The Text Adventure Development System

Version 2.0

Author's Manual

The Text Adventure Development System Author's Manual

Cover illustration by Andrew Van Miller II.

Revision 2.0 (November, 1992)

Printed in the United States.

Contents

Preface

About TADS

TADS is a computer programming language system created especially for writing text adventures. The syntax and structure of the language are designed to make sophisticated adventure games simple and convenient to implement. TADS provides a complete run-time system with all of the features required for a professional adventure game, so the game author can concentrate on his adventure and not on the underlying software.

This section provides an overview of TADS, of text adventures in general, and of this manual.

About Text Adventures

The first text adventure game was called, appropriately enough, Adventure. The game was an interactive simulation of a cave. The player interacted with the game by typing commands such as "north" and "get rod," and the game responded by describing the consequences of the commands.

In many ways, Adventure was quite rudimentary; each player command was limited to one or two words, and the range of possible actions was quite small. However, Adventure was different from any game that had gone before it, on a computer or otherwise, and it quickly attracted a large following. The game became so popular, in fact, that it spawned an entire genre of games, called "adventure games" after the first of their kind. As the genre has matured and computer hardware has become capable of supporting more sophisticated programs, many improvements have been made to adventure games, but the basic ideas have remained largely unchanged.

Adventure games appeal to a wide audience. Unlike most arcade-style games, an adventure can be won, and it can take many days, or even weeks or months, to finish a game. At their best, adventure games are rich stories in which a player can become immersed.

About TADS

Many people are so intrigued with the adventure game genre that they'd like to write adventures themselves. Most people who start writing these games, though, quickly become overwhelmed by the large amount of code that must be written. To write an adventure, a potential author must first write a parser to read player commands, write code to save and restore games, set up data structures to simulate the objects in the scenario, and write many other complicated routines. Not all imaginative people are computer experts, so many potential adventure authors are discouraged from trying their hand at the genre.

Fortunately, the most complex operations that an adventure game must perform are the same, or very nearly so, in every game. Some people interested in adventure game development have discovered that these common operations—displaying messages, checking and setting states of game objects, saving and restoring games, and so forth—could be done by common subroutine libraries that are shared by any number of games. Others have taken this idea further, realizing that a special-purpose language designed specifically for text adventure implementation could simplify game authoring. TADS is such a language.

Authors writing games with TADS will realize several benefits over writing games in a general-purpose language. Some of the advantages of using TADS are:

- A great deal of code is already written for you: a full player-command parser, which supports multiple objects in a single command, multiple commands on a line, use of the word "all," and many other sophisticated features; file manipulation code to save and restore games; code to save a transcript of a game to a file; string compression logic that reduces the disk and memory space required for a game; user-interface code; and many other functions that would be time-consuming to write from scratch. Hence, the game author can concentrate on the game, and ignore the low-level utility code that supports it.
- TADS source code requires no modification to be compiled on any computer where the TADS system runs. TADS is available on many computer systems, which means that games are automatically portable to many computers with no effort beyond file transfer. Furthermore, with TADS version 2, the compiled binary game file is fully portable, so you don't even need to recompile your game when moving it to a new system. While porting a game written in a general-purpose language generally requires substantial work, TADS programs will always run the same way in every environment without any changes.
- It is much easier to write a game in TADS than in a general-purpose language, because TADS is specifically designed for text adventures. The language has several constructs aimed at making adventure games convenient to write, and the object-oriented nature of the language makes game design natural. Furthermore, the standard object classes provided with the system are designed so that common components of adventures can be implemented with practically no coding. So, even

if all of the support code and portability that TADS offers were available in a general-purpose language, the syntactic and structural efficiency of TADS make it superior for text adventure implementation.

About Version 2

TADS Version 1 was a very successful and useful piece of software, but there's always room for improvement. We wanted to make TADS even more flexible and powerful, and we wanted to address many of the suggestions that users of version 1 made for enhancing the system. Some of the more important features that are new in version 2 are listed below.

- Virtual Object Cache: one of the most important new features in version 2 is a virtual memory subsystem that allows your game to be bigger than available memory. Instead of requiring your game to fit into memory all at once, as version 1 did, TADS version 2 only requires the objects that are being used at any given time to be in memory. Other objects are stored in a disk file. This new feature makes it possible to run an enormous game on a computer with little memory, and makes the capacity of TADS version 2 effectively unlimited on all computer systems.
- Undo: the run-time system now allows a player to "undo" commands. Unlike some adventure game systems, TADS keeps large amounts of undo information, allowing the player to undo many turns (over a hundred in the default configuration). Undo should make games a lot more fun to play, because it will remove the tedium of having to save and restore frequently when trying dangerous things.
- Additional C-like Syntax: TADS version 2 adds many new constructs from the C language. The `for`, `do-while`, `switch`, and `goto` statements are now supported, exactly as in C. Local variables can be initialized in their declarations, and you can have multiple `local` statements at the start of a block. The operators '`+=`', '`-=`', '`*=`', '`/=`', '`++`', '`--`', and '`,`' (the comma operator) have been added, and behave as in C (with the extension that `+=` can be used with lists and strings, and `-=` can be used with lists).
- Implicit `self`: when evaluating or assigning to a property of `self`, you can now omit the "`self.`" prefix on the property—the property name can simply be used as though it were a local variable.
- Property and Function Pointers: you can now store a "pointer" to a property or function in a variable, and then use it to call the property or function.
- Status Line Customization: you can now display whatever you want in the score portion of the status line. You can display any text string in this area, allowing you to display the time of day, the player's rank, or whatever else you want in place of the standard score and turn-counter display.

- Highlighted Text: games can display text that is "highlighted." On some systems, this text will appear in a different color, while on others it will appear with a different typeface (bold or italic).

Version 2 is a nearly complete rewrite of TADS. Fortunately, we only wanted to improve TADS, not replace it, so we made sure that games written for version 1 will continue to work with version 2. Games written for version 1 will automatically take advantage of many of the new features in version 2, such as the Virtual Object Cache and run-time Undo. Authors of new games written with version 2 will have access to even more, such as additional C-like language syntax.

About This Manual

This document is intended for game designers who wish to implement games using TADS. This manual will try to describe the system in such a way that no prerequisites are imposed on a potential author, but you'll have the easiest time learning to use TADS if you're already familiar with a traditional structured language such as Pascal or C. Familiarity with the concepts of object-oriented programming will also help.

Here's an overview of the contents of this book:

- Chapter One leads you through a sample game, so you can see the big picture of how a TADS game works.
- Chapter Two explains object-oriented programming in general. TADS has been heavily influenced by this style of programming.
- Chapter Three is a detailed introduction to the language.
- Chapter Four describes the run-time environment of a TADS program; in particular, this chapter describes the player command parser.
- Chapter Five is a reference to the TADS language.
- Chapter Six describes the process of writing a game with TADS, from inspiration to implementation.
- Chapter Seven describes how to implement a variety of common adventure game elements, and includes many examples of how to write TADS programs.
- Several Appendices provide additional details on several topics: using the compiler, error messages, the standard adventure definitions, and how to design a good adventure game.

To use this manual to learn TADS, you should start with Chapters One and Two, but just to get the flavor of the system; don't worry about understanding all of the details right away. You should then proceed to Chapter Three to become familiar with the language. While you are reading this chapter, it may not seem immediately clear how to write an adventure with the language; for now, just try

to get an overview of the language. Then, read Chapter Four, which describes how you use the language to write an adventure; this chapter explains the TADS run-time environment in general, and the player command parser in particular. While learning TADS, you will almost certainly find it useful to refer frequently to the lengthy sample game provided with the TADS software, since it has many examples of how the TADS language constructs are used in an actual game.

Chapter Five is reference material, describing the TADS language in detail. This chapter is not intended to be a tutorial, but is designed to provide easy access to the details of writing a TADS game. Likewise, the several Appendices document the microscopic details about the darker corners of the system, such as compiler options and error messages.

Acknowledgments

The Text Adventure Development System represents many years of work, not the least of which was the preparation of this documentation. I wish to thank the many people who have contributed to this project. Jim Cser used an early version of TADS to write the now-classic *High Tech Drifter*, and Steve McAdams worked with me on *Deep Space Drifter*; the experience gained in implementing these full-scale games was invaluable. I also wish to thank the several people who reviewed this manual and play-tested the sample game that is included with the system, and the many other people who have offered advice in the course of the project.

I want to thank the many users of version 1 for their comments and suggestions, and for encouraging me to continue supporting and improving the system. Special thanks to David Baggett, Chris Nebel, and David Leary, authors of the Unnkulian series of TADS games, for their help in testing TADS Version 2, as well as their enormous contribution to the popularity of TADS by demonstrating how good a TADS game can be; and to Andy Miller for the awesome Mount TADSmore cover art.

The vitality of thought is in adventure.
— ALFRED NORTH WHITEHEAD, *Dialogues of Alfred North Whitehead* (1953)

I know engineers—they love to change things.
— DR. McCOY, *Star Trek: The Motion Picture* (1979)

Chapter One

A Sample Game

This section introduces many of the concepts of writing a TADS program by explaining how a simple example works. The chapter starts with a very basic TADS program, to demonstrate the general structure of a game, then expands it to show more features of the language. Although the example is very simple, it is a complete, working TADS program.

We are starting with a sample game because it provides an overview of how a TADS program fits together. When you are reading later sections, which describe the elements of the language in detail, it may be helpful to have an idea of where the details fit into the general structure of a game. This chapter should help provide that overview.

A Very Simple Game

We'll start with about the simplest game possible: two rooms, and no objects. (We could conceivably start with only one room, to make things even simpler, but then there would be nothing to do while playing the game; with two rooms, we at least can move between them.)

If you want to try running this game, create a file with the program text as shown below using your text editor or word processor. The TADS Compiler will accept an ASCII file produced with any editor.

```
/* This is a comment, just like in C */
#include <adv.t>                        /* read generic adventure game "adv.t" */
#include <std.t>                                /* read standard underpinnings */

startroom:  room                      /* the game always starts in "startroom" */
  sdesc = "Outside cave"                /* the "Short DESCription" of the room */
  ldesc = "You're standing in the bright sunlight just
  outside of a large, dark, forboding cave, which
  lies to the north.  "
```

```
  north = cave                    /* the room called "cave" lies to the north */
;

cave:  room
  sdesc = "Cave"
  ldesc = "You're inside a dark and musty cave.  Sunlight
  pours in from a passage to the south.  "
  south = startroom
;
```

To run this example, all you have to do is compile it with `tc`, the TADS Compiler, and run it with `tr`, the TADS Run-time system. If you named your sample program `sample.t`, on most operating systems you can compile your game by typing this:

```
tc sample
```

and you can run it by typing this:

```
tr sample
```

Those two commands—`tc` and `tr`—are the only operating system commands you'll need to learn to write games, apart from the commands you use to run your text editor and operating system utilities.

Now we'll walk through the sample game line by line.

The `#include` command inserts another source file into your program. The file called `adv.t` is a set of basic definitions common to most adventures. You should be able to use these definitions, with few changes, for most adventure games. By including `adv.t` in your game, you don't need to worry about definitions for basic words such as "the," a large set of verbs (such as "take," "north," and so forth), and many object classes (more on these in a bit).

The line including `std.t` is similar; it contains some additional standard definitions. The reason for placing these definitions in a separate file is that they will usually be customized in a finished game. When you're in the final stages of polishing your game, you'll usually want to remove the `std.t` definitions and use your own (for example, you will probably replace the `init` function in `std.t` with your own version which displays an introductory message appropriate for your game). However, the definitions in `std.t` are good enough to let you get started with a game.

The line that says `startroom:  room` tells the compiler that you're going to define a room named "startroom". Now, a `room` is nothing special to TADS, but the file `adv.t` that you included defines what a `room` is. A `room` is one of those object classes we mentioned. The next line defines the `sdesc` for this room. An `sdesc` is a short description; for a room, it is normally displayed whenever a player enters the room. The `ldesc` is the long description; it is normally displayed the first time a player enters the room, and can be displayed by the player by typing "look". Finally, the `north` definition says that another room, called `cave`, is reached when the player types "north" while in `startroom`.

