm sun is quickly drying out
the puddle. It probably will be gone by noon." CR>)>>
```

RUNNING?: This routine takes the name of an interrupt routine and determines whether that routine is currently running; that is, whether it will be called by CLOCKER at the end of the current turn. Here's an example:

```
<ROUTINE GAZEBO-EXIT-F ()
     <COND (<RUNNING? I-RAINSTORM>
            <TELL "You'd ruin your new perm!" CR>
            <RFALSE>)
           (T
            ,GARDEN)>>
```

GLOBAL-IN?: This routine takes two arguments, an object and a room, and returns true if the object is a local-global in that room. Let's say you had a room called DINING-ROOM. If the definition for DINING-ROOM included:

```
(GLOBALS CARPET)
```

then

```
<GLOBAL-IN? ,CARPET ,DINING-ROOM>
```

would be true. If DINING-ROOM had no GLOBALS list, or if its GLOBALS list didn't include CARPET, then the call to GLOBAL-IN? would be false. Here's another example:

```
<ROUTINE V-FILL ()
     <COND (<GLOBAL-IN? ,WATER ,HERE>
           <TELL "You fill" T ,PRSO " with water." CR>)
```

```
               (T
                <TELL "There's nothing here to fill" T ,PRSO
     "with!" CR>)>>
```

SEE-INSIDE?: A small routine which takes an object—a container—and returns true if the player can see the contents of the container (i.e. is it open or transparent).

CAPITAL-NOUN?: This is used by CANT-SEE and some parser routines to determine whether to capitalize a word when repeating back a noun from the player's input, and whether "any" should appear before it:

```
     [You can't see Cincinatti here.]
     [Which Elvis do you mean, Elvis Presley or Elvis
     Costello?]
```

rather than

```
     [You can't see any cincinatti here.]
     [Which elvis do you mean, Elvis Presley or Elvis
     Costello?]
```

Whenever you create a vocabulary word that should be capitalized, be sure to add it to the list or words in CAPITAL-NOUN?.

FIND-IN: If you pass this routine a location and the name of a flag, it will return the one object there which has that flag. If there are no objects in that location with that flag, or if there are more than one object with that flag, FIND-IN will return false. Example:

```
     <COND (<SET PREY <FIND-IN ,TRAP ,ANIMALBIT>>
            <TELL "A " D .PREY " is caught in the trap,
     whimpering pitifully." CR>)>
```

Optionally, FIND-IN takes a string. If supplied with a string, FIND-IN will print the string and the found-object in brackets before the normal response, simulating what the parser does when it does a FIND. Example:

```
     <COND (<AND <VERB? UNLOCK>
                 <NOT ,PRSI>
            <SET KEY <FIND-IN ,PLAYER ,KEYBIT "with">>>
            <TELL "You unlock the door with" TR .KEY>)>
```

would produce:

```
     >UNLOCK DOOR
     [with the purple key]
     You unlock the door with the purple key.
```

Appendix D: ZIL Instructions

ZIL instructions, also called op-codes, are the method by which you communicate with the interpreter. This is just a partial list; some op-codes are so obscure you'll never need to know them. A complete description of every instruction can be found in the ZIP Specification.
In the list below, arguments to the instruction are listed after the name of the instruction. Optional arguments are italicized.

Arithmetic Instructions

ADD integer1 integer2
Adds the two given numbers and returns the sum. This usually appears in ZIL code as a "+"—the compiler changes it to "ADD." Example:

```
<+ ,FRONT-SEAT-PASSENGERS ,BACK-SEAT-PASSENGERS>
```

SUB integer1 integer2
Subtracts integer2 from integer1 and returns the difference. The compiler changes "-" to "SUB." Example:

```
<- ,LT-BLATHER-ANGER 5>
```

MUL integer1 integer2
Multiplies the two given numbers and returns the product. The compiler changes "*" to "MUL." Example:

```
<* <MARTIANS-IN-ROOM> ,ANTENNA-ON-A-MARTIAN>
```

DIV integer1 integer2
Divides integer1 by integer2 and returns the quotient, truncated to an integer if necessary. The compiler changes "/" to "DIV." Example:

```
</ ,SCREEN-WIDTH 2>
```

MOD integer1 integer2
Divides integer1 by integer2 and returns the remainder. Example:

```
<MOD ,PEBBLES-IN-PILE 10>
```

RANDOM integer
Returns a random number between one and the given number, inclusive. Example:

```
<RANDOM 17>
```

Predicate Instructions

EQUAL? arg1 arg2 *arg3 arg4*
Returns true if arg1 is equal? to any of the subsequent args. Example:

```
<EQUAL? ,HERE ,OVAL-OFFICE ,ROSE-GARDEN ,PORTICO>
```

ZERO? arg
Returns true if the value of arg is zero. This often appears as

```
<EQUAL? ,WHATEVER 0>
```

in your game code; the compiler converts it to the ZERO? instruction. Example:

```
<ZERO? ,FUEL-LEVEL>
```

LESS? integer1 integer2
Returns true if integer1 is less than integer2. The compiler converts L? to LESS?. Example:

```
<L? ,AIR-PRESSURE 4>
```

GRTR? integer1 integer2
Returns true if integer1 is greater than integer2. The compiler converts G? to GRTR?. Example:

```
<G? ,GONDOLA-WEIGHT ,BALLOON-LIFTING-CAPACITY>
```

FSET? object flag
Returns true if flag is set in object. Example:

```
<FSET? ,BRASS-LAMP ,ONBIT>
```

IN? object1 object2
Returns true if object2 is the LOC of object1. (NOTE: will return false if object1 is merely inside an object which is inside object2.) Example:

```
<IN? ,SECRET-WILL ,WALL-SAFE>
```

Object Operations
MOVE object1 object2
Puts object1 into object2. Example:

```
<MOVE ,BREAD ,TOASTER>
```

REMOVE object1
Removes object, setting its LOC to false. Example:

```
<REMOVE ,ICE-CUBE>
```

LOC object1
Returns the location of object. Returns false if object has no location. Example:

```
<LOC ,SMOKING-GUN>
```

FIRST? object1
Returns the first object within object1. Returns false if object1 has no contents. Example:

```
<FIRST? ,REFRIGERATOR>
```

NEXT? object1
Returns the next object in the linked contents of object1's LOC. Returns false if object1 is the "last" object in its LOC. Example:

```
<NEXT? ,MAYONNAISE>
```

FSET object1 flag1
Sets flag1 in object1. Example:

```
<FSET ,OILY-TORCH ,FLAMEBIT>
```

FCLEAR object1 flag1
Clears flag1 in object1. Example:

```
<FCLEAR ,GUARDED-DIAMOND ,TRYTAKEBIT>
```

GETP object1 property1
Returns the specified property of object1. Example:

```
<GETP ,HERE ,P?LDESC>
```

PUTP object1 property1 thing
Changes the value of the given object's given property to thing. Example:

```
<PUTP ,ROTTING-TOMATO ,P?SDESC "rotten tomato">
```

Table Operations
GET table1 integer1
Returns the value that is stored in the integer1th slot in the given table table. Example:

```
<GET ,LATITUDE-TABLE 30>
```

PUT table1 integer1 thing
Changes the integer1th slot of the given table to thing. Example:

```
<PUT ,SUSPECTS-TABLE ,SUSPECTS-POINTER ,BUTLER>
```

INTBL? thing table1 length
Returns true if thing is found within the given table. The third argument is an integer representing the number of elements in the given table. Example:

```
<INTBL? ,RUDOLPH ,REINDEER-TABLE 8>
```

COPYT table1 table2 integer1
Copies table1 into table2. The process stops at the integer1th slot number; if you desire, all of table1 doesn't have to be copied to table2. Example:

```
<COPYT ,CURRENT-MOVE-TBL ,OLD-MOVE-TBL ,MOVE-TBL-LEN>
```

Input Operations

READ table1 *table2 integer1 routine1*
This is the most common way for getting the player's input in an interactive fiction game. It tells the interpreter to get the player's input and store it in table1. The main READ, which reads the normal input following the normal prompt, lives in the parser and you never have to worry about it. However, you'll occassionally want to do a READ without going through the parser, such as the routine FINISH does to determine whether you want to restart, restore, or quit. Don't worry about the optional arguments. Example:

```
<READ ,P-INBUF-TBL>
```

INPUT integer1 *integer2 routine1*
Input is similar to read, except that it reads a single keystroke, rather than a line of text. The argument is a number corresponding to an input device; as of now, the only input device defined is the keyboard, with number 1. The first optional argument tells INPUT, rather than waiting forever for a keystroke, to wait only that many tenths of a second. The second optional argument is the name of a routine that INPUT should call if it "times out"—that is, if it gets no keystroke within the allotted time. Example:

```
<INPUT 1>
```

MOUSE-INFO table1
The interpreter will put four pieces or information about the mouse into table1, which naturally must be at least 4 elements long. The four pieces of info are, in order: the y position of the mouse cursor (in pixels), the x position, which (if any) mouse button has been pressed, and the menu or menu item selected. Example:

```
<MOUSE-INFO ,MOUSE-INFO-TBL>
```

MOUSE-LIMIT window
You can have up to 8 windows in YZIP. The main, scrolling, text window is Window 0. Normally, the mouse is active in every window; this restricts the mouse to the given window. Example:

```
<MOUSE-LIMIT 0>
```

MENU integer1 table1
This allows you to add a menu to the menu bar (for those computers that have them). As of this writing, only the Mac interpreter has this feature implemented. Integer1 is the number of the menu bar slot where your menu should appear;

this number must be greater than 2, because slot below that are reserved for permanent menus. Table1 is an LTABLE of strings for the menu; the first string should be the name of the menu. Example:

```
<MENU 3 ,BATTLE-COMMANDS-TBL>
```

Output Operations

PRINT string1

Prints the given string to the current window. Like most of the printing-related instructions that follow, the compiler will convert your TELLs into the appropriate set of printing instructions. Example:

```
<PRINT "Not bloody likely.">
```

PRINTD object1

Prints the DESC of the given object. Example:

```
<PRINTD ,WICKER-BASKET>
```

PRINTN integer1

Prints the given number. Example:

```
<PRINTN ,DIAL-SETTING>
```

BUFOUT integer1

Tells the interpreter whether to buffer output. If integer1 is 1, output is buffered and sent to the screen a line at a time (this is the normal, default behavior). If integer1 is 0, all output is sent immediately to the screen without buffering. Example:

```
<BUFOUT 0>
```

CRLF

Prints an end-of-line sequence. Example:

```
<CRLF>
```

HLIGHT integer1

Tells the interpreter how to display text, according to the following values of integer1: 0 - no highlighting; 1 - inverse video; 2 - bold; 4 - underline/italic; 8 - monospaced font. Constants, such as H-INVERSE and H-ITALIC exist so that you don't have to remember these numbers. Example:

```
<HLIGHT ,H-BOLD>
```

COLOR integer1 integer2

Sets the foreground and background colors according to the following values: -1 - color of pixel at cursor location; 0 - no change; 1 - default color; 2 - black; 3 - red; 4 - green; 5 - yellow; 6 - blue; 7 - magenta; 8 - cyan; 9 - white. The Amiga has three additional colors: 10 - light gray, 11 - medium gray; 12 - dark gray. Example:

```
<COLOR 0 ,FANUCCI-BACKGROUND>
```

DIROUT integer1

Tells the interpreter to commence sending output to the specified device, according to the following values of integer1: 1 - screen; 2 - printer; 3 - table; 4 - command file. Example:

```
<DIROUT ,D-PRINTER-ON>
```

DIRIN integer1

Tells the intepreter to commence receiving input from the specified device, according to the following values of integer1: 0 - keyboard; 1 - command file. Example:

```
<DIRIN 1>
```

Window Operations

CURSET integer1 *integer2 integer3*

This moves the cursor to a point on the screen corresponding to y = integer1 (in pixels) and x = integer2. Integer3 is the number of the window; if not supplied, the cursor will move to that point in the current window. The only case where integer2 is optional is when integer1 is -1 or -2; -1 tells the interpreter to "turn off" or "hide" the cursor; -2 means to turn it back on. Example:

```
<CURSET 1 </ ,WIDTH 2>> ;"top center of the screen"
```

CURGET table1

Puts the location of the cursor into table1; the y coordinate will be element 0 and the x coordinate will be element 1. Example:

```
<CURGET ,CURSOR-LOC-TBL>
```

SCREEN integer1

Moves you to the window of that number; all subsequent output will be sent to that window. Example:

```
<SCREEN 2>
```

CLEAR integer1

Clears one of the 8 windows, depending on the given integer. Example:

```
<CLEAR ,S-TEXT>
```

WINPOS integer1 integer2 integer3

Sets the position of the integer1th window to a location on the screen identified by y = integer2 (in pixels) and x = integer3. Example:

```
<WINPOS ,FOOTNOTE-WINDOW 33 12>
```

WINSIZE integer1 integer2 integer3

Sets the size of the integer1th window to a height of integer2 (in pixels) and a width of integer3. Example:

```
<WINSIZE 0 ,TEXT-WINDOW-HEIGHT ,TEXT-WINDOW-WIDTH>
```

WINATTR integer1 integer2 *operation*

Sets the characteristics of the integer1th window, according to the the following values of integer2: 1 - wrapping; 2 - scrolling; 4 - scripting; 8 - buffering. I have no idea what "operation" means. Example:

```
<WINATTR ,TEXT-WINDOW 15> ;"all four attributes"
```

SPLIT integer1

Splits the screen into window 0 and window 1, with the division at a distance of integer1 from the top of the screen (in lines of text). This has the effect of simulating the old-fashioned style of pre-YZIP interactive fiction. SPLIT is now discouraged in favor of WINPOS and WINSIZE. Example:

```
<SPLIT 2>
```

MARGIN integer1 integer2 *window1*

Sets the left margin to integer1 (in pixels) and the right margin to integer2. This refers to the width of the margin, not its location. In other words, margins set to 0 and 0 would be the entire width of the screen (which is the default case).

Window1 is the window number of the window you want changes the margins in; if not supplied, it defaults to the current window. Example:

```
<MARGIN 20 20>
```

SCROLL integer1 *integer2*
Scrolls the integer1th window. If integer2 is supplied, the window will scroll by that number of text lines. If not supplied, the window will scroll one line. Example:

```
<SCROLL ,S-TEXT 4>
```

Pictures and Sound

DISPLAY integer1 integer2 integer3
Displays the picture whose number is integer1. It will be displayed with its top-left corner at a point specified by y = integer2 and x = integer3 (in pixels). Example:

```
<DISPLAY ,P-TITLE 1 1>
```

PICINF integer1 table1
Stores the dimensions of the picture whose number is integer1 in table1. The height of the picture (in pixels) will be in element 0 and the width will be element 1. Example:

```
<PICINF ,WATERFALL-PIC ,PICINF-TBL>
```

SOUND integer1 *integer2 integer3 integer4*
Produce the sound whose number is integer1. Integer2, if supplied, has the following meanings: 1 - initialize the sound; 2 - start the sound (this is the default value); 3 - stop the sound; 4 - clean up sound buffers. If integer3 is supplied, it determines the volume at which the sound will be played. If integer4 is supplied, it repeats the sound that many times. Example:

```
<SOUND ,CAR-BACKFIRE 2 5 2>
```

Control Operations

CALL routine1 *arg1 arg2 arg3*
When the compiler encounters a bracketed call to a routine in your ZIL code, it converts it into the CALL op-code. You don't need to use CALL in your ZIL code.
RETURN anything
Returns any to the routine which called the current routine. Example:

```
<RETURN .COUNT>
```

RTRUE
Returns 1 (that is, true) to the routine which called the current routine. Example:

```
<RTRUE>
```

RFALSE
Returns 0 (that is, false) to the routine which called the current routine. Example:

```
<RFALSE>
```

GameCommands

QUIT
The game should end, in whatever way is appropriate for the machine and its operating system. Example:

```
<QUIT>
```

RESTART
Causes the game to act as though it had just started from scratch. Example:

```
<RESTART>
```

VERIFY
Does a checksum of the bytes in the program to make sure that the program is correct. Returns true if correct. Example:

```
<VERIFY>
```

SAVE *table1 integer1 table2*
Saves the "impure" part of the game in some permanent, recoverable way, as determined by the characteristics of the particular micro. I haven't the foggiest idea what the optional arguments are; never seen 'em used. Example:

```
<SAVE>
```

RESTORE table1 integer1 table2
Recovers a previously made save, once again according to a procedure determined by the particular interpreter. Example:

```
<RESTORE>
```

ISAVE
This makes a save, invisible to the player, to the machine's memory (provided that the machine has enough memory). Example:

```
<ISAVE>
```

IRESTORE
This restores the internally-stored RAM-save. Together with ISAVE, this is what allows UNDO to work. Example:

```
<IRESTORE>
```

Other instructions (see YZIP Spec for more details):

BTST	PRINTC
BAND	PRINTB
BOR	PRINTI
BCOM	PRINTR
SHIFT	PRINTT
ASHIFT	PRINTF
NEXTP	FONT
GETB	ERASE
PUTB	WINGET
GETPT	WINPUT
PTSIZE	DCLEAR
VALUE	CALL1
SET	CALL2
ASSIGNED?	XCALL
INC	ICALL1
DEC	ICALL2
IGRTR?	ICALL
DLESS?	IXCALL
PUSH	CATCH
XPUSH	THROW
POP	JUMP
FSTACK	RSTACK
LEX	NOOP
ZWSTR	ORIGINAL?

Index