A bit of terminology: `startroom` and `cave` are *objects*, belonging to the *class* `room`; `sdesc`, `ldesc`, `north`, and the like are *properties* of their respective objects. In the context of TADS programming, an object is a named entity which is defined like `startroom`; each object has a class, which defines how the object behaves and what kind of data it contains. Note that our usage is sometimes a little loose, and we will also use "object" the way a player would, to refer to something in the game that a player can manipulate. In fact, each item that the player thinks of as an object is actually represented by a TADS object (sometimes several, in fact); but your TADS program will contain many objects that the player doesn't directly manipulate, such as rooms.

If you are familiar with other programming languages, you may notice that the program above appears to be entirely definitions of objects; you may wonder where the program starts running. The answer is that the program doesn't have a beginning, exactly.

TADS is a different style of programming than you have encountered before; this new style may take a little getting used to, but you will find that it is quite powerful for writing adventure games and simplifies the task considerably. Most programming languages are "procedural"; you specify a series of steps that the computer executes in sequence. TADS, on the other hand, is more "declarative"; you describe objects to the system. While TADS programs usually have procedural sections, in which steps are executed in sequence, the overall program doesn't have a beginning or an end.

The reason TADS programs aren't procedural is that the player is always in control of the game. When the game first starts, the system calls a bit of procedural code in your program that displays any introductory text you wish the player to see, then the system waits for a command from the player. Based on the command, the system will manipulate the objects you defined according to how you declare these objects to behave. You don't have to worry about what the player types; you just have to specify how your objects behave and how they interact with one another.

Adding Items to the Game

Now let's add a few items to the game that can be manipulated by the player, so he can do something besides walk back and forth between our two rooms. We'll add a solid gold skull, and a pedestal for it to sit upon.

```
pedestal:  surface, fixeditem
  sdesc = "pedestal"
  noun = 'pedestal'
  location = cave
;

goldSkull:  item
  sdesc = "gold skull"
  noun = 'skull' 'head'
  adjective = 'gold'
  location = pedestal
```

```
;
```

Here we've defined two objects, `pedestal` and `goldSkull`.

The `pedestal` belongs to two classes, `surface` and `fixeditem`. This means that it has attributes of both classes; when there's a conflict, the `surface` class takes precedence, because it's first in the list of classes. Objects of the `surface` class can have other objects placed on top of them; objects of the `fixeditem` class can't be carried. Note that the `surface` class has a default `ldesc` property which lists any objects on the surface.

The `goldSkull` belongs to the `item` class, which is the generic class for objects without any special properties.

Since these objects can be manipulated directly by the player, the player needs words to refer to them. This is what the `noun` and `adjective` properties define. All objects that the player can manipulate must have at least one `noun`. Note that the `goldSkull` has two nouns; they are simply listed with a space between them. Objects can also have adjectives; these serve to distinguish between objects which have the same noun, but are otherwise optional. A good game will recognize all of the words it uses to describe an object, so if you describe the skull as a "gold skull," you should understand it when the player says "take the gold skull."

This brings us to a subtlety. Notice that the `ldesc` and other properties use *double* quotes around their strings, but the `noun` and `adjective` properties have *single* quotes. Vocabulary words (nouns, adjectives, verbs, prepositions, and articles) *always* use single quotes. Almost everything else will use double quotes. The distinction is that a string enclosed in double quotes is displayed immediately every time it is evaluated, while a string enclosed in single quotes is a string value that can be manipulated internally. Double-quoted strings are displayed automatically as a convenience, since most strings in text adventures are displayed without further processing. (Note that the double quote mark is a separate character on the keyboard, and is not simply two single quote marks.)

These two objects have another new property, `location`. This simply defines the object that contains the object being defined. In the case of the `pedestal`, the containing object is the `cave` room; since the `goldSkull` is on the `pedestal`, its location is `pedestal`. (Note that the system makes no distinction, at this level, between an object being *inside* another object and the object being *on* another object. This means that an object can't be both a `surface` and a `container`.)

Making the Items Do Something

The game is still rather bland; it has no puzzles. So, let's introduce a small puzzle. Let's assume that the gold skull wasn't merely left laying around; instead, whoever left it there arranged for a trap to go off if it should be lifted off the pedestal. To implement this, we need to add a *method* to the `goldSkull` object. A *method* is a special type of property which contains code to be executed; it is very much like a function in C or Pascal. The new `goldSkull` with the method looks like this:

```
goldSkull:  item
  sdesc = "gold skull"
  noun = 'skull' 'head'
  adjective = 'gold'
  location = pedestal
  doTake( actor ) =
  {
    "As you lift the skull, a volley of poisonous
    arrows is shot from the walls!  You try to dodge
    the arrows, but they take you by surprise!";
    die();
  }
;
```

The method `doTake` (which stands for "**d**irect **o**bject **take**") takes as an argument the character who is trying to take the object (since we have no characters besides the player in this game yet, `actor` will be the player's character, named `Me`.) Here, we've simply defined it to display a message (since the message is enclosed in double quotes, it is displayed immediately upon being evaluated), then calls a special function called `die`. (The parentheses following `die` tell TADS that you would like to call `die` as a function.)

You should note that we didn't just pick the name `doTake` out of thin air. The `doTake` method in the `goldSkull` object is called by TADS when the player types a "take" command with `goldSkull` as the direct object. Each verb the player types results in the system calling particular methods in the object or objects named in the command. The naming of these methods is described in detail later in this manual.

You might wonder why we didn't need a `doTake` method in our original definition of `goldSkull`, or you might have assumed that the system automatically knows what to do if no `doTake` is defined for an object. In fact, all objects do need a `doTake` method, and the system doesn't automatically know anything about it. However, since practically every object has the same `doTake`, with a few exceptions such as `goldSkull`, it would be extremely tedious to type a `doTake` method for every object in the game. Instead, we use something called "inheritance." By defining the `goldSkull` to be a member of the `item` class, you tell TADS that `goldSkull` "inherits" all of the definitions for `item`, in addition to any definitions it makes on its own. The `item` class, which appears in the `adv.t` file included at the beginning of the program, defines a `doTake` method, so anything that is defined to be an `item` inherits that definition. However, if something is defined in both `item` and `goldSkull`, as `doTake` is in this example, the definition in `goldSkull` takes precedence—it "overrides" the inherited method.

We actually don't have a very good puzzle here, because there's no way to take the gold skull without dying. So, let's put a rock on the cave floor:

```
smallRock:  item
  sdesc = "small rock"
  noun = 'rock'
```

```
  adjective = 'small'
  location = cave
;
```

Now, let's change the `doTake` method of the `goldSkull`.

```
doTake( actor ) =
{
  if ( self.location<>pedestal or                  /* am I off the pedestal?  */
  smallRock.location=pedestal )                     /* or is the rock there?  */
  {
    pass doTake;                                       /* yes - take as usual */
  }
  else                                              /* no - the trap goes off!  */
  {
    "As you lift the skull, a volley of poisonous
    arrows is shot from the walls!  You try to dodge
    the arrows, but they take you by surprise!";
    die();
  }
}
```

This new `doTake` first checks to see if the object being taken (the special object `self`, which is the object to which the message `doTake` was originally sent), which in this case is the gold skull, is already off the pedestal; if it is, we don't want anything to happen, so we *pass* the `doTake` message. We also pass the message if the small rock is on the pedestal. When we *pass* the message, the `doTake` method that we inherit from our parent class (in this case, `item`) is invoked. This allows us to override a method only under certain special circumstances, and otherwise do business as usual. If we don't satisfy one of these two requirements, the volley of poisonous arrows is released as before.

So, the solution to the puzzle is to put the rock on the pedestal before taking the skull, thereby fooling the pedestal into thinking the skull is still there.

This should give you some idea of how a TADS program looks. In later sections, we'll cover the details of the language, and explore all of the classes defined in `adv.t`. You should note that the source code to a much more complex sample game is included with the TADS software; you may find it useful to refer to that sample while reading the rest of this manual for examples of the language constructs. You may also find that parts of the sample game can provide a model for objects in your own games. Also, note that chapter six describes the implementation process in somewhat more detail, and shows you how to build a larger sample game; you may want to look at chapter six after you've become a little more familiar with TADS.

Example is always more efficacious than precept.
— SAMUEL JOHNSON, *Rasselas* (1759)

Chapter Two

Object-Oriented Programming

Somewhere around 1985, the software marketing community discovered the term "object oriented," and latched onto the phrase with a devotion that even "hypertext" and "multi-media" have yet to approach. With so many marketers using the term, and so few knowing what it means, "object oriented" has come to mean almost anything, and hence nothing.

Despite the confusion, the term really does have a meaning. This section will attempt to clarify what object-oriented programming is and how it applies to TADS. This chapter will also explain why an object-oriented language is desirable when writing adventures.

This section is not "required reading" in learning TADS, but it may help you understand some of the general ideas that influenced the design of the system. You may prefer to skip this chapter the first time you read this Author's Manual, and come back to it after you have a general understanding of the language.

Object-Oriented Programming

TADS is an *object-oriented* programming language. Object-oriented languages are similar in many ways to "procedural" languages such as C and Pascal, but approach problem-solving from a different perspective. Because of the shift in viewpoint, object-oriented programming is effective in many different kinds of applications, but it is particularly applicable to simulations. A text adventure is fundamentally a simulation.

What is this "new perspective" of object-oriented programming? Basically, it is the view that one takes of data. In a traditional language, you write a series of procedural steps that are applied to a collection of data; code and data are firmly separated. In object-oriented programming, on the other hand, you partition a problem into a set of entities, called *objects*; each object contains both the data and the code that describes its state and behavior. In a simulation, an object in the program usually corresponds directly to an object being modelled.

Bouncing Balls

Consider a simulation of a physics problem involving bouncing balls of various kinds. In a traditional language, we would define a set of data for each ball (such as its position, mass, and velocity); we would give each ball some sort of identification (perhaps a number, giving an array index) that would let us tell the balls apart. Finally, a subroutine named `bounce` would be written; the subroutine would take a ball number as its parameter, and would apply the appropriate changes to the ball's data.

In contrast, an object-oriented version of the program would model each ball with an object. Not only would the data about a ball be contained in the object representing the ball, but the code needed to describe its behavior under various interactions would be contained in the object as well. So, each ball would have its own `bounce` *method* (the object-oriented word for a subroutine that is associated with a particular object or group of objects). Rather than calling the `bounce` subroutine with an argument specifying ball 3, you would send ball 3 a *message* telling it to bounce ("send a message" is the object-oriented term for calling a method).

Inheritance and Overriding

This may sound like a lot of coding, writing the same subroutine for each ball. Fortunately, object-oriented languages have a feature that avoids this type of duplication. This feature is *inheritance*. An object can be defined as belonging to a particular *class*, and it will inherit all of the code and data defined for the class. Of course, it can still define code and data itself, which can add to or modify the attributes it inherits. When an object defines code or data that is already defined by its class, the definitions the object makes *override* what it inherits from its class.

The utility of inheritance and overriding is that it becomes very simple to represent general and special cases. For example, if you had twenty identical balls, but a twenty-first ball with special behavior when bouncing (but apart from bouncing is the same as all the other balls), you'd first define a `ball` class, and define all twenty-one balls as belonging to this class. The twenty-first ball, though, would override the `bounce` method with its own special version. In a traditional language, on the other hand, the `bounce` subroutine would need a special test to see if the odd ball was bouncing and act accordingly.

This may seem like six of one and half a dozen of the other, but it has a big advantage: it isolates the special case code with the special case object. Just as block-structured languages allow code to be modularized, object-oriented languages allow entire objects—both their state *and* their behavior—to be modularized. Most people find that this makes writing a program much easier, because they only need to think about a special case in one place, rather than tracking down all the subroutines that need to be modified. It is especially helpful when debugging and maintaining a program, because everything pertaining to an object is kept in one place.

Note that since a class can itself be a member of another class, this type of specialization can be extended indefinitely. For example, you could define the classes `rubberBall` and `billiardBall`,

each inheriting characteristics from the more general class `ball` and defining some of their own. Now you could define ten objects of each class, and these objects could themselves define characteristics overriding their two parent classes.

Object-Oriented Adventures

Object-oriented languages are quite useful for writing adventure games. When a player types a command in a TADS game, the system sends certain messages to the object or objects involved. In this way, it's easy to define classes of objects that behave in particular ways in response to player commands. For example, objects that the player can't pick up (such as phone booths and anvils) respond with a simple "You can't have that" to a "take" command, whereas objects that can be carried will move themselves into the player's inventory list. The basic adventure file `adv.t` defines a large number of basic classes, but the real power of TADS is the ability to add classes of your own.

TADS and other Object-Oriented Languages

Users of other languages that have object-oriented features, such as C++ or Smalltalk, will find that TADS has a slightly different approach to object-oriented programming. In particular, TADS makes much less of a distinction between classes and objects than other languages.

In C++ and Smalltalk, a class is a template that specifies the data types stored by members of the class, but only an object actually stores values for the data. (This is analogous to the distinction between a structure and instances of the structure in a language such as C or Pascal: the structure defines the layout of data, but only an instance of the structure actually contains any data.) The inheritance structure pertains to the classes; an object is an instance of a particular class, so only methods and "slots" for data can be inherited—values for data items are not themselves inherited.

In TADS, there is no distinction between classes and instances of classes, in that a class is also an object. Hence, data values as well as methods can be inherited from a parent object. In TADS, the only distinction between a class and a normal object is that a class object is ignored by the player command parser.

This different approach leads to a different style of programming. A TADS program consists mostly of definitions of objects (instances of classes). All of the instances are specifically defined, with values for properties, in the game program. In contrast, a C++ or Smalltalk program usually defines mostly classes, and creates instances at run-time.

The noblest function of an object is to be contemplated.
— MIGUEL DE UNAMUNO, *Mist* (1914)

Chapter Three

Language Overview

The Text Adventure Development System offers game authors a versatile and powerful language, well suited to creating the world models that underlie text adventure games.

The TADS language is a powerful object-oriented language based on C. TADS uses most of the same keywords and operators as C, but has a few changes that make it easier to use. TADS also uses "run-time typing," which means that you don't have to declare in advance the datatypes of your variables, functions, and properties. In addition, TADS has high-level datatypes, such as lists and strings, that make memory management totally automatic.

This chapter provides an overview of the general features of the language.

Functions

Functions will be familiar to users of languages such as C and Pascal. A function is a bit of code that is grouped together and given a name; another part of your program can cause a function to carry out its code by calling it. This is an example of a function in TADS:

```
showSum:  function(arg1, arg2, arg3)
{
  "The sum is:  ";
  say(arg1 + arg2 + arg3);
  "\n";
}
```

This function is called `showSum`. A function name can be any combination of letters and numbers, but must start with a letter. Capital and lower-case letters are different in TADS, so `showSum` is a distinct function from `showsum` and `ShowSum`.

The curly braces indicate where the function starts and ends; the code inside is the body of the function, and is executed each time the function is called.

The items in parentheses, `arg1`, `arg2`, and `arg3`, are the *arguments* to the function. Some functions have no arguments at all; for these functions, the parentheses and argument list are not present. When the function is called, values are specified for the arguments. For example, if the following code is executed:

```
showsum(1, 2, 3);
showsum(9, 8, 7);
```

then the function is called twice: the first time, the values of `arg1`, `arg2`, and `arg3`, respectively, are 1, 2, and 3; the second time, they are 9, 8, and 7. The output is thus:

```
The sum is:  6
The sum is:  24
```

The arguments are entirely "local" to the function; `arg1` is meaningful only within the function `showSum`. The function can also define local variables of its own with the `local` statement; this is explained in more detail later.

Functions can do a great deal more than just display the sum of three numbers. The `if` statement allows statements to be executed conditionally, and the `while` statement allows statements to be executed repeatedly. TADS provides full integer arithmetic, and it also provides operations on strings and lists (which will be described shortly). In addition, a function can return a value to its caller. And, of course, one function can call another (or even itself). In short, most programs you could write with a language such as C or Pascal or BASIC could be written with TADS functions.

Objects

Objects are the primary elements of any TADS game. Each real-world entity that a text adventure models is described by a TADS object.

Note: The term "object" has a specific meaning in the context of TADS programming, and is different from its usual use in text adventures. While playing a game, you think of objects as those things you can pick up and manipulate. In a TADS program, these are indeed represented as objects, but so are many other things that the player doesn't directly manipulate, such as rooms, actors, verbs, and many other things.

A TADS object is something like a C structure or Pascal record, which are collections of related data (such as numbers and character strings) gathered together under a single name for convenience of manipulation. An object has *properties*, which are data items associated with the object, but it also has *methods*, which are bits of code tied to the object. Methods are very much like functions; the difference is that a method is part of an object, whereas a function stands alone, and is not part of any object.

In the following example, only properties, not methods, are defined for an object.

```
robot:  object
  name = 'Lloyd'
  weight = 350
  height = 72
  speed = 5
;
```

This defines `robot` as an object, and specifies a list of properties and their values. The line `name = 'Lloyd'` says that the property `name` has the value `'Lloyd'`, a character string. Likewise, `weight = 350` says that the property `weight` has the numeric value `350`, and so forth. (Note that only integral numbers are supported in TADS; you can't use floating point numbers such as 3.1415926. The maximum size of the numbers that TADS allows is about 2 billion.) The semicolon ends the object definition.

To get at an object's properties, you use the name of the object, then a dot, then the property name. For example, `robot.name` evaluates `name` property of the object `robot`. The function below prints out the value of an object's `name` property.

```
showName:  function(obj)
{
  say(obj.name);
}
```

This function's single statement calls another function, `say`, with the argument `obj.name`. `say` is a built-in function that displays a number or a string. (Note that `say` will determine when it is called which type of data it should display, and act accordingly. Many built-in functions will perform their action based on the type of data they are sent.)

In `robot`, we defined a number and a string. There is an additional special type of string in TADS, enclosed in double quotes. This type of string is displayed automatically whenever it is evaluated. This is a convenient shorthand, because it removes the need to call `say` every time you want to print a string. Since text adventure games display a lot of text, this feature proves to be very convenient.

```
newobj:  object
  greeting = "Hello!\n"
;
```

The `\n` at the end of the string prints a newline (that is, it moves the cursor to the start of a new line). There's a similar special code, `\b`, that prints a blank line. You might wonder why `\b` is needed, when you could just use two `\n`'s; the reason is that TADS will ignore redundant newlines. Most of the time, this makes output formatting easier, because you don't have to worry about having

too many newlines. Once in a while, though, you really want a blank line; you can use `\b` at these times.

Property `greeting`, when evaluated, will simply print the string `Hello!` and a newline. So, rather than calling the built-in `say` routine to display the value of `newobj.greeting`, you would simply evaluate the property itself:

```
printGreeting:  function(obj)
{
  obj.greeting;
}
```

Properties can also have *list* values. A *list* is a set of values enclosed in square brackets, such as this:

```
listobj:  object
  mylist = [1 2 3]
;
```

Elements of a list need not be of the same datatype, but they generally are. Lists are convenient when you need to keep related items together in one place, especially when the group of items changes from time to time.

Several operators and built-in functions perform list operations. Operators are provided to add items to and remove items from a list, and to reference an individual item in a list. Built-in functions are provided to scan through a list, and to find an item within a list.

As an example of how to scan through a list, the function below displays all the elements of the list contained in `listobj.mylist`. This example uses the list indexing operator, [*index*], to obtain the individual entries in the list. It also uses the built-in function `length()` to determine how many elements are in the list.

```
showList:  function
{
  local ind, len;
  len := length(listobj.mylist);                    // Find the list's length
  ind := 1;                                         // start at first element
  while (ind <= len)                                // loop over each element
  {
    say(listobj.mylist[ind]);                           // display this entry
    "\n";                                                // Display a newline
    ind := ind + 1;                               // move on to the next element
  }
}
```

If you're familiar with C, you would probably want to write the loop in the example above using the `for` statement, which would allow you to put the loop initialization, condition test, and loop variable increment together in a single statement. You can use the `for` statement in TADS the same way you use it in C, so you can rewrite the `showList` function with a `for` loop as shown below.

```
showList:  function
{
  local ind;
  local len := length(listobj.mylist);                    // save list's length
  for (ind := 1 ; ind <= len ; ind++)
  {
    say(listobj.mylist[ind]);                             // display this entry
    "\n";                                                  // Display a newline
  }
}
```

An alternative way of scanning items in a list is to use the built-in functions `car()` and `cdr()`. The example below demonstrates these functions.

```
showList2:  function
{
  local cur;                          // A variable for the remainder of the list
  cur := listobj.mylist;                              // Start with the whole list
  while (car(cur))                    // car(list) is the first element of a list
  {
    say(car(cur));                   // Display the first element of rest of list
    "\n";                                                 // Display a newline
    cur := cdr(cur);                        // cdr(list) is the rest of the list;
                                   // that is, everything but the car of the list
  }
}
```

This function loops through the list element by element. Each time through the `while` loop, `cur` is replaced by the `cdr` of `cur`, which is the list minus its first element; the loop finishes when `car(cur)` is `nil`, which means that nothing is left in the list (`nil` is a special datatype which means, effectively, the absence of a value). A `while` loop continues until its condition is either zero or `nil`.

Note that we assigned the list to a local variable before starting the loop, because we want to take the list's `cdr()` each time through the loop; if we assigned this to `listobj.mylist` each time, `listobj.mylist` would end up with nothing left when the function finished. By assigning the list to the local variable, we leave `listobj.mylist` intact.

These are some of the basic datatypes of TADS: numbers, strings (enclosed in single quotes, which are simply values passed around), printing strings (enclosed in double quotes, which have no value but are printed whenever evaluated), lists (values enclosed in square brackets), `nil` and `true` (returned by expressions such as `1 < 2`), and objects. A property can have any of these datatypes.

Methods

This is where the object-oriented nature of this language becomes more visible: a property can contain code instead of a simple data item. When a property contains code, it is called a *method*.

```
methodObj:  object
  c =
  {
    local i;
    i := 0;
    while (i < 100)
    {
      say(i); " ";
      i := i+1;
    }
    return('that is all');
  }
;
```

This object is considerably more complicated than those we have seen so far. The method `c` is a set of statements, which are executed whenever `methodObj.c` is evaluated. The method in this case has returned a value, but this is not necessary; quite often, a method is evaluated strictly for its side effects, such as a message it prints.

Note that a property whose value is a double-quote string acts exactly like a method with the same double-quote string as its only statement, so the following three definitions are synonymous:

```
string1:  object
  myString = "This is a string."
;

string2:  object
  myString =
  {
    "This is a string.";
  }
;

string3:  object
  myString =
  {
    say('This is a string.');
  }
;
```

Note that double-quote strings do not have any value; their only function is that their evaluation displays the string.

Like functions, methods can take arguments. To specify arguments, simply list them as you would with a function after the method name; they act as local variables inside the method just as function arguments do.

```
argObj:  object
  sum(a, b, c) =
  {
    "The sum is:  ";
    say(a + b + c);
    "\n";
  }
;
```

To invoke a method with arguments, the argument values are enclosed in parentheses and placed after the method name, just as with a function call:

```
argObj.sum(1, 2, 3);
```

Inheritance

Objects can inherit properties and methods from other objects. The `object` keyword defines the most general kind of object; in its place, you can use the name of another object. For example,

```
book:  object
  weight = 1                                          // Books are fairly light
;

redbook:  book
  description = "This is a red book."
;

bluebook:  book
  weight = 2                                        // A heavier-than-usual book
  description = "This is a big blue book."
;
```

The first object, `book`, defines a general category. `redbook` defines a particular book, which means it has all the properties of a book, plus any that are specific to it. Likewise, `bluebook` defines a different book. Again, it has all the general properties of a book; however, since it has its own `weight` property, the `weight` property of the more general book object is ignored. Hence, `redbook.weight` is 1, whereas `bluebook.weight` is 2.

Classes

When an object inherits properties from a second object, the second object is called the first object's *superclass*. When an object is the superclass of other objects, the object is called a *class*. Some object-oriented languages make a strong distinction between objects and classes; in TADS, there is very little difference between the two. However, a `class` keyword is provided that specifies that an object is serving strictly as a class; `book` could thus have been defined as follows:

```
class book:  object
  noun = 'book' 'text'
  weight = 1                                          // Books are fairly light
;
```

The only time that the `class` keyword is required is when a class that is not itself an object has vocabulary word properties or a `location` property. The `class` specification prevents the player command parser from mistakenly believing that the player might be referring to the class with his commands.

Multiple Inheritance

An object can inherit properties from more than one other object. This is called *multiple inheritance*. It complicates things considerably, primarily because it can be confusing to figure out exactly where an object is inheriting its properties from. In essence, the order in which you specify an object's superclasses determines the priority of inheritance if the object could inherit the same property from several of its superclasses.

```
multiObj:  class1, class2, class3
;
```

Here we have defined `multiObj` to inherit properties first from `class1`, then from `class2`, then from `class3`. If all three classes define a property `prop1`, `multiObj` inherits `prop1` from `class1`, since it is specified first.

Multiple inheritance is used only rarely, but you will find that once in a while it is a very useful feature. For example, suppose you wanted to define a huge vase; it should be fixed in the room, since it is too heavy to carry, but it should also be a container. With multiple inheritance, you can define the object to be both a `fixeditem` and a `container` (which are classes defined in `adv.t`).

To change your language,
you must change your life.
— DEREK WALCOTT, *Codicil* (1965)

Chapter Four

Writing Adventures

The previous chapter described TADS as a general-purpose object-oriented programming language, with only a few references to its application as a text adventure system. This chapter introduces the "player command parser," which is the core of the user interface to your TADS game.

The player command parser reads commands from the player, resolves those commands into objects in your game program, and invokes methods in those objects according to those commands. The TADS parser is highly sophisticated; it provides advanced features to the player while relieving your game of the need to implement those features. To provide you with a way to customize many of the parser's functions, though, the system requires that you define certain functions and objects.

This chapter describes the features of the parser, how it works, and how it interacts with your game program. The special functions and objects that your game must define are also explained. In short, this chapter describes the complete run-time environment of your game program.

The "preparse" Function

Each time the player enters a complete line of text, and before the TADS player command parser starts looking at the line of text, a special user-defined function called `preparse` is executed.

The function takes a single argument, which is a string containing the text that the player typed on the command line. The complete text is provided, unchanged, preserving the original spaces, punctuation, and capitalization. The actor, verb, and other objects are still unknown when `preparse()` is called, since the parser hasn't seen the string yet.

Your `preparse` function can change the command string to any new string it desires, or it can abort the command altogether, or it can choose to do nothing and let the original player's command be processed as normal.

If `preparse()` returns a string value, the new string replaces the original command that the player typed. The parser processes the new string as though the player had typed it.

If the function returns `nil`, the command is aborted altogether, and the player is immediately prompted for a new command.

If the function returns `true`, the original string typed by the player is used unchanged.

This function is optional. If it's not defined by your game program, the compiler will issue a warning, but your game will still compile and run. The `preparse` function is useful only in certain special situations; most games won't need it. For example, you could use `preparse` to implement a special effect room where anything typed by the player (even complete nonsense) is just echoed until a special command is typed.

The Player Command Parser

The player interacts with your TADS adventure game strictly at the level of English sentences. The game describes the player's whereabouts, and the player responds with a command (an imperative sentence) to do something; the game then responds with a description of the results of this action. The player then issues a new command, and the game responds to this one. Play proceeds in this manner until the player gets tired of this and quits (or completes the game).

The most important part of the game as far as the player is concerned is the story; that is, the descriptions of locations, people, and things, and the responses to commands. The adventure writer's job is to make this part interesting.

However, a very important part of the game, which is ideally invisible to the player, is the parser. The parser inspects the player's commands, and attempts to make them intelligible to the game as implemented by the adventure writer. We say that the parser is "ideally invisible" because the player shouldn't have to worry about—or even notice—the parser. Players never notice the parser when it understands what they say; it's when the parser can't understand, or worse, misunderstands a command, that the player notices the parser.

The job of making the parser invisible is mostly in the hands of the TADS run-time system, which tries to ensure that many different kinds of sentences are understood. However, the adventure writer is responsible for making sure that many synonyms are recognized for each word, because different people have different vocabularies. A good rule of thumb is that game should understand all the words it uses, particularly those referring to important objects.

All player commands have a verb. A verb is all some commands have; for example, the commands "look" and "east" have only a verb.

Other commands take a direct object as well: "take the book" and "drop the box." Note that the article "the" is optional. Unlike some adventure parsers, the TADS parser will not ignore the article; it will ensure that articles are used only with nouns, but the article will obviously not affect the meaning of the noun.

Some commands take an indirect object as well: "put the book in the box" and "give the librarian the book." The first command uses a preposition to identify the indirect object; the second uses the positions of the words in the sentence to specify which object is which.

Some verbs are a little more complicated; consider "pick up the book" and "pick it up." In these cases, the preposition is considered part of the verb, so the verb part is "pick up" and the direct object is "the book."

Note that the TADS parser can only handle prepositions that consist of a single word. Unfortunately, English has many constructions in which multiple prepositions are needed; for example, "take the book out of the bag" has two prepositions in a row.

There's a special mechanism that allows you to define words that are "glued together" when they appear in certain combinations. This mechanism is intended to allow for prepositions that consist of multiple words, but which can be treated as a single word unit. For example, "out of" can be converted to a single word. In order to define words that can be converted into a unit when used together, you use the `compoundWord` statement. This statement appears in your game file outside of functions and objects; `adv.t` defines a large list of compound words, which should be sufficient for most games.

A `compoundWord` statement looks like this:

```
compoundWord 'out' 'of' 'outof';
```

This tells the parser that when it sees the two-word sequence "out of", it should convert it to the single word "outof". The compound word "outof" is now usable in vocabulary (such as in a `preposition` property).

Note that you can't directly define three-word (or longer) compound words, but you can do so indirectly by constructing the longer words out of two-word directives. For example, to convert "out from under" into "outfromunder", you'd do this:

```
compoundWord 'out' 'from' 'outfrom';
compoundWord 'outfrom' 'under' 'outfromunder';
```

Note that the resulting word (the third word in a `compoundWord` command) doesn't have to be the second word appended to the first; older versions of TADS built their compound words automatically in this manner, though, so it is a reasonable convention to follow. You could just as easily define the compound word built from "out from" to be `'out-from'`, or `'asdf'`, or anything else.

There's one more feature: the player can issue a command to another character in the game (called an "actor") by prefacing a sentence with the character's name and a comma, as in "Librarian, take the book off of the shelf."

When parsing a player's command, the entire sentence is converted to lower case letters before anything else is done, so the case of the player's command doesn't matter. In addition, the player need only type the first six letters of any word; if he chooses to type more, though, the extra letters are not simply ignored. This allows words that are longer than six letters to be ambiguous with

other words in their first six letters and still be correctly identified. This is especially useful for dealing with plurals of long words; hence, if a player types "barrel", it will be identified as "barrel" rather than "barrels".

Names of objects are made up of a noun and one or more adjectives. For example, "book," "red book," and "large red book" might all refer to a particular object.

In addition, the word "of" may connect additional qualifying phrases to an object's name. For example, "the large pile of yellow straw." Internally, when designing a game, the adventure writer simply lists the adjectives and nouns that can be used in reference to an object, and doesn't have to worry about placement of the words. In the example above, the player would say that the pile of straw is identified by the nouns "pile" and "straw", and the adjectives "large" and "yellow". Hence, all of these phrases are also recognized as the pile of straw: "straw," "pile," "pile of straw," "large pile," and "yellow straw." Note that the use of the word "of" is completely transparent to the game program; the parser automatically removes the "of" and rearranges the words appropriately.

One more special case occurs when numbers are used as adjectives. In defining an object's adjectives, if you want a number to be used as an adjective, simply include it in the `adjective` list in single quotes just like any other adjective. When the player is typing commands, however, it is handled a little differently than other adjectives: the number can *follow* the noun. This is useful in cases with numbered items, since it's more natural to place the number after the noun. For example, elevator buttons would be called "button 1," "button 2," and so forth. (Of course, referring to them as "1 button," "2 button," and so on, is accepted as well, but it's somewhat ambiguous, in that it sounds like a number of buttons rather than the number of the button.) As with "of," this special case is transparent to the game program, since only the parser needs to worry about the order of the words referring to an object.

Sometimes, there will be ambiguous objects; for example, a red book and a blue book. If the player types simply "book", the parser will try its best to figure out which book the player means. First, it will attempt to identify all the nouns that are meaningful with the command the player is trying to issue, which frequently narrows it down enough that no further information is required. For example, suppose the player is carrying the blue book, and the red book is in the current room; the parser will know that "take the book" refers to the red book, because the blue book doesn't make sense with the take command. However, if both books are in the room, the parser won't know what to do, so it will ask:

```
Which book do you mean, the red book, or the blue book?
```

At this point, the player can answer "red" or "blue," or "the red book," or even "the blue one," in which case the parser will complete the command. He can also say "both" or "all," or even "the red one and the blue one," and the parser will take both books. Now, the player could have said this in the first place, assuming the writer provided a plural: "take the books."

This brings us to the subject of multiple direct objects. The player can list several objects he wants to take, or perform another action on, simply by using a comma between objects: "Take the red book, the box, and the lamp off the shelf."

He can also abbreviate this to some extent with the words "everything" and "all" (which are synonymous); for example, "Take everything off the shelf," or "drop all." If he wants to be a little more specific, he can do something like this: "Drop everything except the keys on the desk," or "Take all but the rusty knife, the useless lantern, and the skeleton."

From the adventure writer's standpoint, there's no more work to be done when the player uses multiple direct objects, because the parser breaks up the command into simpler commands. For example, if the player types, "Take the red book, the blue book, and the nasty knife," the parser pre-digests this, so the game sees three separate commands: "Take the red book," then "take the blue book," and then "take the nasty knife." (In fact, the game sees an even simpler representation, because the parser does even more work than this. The final digested form of a command is discussed later.)

Note that we've ended the last few commands with a period. This is optional, but it does tell the parser that the sentence is finished, which can be useful if the player wants to put several sentences into a single command:

```
Take everything; go east, then take the lamp and look at it!
```

Semicolons, exclamation marks, periods, question marks, and the word "then" all mean the same thing: the sentence is finished. Commas and the word "and," in addition to being useful for listing objects to a single command, can be used to separate commands as well, so long as the words that follow the comma or "and" can't be interpreted as another object.

This also brings up "it," and its partner "them." These words can be used to refer to the last direct object, or, in some cases, an object that the game referred to in its description of a place or event. (The built-in function `setit()` can be used by the adventure writer to define what "it" means in the next command.) "Them" works like "it," but refers to the last list of objects if multiple direct objects were used, as in, "Take the red book and the blue book and put them in the box."

In addition to "it" and "them," the parser understands "him" and "her" to refer to actors. If an object has the property `isHim` set to `true`, the parser allows the player to refer to the object as "him". If the object has `isHer` set to `true`, the player can use "her" to refer to the object. If neither `isHim` nor `isHer` is set to `true` for the object, only "it" can be used to refer to the object. Note that both `isHim` and `isHer` can be set to `true`, in which case either "him" or "her" can be used with the object. By default, both are `nil`.

There's one more issue, which is a bit specialized: what happens if the player uses quoted strings or numbers in the command? For example:

```
turn dial to 651
type "hello" on the computer keyboard
```

Clearly, it wouldn't be convenient for the game designer to list all possible numbers the player might dial, or all possible strings the player might type on the computer keyboard. Instead, such items are treated as special pseudo objects; these objects appear to the game as ordinary (but specific) objects, which have a special property which allows the game to learn the value (such as 651 or

"hello"). In this way, the game designer can deal with numbers and strings in a general way, letting the parser do all the work.

We've described the parser from the user's perspective, with some insights into how the game program sees commands. Now we'll describe the lower layers of the work the parser does.

Associating Words with Objects

Once the parser is through with a command, assuming all went well with the grammar of the sentence, the command has been digested into only five parts: an actor, a verb, a direct object, an indirect object, and a preposition. Of these, only the actor and verb are necessarily present; the others may be `nil`. All of these items are either `nil` or are an object defined by the game program.

The parser resolves words to objects based on vocabulary. The words that make up the vocabulary available to the user consist of the several built-in words (such as "and" and "then") and of the words defined in the game. A vocabulary word is created simply by associating a special property with an object:

```
RedBook:  object
  noun = 'book'
  adjective = 'red' 'large'
  plural = 'books'
;
```

In this object definition, the special property names `noun`, `adjective`, and `plural` are defined to be one or more words, enclosed in single quotes; these words become vocabulary that the player can use, and specify the context in which the words can be used. When a player uses the words "red book" in a sentence, the parser will know that he is referring to the object `RedBook`.

Verbs are defined similarly, with a slight catch. Recall that some verbs had a preposition built into them. These are defined like this:

```
TakeVerb:  object
  verb = 'take' 'pick up'
;
```

This associates the word "take" with the object `TakeVerb`, as well as the words "pick up". When a player types a command such as "pick it up", the parser will decide that the verb is the object `TakeVerb`. This notation, in which two words appear in a single vocabulary word string, is special to verbs and is not allowed elsewhere.

It is important at this point to distinguish between two types of objects. The first type of object is the kind that appears in the game program; these have superclasses and properties, and are manipulated by code. The other type of object is the kind the player thinks of when he says "take the book." So, when we call a verb an object, we mean it in the first sense, but when we talk about the user specifying an object with nouns and adjectives, we're using the other sense.

There are two other kinds of vocabulary words: prepositions and articles. For example:

```
OutP: object
  preposition = 'out'
;

AAnThe:  object
  article = 'the' 'a' 'an'
;
```

Note that the second object doesn't distinguish between the articles; this is perfectly all right, because they'll never make it down to the game program. They are defined merely to make the command syntax more natural for the user. Prepositions, on the other hand, should be distinguished from each other, because "put the book in the safe" is different from "put the book on the safe".

Note also that using a preposition in a verb doesn't make it a preposition; hence, even though we defined `'pick up'` as a verb earlier, we should still define a separate preposition for "up":

```
UpP: object
  preposition = 'up'
;
```

This brings us to ambiguity. Most adventures allow you to say "up" when you mean you want to "go up". Hence, we should define a verb:

```
GoUp:  object
  verb = 'go up' 'up'
;
```

This might look like a problem, because we've defined a word as a preposition, as a verb's preposition, and as a verb! Fortunately, the parser understands the importance of context and will not be confused in this case. This shouldn't be taken as license to define every word as every part of speech; the parser can get confused and interpret sentences strangely if contextual disambiguation is overused. If you run into strange interpretations of sentences, you might check for part-of-speech ambiguities.

The vocabulary properties, `verb`, `noun`, `adjective`, `article`, `preposition`, and `plural`, have special characteristics. They are not properties in the normal sense, for a number of reasons. First, they are specially recognized as creating vocabulary words and associating them with objects. Second, you might have noticed the non-standard syntax: a list of words follows the property name *without* being enclosed in square brackets. In addition, you cannot reference vocabulary properties as you can with other properties; for example, the following example will not work:

```
say(dobj.noun);                                    // This won't work!
```

For this reason, all objects which have vocabulary associated with them should have a short description, or `sdesc` property. So, the full definition of the "up" preposition above should be:

```
UpP: object
```

```
    preposition = 'up'
    sdesc = "up"
  ;
```

In constructing certain complaints about bad grammar or other problems, the parser will occasionally need the `sdesc` of a preposition, noun or verb, so be sure each object has one. Articles never need an `sdesc` as far as the parser is concerned. `sdesc` is a built-in property that the parser uses to display the short description of certain objects, but it, unlike vocabulary properties, behaves like a normal property.

Vocabulary words are normally made up solely of letters. Numbers are also allowed, so long as the word starts with at least one letter or is made up entirely of numbers. (Note that only adjectives can be made up entirely of numbers.) Hence, 'a123' is valid for any part of speech, and '123' is valid as an adjective. In addition, hyphens ('`-`') and single quotes are allowed within words, as long as the first character of the word is a letter. Finally, a word can have a period ('`.`') as its *last* character; this allows words such as "Mr." and middle initials in names to be used as vocabulary words.

In addition to `sdesc`, nouns should define the properties `thedesc` and `adesc`, which display the name of the object preceded by its definite article and indefinite article, respectively. In most cases, `thedesc` is simply the word "the" followed by the object's `sdesc`; likewise, `adesc` is "a" followed by the `sdesc`.

```
Thing:  object
  thedesc =
  {
    "the "; self.sdesc;
  }
  adesc =
  {
    "a "; self.sdesc;
  }
;
```

Note that `adesc` needs to be overridden for certain object. For example, objects whose `sdesc` starts with a vowel should use "an" rather than "a." Objects which represent collections or people or quantities of something often need a special `adesc` to make sense: "some milk" is better than "a milk," and "Lloyd" should be used rather than "a Lloyd."

Executing the Command

A command is decomposed into its actor, verb, direct object, indirect object, and preposition. Hence, a command such as "pick up the red book" might decompose as follows: the actor is `Me`, the player actor; the verb is `takeVerb`; the direct object is `RedBook`; the indirect object and preposition are `nil`.

So, what does the game do with this information? Once again, the parser helps the game designer by doing much of the work. First, the parser determines which objects are implicated in the command, then checks their availability with the `validIo` and `validDo` methods of the verb (see the discussion of disambiguation, below). If the objects can be successfully identified and are valid for the command, the parser sends a set of messages to these objects to inform them that they have been used in the player's command. The sequence of events is shown below in pseudo-code.

actor.roomCheck(*verb*)
actor.actorAction(*verb*, *direct object*, *preposition*, *indirect object*)
actor.location.roomAction(*actor*, *verb*, *direct object*, *preposition*, *indirect object*)
if (*An indirect object was specified*)
  direct object.verDo*Verb*(*actor*, *indirect object*)
  if (*No Output resulted from verDo*Verb)
    indirect object.verIo*Verb*(*actor*)
    if (*No output resulted from verIo*Verb)
      indirect object.io*Verb*(*actor*, *direct object*)
else if (*A direct object was specified*)
  direct object.verDo*Verb*(*actor*)
  if (*No output resulted from verDo*Verb)
    direct object.do*Verb*(*actor*)
else
  verb.action(*actor*)
Run each daemon that is currently active
Run and remove each fuse that has burned down

This may look complicated, but you don't generally have to worry about very much of it. Normally, you will only have to customize one or two of these steps to obtain a desired effect in your game, and let the standard definitions take care of the rest. Let's go through the sequence in detail.

The first step checks with the actor to ensure that the verb is even remotely allowed here, by calling the actor's `roomCheck` method with the verb object as its parameter. The `roomCheck` method returns `true` if the verb is acceptable, or `nil` if not. This check is intended to be the coarsest possible level of check; its purpose is to allow rooms with special conditions, such as darkness, to disallow all activity for most verbs. You will almost never override this method, as suitable definitions are already provided for all of the common types of rooms (see the "Adventure Definitions" appendix). Note that `roomCheck` is sent to the actor, which generally simply returns its location's `roomCheck` value.

Next, we inform the actor implicated in the command, via `actorAction`, that it's been asked to do something; the `Me` actor, the player, doesn't usually do much of anything here, but other actors would generally respond in one way or another to the request. In general, actors other than `Me` won't carry out arbitrary requests, as a matter of game design; this is their chance to object, by printing out a message (such as "The guard looks at you harshly and continues to block your way"), and then `exit`. When one of these routines executes an `exit` command, control passes down to the daemons and fuses, and everything else is skipped.

Assuming the actor agrees to the command and does not issue an `exit` command, the actor's location is informed of the event via its `roomAction` method. In general, the room doesn't care what goes on inside it, so the `roomAction` usually doesn't do anything. Some rooms, however, might restrict or change the actions performed by actors in them. A room with a trap, for example, could use `roomAction` to set off the trap if (or unless) certain commands are issued. As with `actorAction`, if the room simply wants to disallow the command, it can print a message and use the `exit` command.

The difference between `roomCheck` and `roomAction` is that `roomCheck` is called *before* any object disambiguation is performed. None of the objects associated with the command are known when `roomCheck` is called. `roomAction`, on the other hand, is called *after* the objects have been disambiguated. Both checks are needed. When the room is dark or the command will fail for some other reason, you don't want the parser offering a list of the objects that the command can act upon, since the player might not even be aware that some of the objects are present—so a check before disambiguation is needed. A check after disambiguation is also needed, since the room may need to take different actions depending on the exact objects used in the player's command.

If the room has no objection, we next inform the indirect and direct objects of the command. Generally, these will actually carry out the function of a command (whereas the actor and room methods will do little more than approve or reject the command). Depending on whether two objects, one, or none are specified in the player's command, a different series of messages is sent.

If no objects are in the command, the verb simply receives an `action` message; the actor is the only parameter, as it's the only relevant information. Verbs such as "north" and "look" take no objects, so their functionality is handled by their `action` routines.

If only a direct object is present, then there is no verb `action` method invoked. Instead, the direct object itself carries out the command.

The object, like the actor and room, gets a chance to object to the whole command; this is via its `verDo`*Verb* method, which stands for "verify direct object usage for *Verb*." For example, if the verb was "take," the `verDoTake` method will be invoked. (The actual name of this method is defined in the verb itself; the `doAction` property names the method that will be invoked. If `doAction = 'Take'` is specified in the `takeVerb` object, the method invoked will be `verDoTake`.) The verification method is special, because it can object to the command simply by printing a message—no `exit` command is necessary. The object verification routines are designed this way for convenience; since they don't actually carry out the action of the verb, the only reason it would want to display anything is to object, so the system detects this fact and treats it like an `exit` command.

Assuming the verification succeeds, the direct object's `do`*Verb* method is invoked. (As with the verification, the name of the method is derived from the verb's `doAction` property; if `takeVerb` has `doAction = 'Take'` defined, a "take" command will send a `doTake` message to the direct object.) This method is responsible for carrying out the function of the verb and reporting to the user on the command's success; for example, the `doTake` method may move the direct object to the user's possessions, and display "Taken."

When both a direct and indirect object are present, the verification method of the indirect object is called first; if that doesn't generate any messages, the verification method of the direct object is called, and if that doesn't generate any messages either, the `io`*Verb* method of the *indirect* object is

invoked. (Note that the `verIo`*Verb*, `verDo`*Verb*, and `io`*Verb* method names are derived differently from the way single-object command methods are derived. Each verb accepting multiple objects has an `ioAction` property for each preposition it accepts between the objects. For example, "put the book on the table" would only work if the `putVerb` object defines a property `ioAction(onPrep)`. If such a property is defined, its value is used as the *Verb* suffix in each method name. For example, if `putVerb` defines `ioAction(onPrep) = 'PutOn'`, then the messages sent are `verIoPutOn`, `verDoPutOn`, and `ioPutOn`.)

Once all of this processing is completed, all of the daemons are executed once, then all of the fuses that have burned down are executed and removed from the active fuse list. Note that messages to objects scheduled with the `notify()` function are treated the same as other fuses and daemons; hence, messages to objects scheduled with calls to `notify()` with *turns* set to zero are handled at the same time as other daemons, and other `notify()` calls are handled with the other fuses.

Note that the `abort` command causes the command to be entirely stopped, and not even the fuses or daemons are executed. It is generally used when a special system function is carried out, such as saving a game, which should not be counted as a turn.

Disambiguation

One of the more difficult parts of understanding the player's input is deciding which object is being referred to when several objects go by the same name. For convenience, the player does not want to have to spell out his desires in infinitessimal detail. For example, if the player is carrying a red book, and there is a blue book in the room, he should be able to say "take the book" without having to specify the "blue book." Likewise, while in an airlock with two doors, one of which is open, the command "close the door" should be sufficient, without needing to specify the open door. The process of deciding based on context which object the player is referring to when multiple objects use the same vocabulary words is called "disambiguation."

When a player refers to a "book," the parser will note all of the objects that share that vocabulary word. If the player is more specific, the system will consider only those objects which go by *all* of the words the player used. If multiple objects are still implicated, the parser performs two levels of disambiguation before asking the player for more detail. First, the system will eliminate from consideration those objects that don't pass the `validDo` or `validIo` test, as appropriate. If a unique object still doesn't result, the system will next apply a silent `verDo`*Verb* or `verIo`*Verb* test, as appropriate, to each object still being considered; the system suppresses the messages from these routines, if any, so the player doesn't see them. However, the parser does note whether output would have resulted; any objects that would produce output fail the test, and any that wouldn't pass the test. If no objects pass the test, all remain under consideration; if one or more objects pass the test, only those passing remain under consideration.

If, at this point, only one object remains, the parser decides that the player must mean this one object. If more than one object remains, the system will ask the player which one he meant, with a question such as "Which book do you mean, the red book, or the blue book?"

The idea behind using `validDo` and `validIo` is simple; if the object is not accessible to the player, he cannot meaningfully use it in a command, so it is discarded. This also simplifies later verification code, because it doesn't need to check that the actor in the command can access the objects mentioned.

The idea behind `verDo`*Verb* and `verIo`*Verb* is a little more complicated. Since these methods are supposed to issue a message if the object in question doesn't make sense—for example, if the player tries to take an object he is already carrying, or close a door that is not open—the parser can guess that the player probably didn't mean an object that fails these tests. So, if the player types "take the book," the red book, which the player is already carrying, fails the test and is eliminated, but the blue book, which is in the room waiting to be taken, passes the test. Likewise, if the player types "close the door," the door that is closed fails the test, while the open door passes.

Using this scheme, the system can generally make reasonably intelligent guesses about what the player meant to say. It is important to design your `verDo`*Verb* and `verIo`*Verb* routines with this function in mind, as well as with their primary function of ensuring that the object is valid with the verb.

Visibility and Accessibility

Sometimes, you want to make objects that are visible, but not accessible. For example, if a marble is inside a closed glass jar, the marble is visible to the player (since the glass jar is transparent, its contents can be seen), but the marble can't be manipulated without opening the jar.

To support this, the parser makes a distinction between objects which simply aren't visible, and objects which are visible but not accessible. The distinction is minor, and really only affects what kind of error message the parser displays when you can't access an object. When the marble is not visible and not accessible (such as when it's in a different room), the parser will display the message "I don't see any marble here." However, when the marble is visible but not accessible (such as when it's in a closed glass jar), the parser will call the method `cantReach(`*actor*`)` in the marble object.

The parser determines whether the marble is visible by calling its `isVisible(`*vantagePoint*`)` method. If the method returns `true`, the marble is visible; if it returns `nil`, the marble cannot be seen from the vantage point. The argument *vantagePoint* is the actor in the current command.

Note that certain verbs use `isVisible` to determine whether an object is valid or not (that is, the verb's `validDo` method is defined in terms of `isVisible`). For example, `inspectVerb` allows you to examine an object that you can't manipulate, as long as you can see it.

The default `isVisible` method (defined in `adv.t` for the object `thing`) is somewhat complicated, but the basic rule is: return `true` if the object's location is the vantage object, or if the location's contents are visible and its location is visible from the vantage object, or if the vantage has a location and the vantage's location's contents are visible and the object is visible from the vantage's location; otherwise return `nil`. Stated more simply, this is the first law of optics: if you can see me, then I can see you. (Being a computer program, TADS turns this easily understood concept into a complicated recursive algorithm.)

The default `isVisible` will suffice for most occasions, as will the default `cantReach` method. You can also use new definitions of `isVisible` to achieve special effects; for example, you could use it to implement a view screen that shows objects in another location; you can look at the objects on the view screen, but you can't otherwise manipulate them.

Default Objects

The special properties `doDefault` and `ioDefault` can be used to supply default information for commands. The purpose of `doDefault` is to return a list of all objects which are sensible with the current command. For example, if the command is "take," all the objects which are immediately accessible to the actor, excluding, of course, those the actor is already carrying, apply to the command. Likewise, the default direct objects to "drop" are those that the object is carrying.

When a player types a command without a direct object, the parser looks at `doDefault` for the verb. It then applies the disambiguation function appropriate to the verb to the resulting list, making up a new list of those objects passing the disambiguation function. If there is exactly one direct object remaining in the list, the parser decides that the player is referring to the one sensible object, so it supplies it in the command by assumption (displaying the assumption to the player). The `ioDefault` property does the same for indirect objects.

For example, if the player types merely "take," the system will first determine that an object is needed because `takeVerb` has no `action()` method. It will then consult `takeVerb` to see if it has a default preposition given by `prepDefault`; if it does, it assumes that this will be a two-object verb and adds the default preposition to the command. In the case of "take," there will be no default preposition, so the system knows this will be a one-object command. Next, the appropriate `verDo`*Verb* routine is identified based on the verb and preposition. In this example, `verDoTake` is the chosen routine. Now the system calls the `doDefault` method in `takeVerb`, and then applies `verDoTake` to each object returned. Each object that passes the `verDoTake` test (i.e., produces no output) is kept under consideration. If only one object is left, the object becomes the default object for the command. If no objects, or more than one object, are left in the list, the system will ask the user what he wants to take.

"All"

Another use of `doDefault` is in determining the list of objects meant by "all" in a player's command. "All" is assumed to refer to everything in the list generated by `doDefault`.

For most verbs, the standard `doDefault` and `ioDefault` methods inherited from `deepverb` will be sufficient (refer to the "Adventure Definitons" Appendix for information on the `deepverb` class). `takeVerb` and `dropVerb`, because they are such common cases, customize the `doDefault` method to be a little smarter; this allows "take all" to ignore objects already being carried, and "drop all" to ignore objects that are not being carried.

Object-Level Verb Synonyms

It is often desirable to make certain verbs synonymous for a particular object. For example, if you have a touch-pad on a control panel, the player may try to push the touch-pad instead of merely touching it. With the old system, you had to tediously point the `verDoPush` and `doPush` methods to call the object's `verDoTouch` and `doTouch` methods, respectively:

```
verDoPush(actor) =
{
  self.verDoTouch(actor);
}
doPush(actor) =
{
  self.doTouch(actor);
}
```

A new convenience feature has been added that will do this for you automatically. The `doSynonym` and `ioSynonym` pseudo-properties allow you to specify a list of property root names that are to be synonymous with a particular property root name. For the above example, we would now write this:

```
doSynonym('Touch') = 'Push'
```

This means that whenever a `verDoPush` or `doPush` method is called for this object, the object's `verDoTouch` or `doTouch` method (respectively) is called instead. You can specify more than one synonymous root name (on the right-hand side of the equals sign); for example, if you also wanted to make `verDoTap` synonymous with `verDoTouch`, and `doTap` synonymous with `doTouch`, you'd change the definition above to this:

```
doSynonym('Touch') = 'Push' 'Tap'
```

The `ioSynonym` works the same way.

Built-in Properties

The section on the player command parser described several special built-in property names that the parser uses. This section lists all of the special property names that the TADS run-time system defines, and explains how each is used.

action

`action(`*actor*`)`: the method of a verb invoked when the verb is used without objects. The method should carry out the action of the verb, checking first to ensure that the verb should be allowed at

this time. With system verbs, such as `save`, the command generally shouldn't count as a turn, so it should use the `abort` keyword even if completes successfully, to prevent the per-turn daemons and fuses from running. Non-system verbs, such as `jump`, shouldn't use `abort` if they complete successfully.

---

actorAction

`actorAction`(*verb*, *direct object*, *preposition*, *indirect object*): sent to the actor in a command by the parser. This method should determine if the actor is interested in accepting the command. If so, it needn't do anything, since the command will be handled by the verb or objects as usual; if not, it should display a complaint and use `exit` to prevent further execution of the command.

---

adjective

`adjective`: vocabulary list of the adjectives that can refer to an object.

---

article

`article`: vocabulary list of article words (such as "the" and "a").

---

cantReach

`cantReach`(*actor*): display an appropriate message when the *actor* can't reach the object (`self`). The parser calls this method to display a message when an object is visible, but not accessible (for example, when a marble is inside a closed glass jar, the marble is visible, but can't be manipulated). The default version defined `adv.t` for the `thing` object is suitable for most situations.

---

contents

`contents`: a list of objects that an object contains. This should not be initialized; see the discussion of the `location` property. However, it should be kept current; that is, whenever an object's `location` property changes, its old container's `contents` list should have the object removed, and the new container's `contents` list should have the object added.

---

doAction

`doAction`: the property of a verb that defines the suffix applied to `verDo` and `do` to form the method name sent to the direct object when the verb is used with only a direct object. This must be a single-quoted character string. If not defined, the verb is not allowed to be used with just one object. For

example, if the "take" verb defines `doAction = 'Take'`, then `verDoTake` and `doTake` messages are sent to the direct object of a "take *object*" command.

---

doDefault

`doDefault`(*actor, prep, indirectObject*): the default direct objects for a verb. When "all" is used, this entire list is substituted for "all." When no direct object is used at all, if this list is unambiguous (i.e., has only one entry), it is used by default as the direct object of the command.

---

ioAction

`ioAction`(*preposition*): the property of a verb that defines the suffix applied to `verIo`, `verDo`, and `io` to form the method name sent to the indirect object when the verb is used with two objects and the given preposition. The value must be a single-quoted character string. For example, if the "ask" verb defines `ioAction(aboutPrep) = 'AskAbout'`, then `verIoAskAbout` and `ioAskAbout` messages are sent to the indirect object of a "ask *direct-object* about *indirect-object*" command; a `verDoAskAbout` message is sent to the direct object. Note that the preposition argument is the name of a preposition object defined elsewhere; the object defines the vocabulary word, with its `preposition` attribute, that the player uses.

---

ioDefault

`ioDefault`: the default indirect objects for a verb.

---

isHim

`isHim`: Returns `true` if the object (`self`) is a male actor and should therefore be referred to with "him" in player commands. If `isHim` is `nil`, the object cannot be referred to as "him". It is legal to set both `isHim` and `isHer` to `true`, in which case either "him" or "her" can be used. If both are `nil`, only "it" can be used with the object (this is the default).

---

isHer

`isHer`:

`isHer`: Returns `true` if the object (`self`) is a female actor and should therefore be referred to with "her" in player commands. If `isHer` is `nil`, the object cannot be referred to as "her". It is legal to set both `isHim` and `isHer` to `true`, in which case either "him" or "her" can be used. If both are `nil`, only "it" can be used with the object (this is the default).

isVisible

`isVisible(`*vantage*`)`: returns `true` if the object (`self`) is visible from the object *vantage*, or `nil` if it is not visible. The implementation defined in `adv.t` in the `thing` object is suitable for situations involving transparent containers; different implementations may be useful for achieving special effects.

location

`location`: the container of an object. May be `nil` or another object. At the start of a game, all objects are moved into their location; that is, for all objects with a `location` defined as a simple object (not a method or `nil`), *object*`.location.contents` has the object added to it. This ensures consistency between `location` and `contents` properties. Note that `class` objects and objects with methods for their `location` property are ignored in this procedure. **Warning: Never assign the `location` property of an object directly at run-time.** Always use the `moveInto(`*newlocation*`)` method instead. For example, if you want to move the knife into the player's current location, do *not* code `knife.location := Me.location`, but do this instead: `knife.moveInto(Me.location)`. The `moveInto` method ensures that the `contents` properties of the old and new containers of the object are updated to reflect the object's new location.

locationOK

`locationOK`: a flag that specifies that the `location` of an object is intentionally something other than an object. Normally, if an object's `location` property is not an object, the compiler flags this as a warning, since the compiler is unable to set the `contents` list of the `location` to contain the current object. However, it is sometimes necessary to make `location` a method to achieve special effects; for example, `theFloor`, defined in `adv.t` (see the "Adventure Definitions" Appendix for details), uses a method for its `location` so that the same object appears to exist in every room. By setting `locationOK` to `true` in `theFloor`, the game informs the compiler that the non-standard `location` value is intentional, which suppresses the warning message. This property has no other effect.

noun

`noun`: vocabulary list of the nouns that can refer to an object.

plural

`plural`: a vocabulary list giving the plurals by which an object may be called.

prepDefault

`prepDefault`: specifies the default preposition for a verb. Used when no `doAction` property is defined, but the player specified only a direct object in his command. The parser will attempt to add the default preposition given to the command and then try to get an indirect object (first from the verb's `ioDefault` list, then by asking the player).

preposition

`preposition`: vocabulary list of preposition words.

roomAction

`roomAction`(*actor*, *verb*, *direct object*, *preposition*, *indirect object*): sent to the object given by the actor's `location` property for all commands. See the section on the sequence of messages sent by the parser in response to a player command.

roomCheck

`roomCheck`(*verb*): sent to the `Me` object (the player actor) at the very beginning of processing a sentence. The property should return `true` if the *verb* is acceptable in the player's room, `nil` otherwise; it should also display a message in the latter case explaining why the verb is not acceptable. This property is used largely to prevent unnecessary processing from taking place in a dark or otherwise limiting room, and it should do nothing apart from checking the most basic features of the room. Note that `roomCheck` is called *before* disambiguating the objects in the command, so it must decide whether the command is appropriate or not regardless of objects; `roomAction` is called *after* the objects are known, so it can apply more specific criteria to the command.

sdesc

`sdesc`: a self-printing short description. May be a double-quoted string, or a method that prints a short description.

statusLine

`statusLine`: this is related to `sdesc` for rooms or other objects (such as chairs and vehicles) that may contain the player. This should print the short description of the location, which should fit on a single line, without the list of objects in the location. Most locations that are descendants of

the `room` object defined in `adv.t` will display `statusLine` first when sent a `lookAround` message (a method defined in `adv.t` for `room` objects).

On some platforms, the TADS run-time system displays a status line across the top of the screen showing the player his current location; this line is updated before every turn. On these platforms, the run-time system uses the `statusLine` method of the player's current location to display the status line information.

Note that `statusLine` prints only the "left half" of the status line, which typically gives the player's current location. The right half is displayed by the `setscore` function.

---

thedesc

`thedesc`: a short description with a definite article. Generally just defined as a method showing "the" and the `sdesc`. Like `sdesc`, the `thedesc` must be self-printing.

---

validDo

`validDo(`*actor*`,` *object*`)`: sent to a verb to check that a direct object is valid for a command. The method should return `true` if the object is valid as a direct object for the verb and actor, or `nil` if it is not. Generally speaking, this routine should do nothing more than check that the object is accessible to the actor; it is *not* necessary for `validDo` to check whether the object makes sense with the verb. Hence, it should return `true` if the actor is trying to take something he is already carrying, while it should return `nil` if trying to take something that is in a different room. The job of deciding whether the object makes sense is done by the object's `verDo`*Verb* method.

---

validIo

`validIo(`*actor*`,` *object*`,` *seqno*`)`: sent to a verb to check that an indirect object is valid for a command. Like `validDo`, the method returns `true` if the object is valid as an indirect object for the verb and actor, or `nil` if it is not. As with `validDo`, this method is responsible only for verifying accessibility, not for checking if the object makes sense with the verb; hence, if the actor is trying to put something into an object that is accessible, the method should return `true` regardless of whether the object in question is a container or not.

The *seqno* parameter is the "sequence number" of the object being tested. Any time the player makes an ambiguous object reference (that is, uses a set of nouns and adjectives that could refer to more than one object), the possible objects are tested with `validIo` one at a time; each object in the set is given a separate sequence number, starting with 1. These sequence numbers are arbitrary, and are generally not even used. However, certain verbs don't care which of a set of ambiguous objects is chosen; these verbs use *seqno* to accept only the first object and reject the rest. For example, `askVerb` in `adv.t` does this (see the "Adventure Definitions" Appendix for information on `adv.t`).

value

`value`: A special property for the `strObj` and `numObj` objects giving their value when used in a player command. For example, "turn the dial to 651" makes `numObj` the indirect object, and sets its `value` property to 651.

verb

`verb`: list of the vocabulary words for a verb object. Like all vocabulary properties, the value must be a list of single-quoted strings, optionally enclosed in square brackets. (If the object is a `class` object, the square brackets are required.) The special two-word notation, as in `verb = 'pick up'`, may be used for verbs with a prepositional part.

Required Objects and Functions

The TADS run-time system requires the game program to define several objects and functions. This allows the user control over the details of these objects, but their names are fixed by the system. The required objects and functions are listed here.

againVerb

`againVerb`: a special verb that specifies the vocabulary for the "again" command, which is used to repeat the last command. Normally, the `verb` property of this object is set to accept "g" and "again" for this command.

init

`init()`: A function that is executed once when the game is started. This function should move the `Me` actor (the player) into his starting room, start any daemons and fuses that are needed, and display the title, copyright message, and introductory text for the game.

See also the related `preinit()` function, which should be used for any time-consuming computations that need to be performed before the game starts.

Me

`Me`: The player's actor. When no actor is specified in a player command, the `Me` object is assumed. The `Me` object receives all of the same messages from the parser that any other actor does.

Note that `adv.t` defines the class `basicMe` as a reasonable implementation of `Me`. If you do not wish further customization of `Me` in your game, you can use `basicMe` unchanged by including this in your game: "`Me: basicMe;`".

---

numObj

`numObj`: The special object used when a command involves a number; its property `value` will be set to the number the player used in the command. For example, if the player types "Turn the dial to 617," the indirect object will be `numObj`, and `numObj.value` will be 617.

Note that `adv.t` defines the class `basicNumObj` as a reasonable implementation of `numObj`. If you do not wish further customization of `numObj`, you can use `basicNumObj` unchanged by including this in your game: "`numObj: basicNumObj;`".

---

pardon

`pardon()`: A special function that is called whenever the player enters a blank line. It normally just displays "I beg your pardon?" The function is up to the game author to implement simply to allow customization of this message.

---

parseError

`parseError(`*errnum, errstr*`)`: This function is called whenever a parser error message is to be displayed. This function is optional; if it's not defined in your game, the system will display a default message. This function allows you to change any parser error message and replace it with your own message.

The argument *errnum* is a number that specifies which message is needed; *errstr* is the text of the default message that the parser would normally display. The reason that the number is provided is that it serves as a version-independent key to the error message; even if a small change is made to the actual text of a message, its basic meaning will remain the same from version to version, so you can count on *errnum* to indicate the meaning of the message. The reason that *errstr* is provided is to allow you to make a small textual change to the message without replacing it altogether; for example, if you want parser messages to be enclosed in square brackets, you could simply append square brackets to *errstr* and return the resulting string.

This function returns either `nil` or a string value. If it returns `nil`, the default message is displayed. If it returns a string value, the string is displayed instead of the default message.

The error numbers and default messages are shown below. Errors in the range 1-99 are normal parser error messages. Numbers 100-109 are special messages that are used to construct disambiguation queries; several error numbers make up a single error display, so you should probably exclude these if you are performing a transformation such as adding parentheses around error messages. The same is

true of the other ranges above 100. 200 is a special message that is used when a cantReach message is about to be generated. Here is an example of a `parseError` function that simply encloses all normal parser error messages in square brackets, leaving the special messages (those numbered 100 and above) unchanged:

```
parseError:  function(errno, str)
{
  if (errno >= 100) return(nil);              /* leave special messages alone */
  return('[' + str + ']');                   /* enclose message in [brackets] */
}
```

Note that the C-style formatting sequences `%c` and `%s` are used in these messages. You can leave out the formatting sequences in your messages if you wish, but you should never include one in a message that doesn't have one in the default version, and you should always make sure that they are of the correct number, type, and order if you do provide them. Note that you don't need to do anything with the formatting sequences—just preserve any sequences that are present, and the system will fill in the correct values before displaying your message.

Normal error messages

1 : "`I don't understand the punctuation "%c".`"

2 : "`I don't know the word "%s".`"

3 : "`The word "%s" refers to too many objects.`"

4 : "`I think you left something out after "all of".`"

5 : "`There's something missing after "both of".`"

6 : "`I expected a noun after "of".`"

7 : "`An article must be followed by a noun.`"

8 : "`You used "of" too many times.`"

9 : "`I don't see any %s here.`"

10 : "`You're referring to too many objects with "%s".`"

11 : "`You're referring to too many objects.`"

12 : "`You can only speak to one person at a time.`"

13 : "`I don't know what you're referring to with '%s'.`"

14 : "`I don't know what you're referring to.`"

15 : "`I don't see what you're referring to.`"

16 : "`I don't see that here.`"

17 : "`There's no verb in that sentence!`"

18 : "`I don't understand that sentence.`"

19 : “There are words after your command I couldn’t use.”

20 : “I don’t know how to use the word "%s" like that.”

21 : “There appear to be extra words after your command.”

22 : “There seem to be extra words in your command.”

23 : “internal error: verb has no action, doAction, or ioAction”

24 : “I don’t recognize that sentence.”

25 : “You can’t use multiple indirect objects.”

26 : “There’s no command to repeat.”

Special disambiguation error messages

100 : “Let’s try it again: ”

101 : “Which %s do you mean, ”

102 : “, ”

103 : “or ”

104 : “?”

Special: used to complain that an object is no good for a verb

110 : “I don’t know how to ”

111 : “ ”

112 : “ anything ”

113 : “to”

114 : “ ”

115 : “.”

Special: used after each object when multiple objects are used

120 : “: ”

Special: used to note objects being used by default

130 : “(”

131 : “)”

132 : “ ”

Special: used to ask for an object when one can't be defaulted

140 : “`What do you want to `”

141 : “ `it` ”

142 : “ `to` ”

143 : “?”

Special: used when an obj is unreachable: obj.sdesc “: ” obj.cantReach

200 : “: ”

---

preinit

`preinit()`: This function is related to `init`. Immediately after successfully compiling your program, the compiler will call `preinit()`; the binary file containing the compiled game is then written.

`preinit()` is provided solely for performance; if the function is not defined in your program, the compiler will display a warning message but will not be otherwise affected. Since the function is called before the binary file is written out, it allows you to perform any pre-game compilations before the player runs your game. The `init()` function, on the other hand, runs after the player has loaded the binary file, so the player must wait for any computations the `init()` function performs to complete before the game can begin.

Note that `preinit()` should not display any messages, or set any fuses, daemons, or alerts, because the game has not begun yet, and any such actions would be lost when the binary file is written. Instead, the `preinit()` function is intended to be used to set up any lists of objects you need to compute at the start of the game (for example, a list of light-providing objects could be compiled by searching through all objects, using `firstobj()` and `nextobj()`, and storing in a list all of those objects with an `islamp` property set to `true`).

---

preparse

`preparse(`*str*`)`: This function is called immediately after the player enters a line of text, and before the player command parser starts to look at the command line. The argument is a string containing the complete, untouched text of the player's command line; the text is intact, complete with the original spaces, punctuation, and capitalization.

If the function returns `nil`, the command is aborted—the parser never sees the command. If the function returns a string, the new string replaces the original text of the player's command, and is processed by the parser as though the player typed it. If the function returns `true`, the original player's command is parsed as normal.

See the beginning of this chapter for more information on `preparse`. This function is optional; if it's not defined, the compiler will issue a warning, but the game will compile successfully.

---

strObj

`strObj`: When a player uses a quoted string in a command, this special object is used, and its property `value` is set to the string the player used in the command. For example, if the player enters `type "launch" on the console`, the direct object is `strObj`, and `strObj.value` is set to the string `'launch'`.

Note that `adv.t` defines the class `basicStrObj` as a reasonable implementation of `strObj`. If you do not wish further customization of `strObj`, you can use `basicStrObj` unchanged by including this in your game: "`strObj: basicStrObj;`".

---

takeVerb

`takeVerb`: the verb used to take objects. This is no different from any other verb, and the action of the verb must be implemented as normal by the game program, but the parser must know about this verb, so its name is required to be `takeVerb`. In general, it is recommended that verbs and other objects follow this general style of naming, so this requirement will fit smoothly into most games. (The reason the parser needs to know about this verb is that it uses its `validDo` method to determine if an actor is present and thus can be addressed.)

The Output Formatter

To relieve the game programmer of the details of how many characters to put on a line, the TADS run-time system automatically filters output with a formatter. The functioning of the formatter is generally transparent to the programmer, but once in a while a special effect is desired; special control sequences are recognized by the formatter which give the programmer full control of output formatting.

When the TADS compiler reads the text from your source file, it ignores redundant spaces, and treats line breaks as spaces. Hence, when your text is output by the system, it will be rearranged from the way it was input. This frees you of the need to keep your source text formatted nicely.

Normally, the formatter will simply fill up each line with as many words as it can, and then go to a new line. It will automatically "word wrap" the output; that is, line breaks will always occur between words. Hence, you can write your text without any line breaks at all, and it will come out looking reasonably good. The formatter will also break text at hyphens that appear in your text when it would allow the line to be filled with more text (although the formatter will never attempt to hyphenate words that aren't hyphenated in your text). When writing text, the formatter will

automatically insert two spaces after periods, exclamation marks, colons, and question marks, as is conventional. In other places, only one space will appear.

Note that you can override the automatic insertion of double spaces by using a quoted space (see the Language Reference chapter's explanation of special control sequences in text) after the punctuation mark that would normally cause a double space to be inserted. You can also insert several spaces in a row using the quoted space. Refer to the Language Reference chapter's section on double-quoted strings for the other special control sequences that you can use to control the output formatter.

Format Strings

TADS doesn't treat the player object specially in most places. This allows you to implement other actors that carry out orders from the player fairly easily: the player can specify an actor to which to direct a command, and if the actor follows it, the actor is specified in all of the parser's method calls to the objects involved in the command.

In order to make the text displays that result from a non-player actor's actions make sense, all of the messages have to be phrased in terms of the actor actually carrying out the command, rather than "you." It would be extremely inconvenient if you had to use `actor.thedesc` constantly in these messages, so TADS provides a much more convenient mechanism called "format strings."

A format string is a special sequence of characters that signals the output formatter to substitute a property of the current command's actor. Instead of using "you" and related words, the messages in `adv.t` use format strings. Your game program should do the same thing where appropriate (i.e., any place that an actor other than the player may be taking some action or described in a message). When the output formatter sees the format strings, it automatically replaces them with a term appropriate to the current actor.

You can define your own format strings, although `adv.t` defines most of the format strings you'll probably need. Basically, a format string associates a special output sequence with a property name. When the format string, enclosed in percent signs (%), appears in a message being displayed, the output formatter removes the format string (including the percent signs) and replaces it by calling the property associated with the format string. For example, `adv.t` defines a format string as follows:

```
formatstring 'you' fmtYou;
```

This tells the system that whenever it sees the sequence:

```
%you%
```

in text to be displayed, it removes the entire sequence, and replaces it with the text that results from evaluating the `fmtYou` property in the current actor.

In the new `adv.t`, messages are phrased in terms of these format strings. For example, the new movement error message is "`%You% can't go that way.`" When the player just types "east," the `%You%` is replaced by `You`, which is the result of evaluating `fmtYou` for the object `Me`, so the message is "You can't go that way," just as before. However, when the player types "joe, east," the message is now "Joe can't go that way."

Note that the capitalization of the text replacing the format string follows the capitalization of the format string itself. Hence, `%you%` is replaced with `you`, while `%You%` is replaced with `You`.

In `adv.t`, format strings are defined for most of the words that you'll need to use in messages that are dependent on the actor. Some of these words are dependent on the actor in order to keep sentences grammatical; for example, "You aren't wearing your spacesuit" would change to "Joe isn't wearing his spacesuit"—the words "you", "aren't", and "your" all need to change. Using the format strings defined in `adv.t`, you'd rewrite the message sentence as:

```
"%You% %are%n't wearing %your% spacesuit."
```

Your game will continue to work without using format strings in your own messages. You need not make use of this feature (although you will automatically use it, to some extent, because `adv.t` uses it in all of its messages). However, it's quite easy to use format strings instead of hardcoded text, and it makes your game quite a lot more adaptable to multiple actors.

Special Words

The parser treats a number of words as special. In older versions of TADS, these words were built in to the system, so your game program had no way to change them. A new construct has been added that allows you to replace these words with words of your own choosing (this may be useful if you are writing an adventure in a language other than English, for example).

The command that lets you override the built-in words is called `specialWords`. This command can appear anywhere that one of the other special commands (such as `formatString` or `compoundWord`) could appear. Following the `specialWords` keyword is a list of all of the special words, in a fixed order. You must specify the words in the order shown below, and you must provide at least one word for every position. However, you may provide more than one word for each position. Each position is separated by a comma; synonyms (multiple words for one position) are separated by an equals sign; and the entire list is terminated by a semicolon. The default list, which matches the built-in list if no `specialWords` list is used at all, looks like this:

```
specialWords
'of',                              /* used in phrases such as "piece of paper" */
'and',                  /* conjunction for noun lists or to separate commands */
'then',                                 /* conjunction to separate commands */
'all' = 'everything',                  /* refers to every accessible object */
'both',                           /* used with plurals, or to answer questions */
```

```
'but' = 'except',                          /* used to exclude items from ALL */
'one',                                  /* used to answer questions:  "the red one" */
'ones',                                   /* likewise for plurals:  "the blue ones" */
'it',                                   /* refers to last single direct object used */
'them',                                       /* refers to last direct object list */
'him',                                  /* refers to last masculine actor mentioned */
'her'                                    /* refers to last feminine actor mentioned */
;
```

When a thought takes one's breath away,
a lesson on grammar seems an impertinence.

— Thomas Wentworth Higginson (1890)

A smile is the chosen vehicle for all ambiguities.

— HERMAN MELVILLE, *Pierre,* book IV (1852)

Chapter Five

Language Reference

This section describes in detail all of the keywords, operators, syntactic elements, built-in functions, and special considerations of the TADS language.

Typographical Conventions

This section uses abstract syntax diagrams to describe the language. In these diagrams, several typographical conventions are used:

italics are used to represent a description of an item that must be replaced with an actual instance of the item.

[Square Brackets] are used to indicate that the enclosed item or items are optional.

`Typewriter` text is to be used verbatim. The same applies to any other punctuation shown.

Components of TADS

The TADS development system includes a compiler, a run-time system, and several TADS source files containing basic adventure definitions.

When you write a TADS program, you use a text editor (which is not included with TADS) to prepare a source file, then you compile the file. The compiler checks the entire program for syntax errors, then creates another file containing a binary representation of your source file; this form is more efficient to execute, so your game will require less memory and run faster.

After you have compiled your program, you use the run-time module to execute the game. The run-time system contains the player command parser and other code for user interaction.

Input File Format

The TADS compiler accepts standard ASCII text files as input. The compiler considers any amount of whitespace to be the same as a single space; spaces, tabs, and newlines all count as whitespace. Whitespace can occur between any two syntactic elements, but need only appear between two identifiers or keywords that have no other intervening punctuation. The compiler knows that punctuation is never part of a keyword or identifier, and can manage to break apart punctuation when it's all run together without spaces. For readability, liberal use of whitespace is recommended.

Including Other Files

The `#include` directive allows one file to insert the contents of another:

```
#include "file.t"
#include <file.t>
```

Here, `file.t` is the name of an operating system file to be included. An included file can include another, and so forth, down to ten levels of nested files. Note that the pound sign, `#`, must be in the first column of the line; no leading spaces are allowed. The include file `adv.t` is normally included at the start of a game file to define the many items that are needed in a normal adventure. These definitions are quite general, but specific games might modify them to customize the game.

Note that when the filename is enclosed in double quotes, the compiler searches for the file first in the current directory, then in the directories in the include path (as set with the `-i` compiler switch on most operating systems). When the filename is enclosed in angle brackets, the compiler searches only in the directories in the include path—the current directory is *not* searched. Generally, a system include file (such as `adv.t`) should be enclosed in angle brackets, while your files should be in the current directory and enclosed in double quotes.

Multiple Inclusions of the Same File

The TADS compiler automatically keeps track of each file you have included, and ignores redundant `#include` directives. Note that this feature is based on the filename as specified in your program; if you refer to the same file by different names (for example, by specifying a path on one inclusion, but using angle brackets to let TADS find the file on the second), the compiler will not be able to identify the duplicate file, and will include it twice, resulting in countless unnecessary errors. For this reason, you are encouraged to use angle brackets to specify include files, rather than using specific paths in your source file. Doing so will help ensure that your source files are portable to different computers, as well, since each operating system uses its own conventions for specifying file paths.

Note that the include files that make up a pre-compiled binary file, loaded with the compiler's `-l` option, are recorded in the binary file. Thus, there is no need for you to remove the `#include`

directive in your source file just because you are using the pre-compiled version of that include file. For more information on using pre-compiled include files, see the section on using the compiler.

Comments

Outside of a quoted string, two consecutive slashes, `//`, indicate that the rest of the line is a comment. Everything up to the next newline is ignored.

Alternatively, C-style comments can be used; these start with `/*` and end with `*/`; this type of comment can span multiple lines.

Examples:

```
// This line is a comment.
/*
This is a comment
which goes across
several lines.
*/
```

Identifiers

Identifiers must start with a letter (upper or lower case), and may contain letters, numbers, dollar signs, and underscores. Identifiers can be up to 39 characters long. Upper and lower case letters are distinct.

Scope of Identifiers

All objects and functions are named by global identifiers. No identifier may be used to identify different things; that is, no two objects can have the same name, an identifier naming a function can't also be used for an object, and so forth.

Property names are also global identifiers. A name used for a property can't be used for a function or object, or vice versa. However, unlike functions and objects, the same property name can be used in many different objects. Since a property name is never used alone, but always in conjunction with an object, the TADS compiler is able to determine which object's property is being referenced even if the same name is used in many objects.

Function arguments and local variables are visible only in the function in which they appear. It is permissible to re-use a global identifier as a function argument or local variable, in which case the variable supersedes the global meaning within the function. However, this is discouraged, as it can be a bit confusing.

Object Definitions

A basic object definition has the form:

identifer: `object` *[property-list]* ;

This defines an object which has no superclass. An object can be defined as having a superclass with the alternative form of object definition:

identifier: *class-name [, class-name [...]] [property-list]* ;

Here, the *class-name* is an identifier which is defined elsewhere in the program as an object or class (either with or without a superclass), and is the new object's superclass; if more than one superclass is present, a comma separates each superclass in the list. The new object named by *identifier* inherits all the properties of the superclass or superclasses. If a property in the optional *property-list* is also in the property list of the superclass (or its superclass, and so forth), the new property overrides the inherited one.

If a property is inherited from more than one of its superclasses (and is not overridden in the object's own property list), the property is inherited from the superclass that appears earliest in the list. For example, suppose you define an object like this:

```
vase:  container, fixeditem
;
```

If both `container` and `fixeditem` define a method named `m1`, and `vase` itself doesn't define an `m1` method, then `m1` is inherited from `container`, because it appears earlier in the superclass list than `fixeditem`.

There is a more complicated case that can occur, but it is very unusual. You will probably never encounter this, so skip this section if you find it confusing. Suppose that in the example above, both `container` and `fixeditem` have the superclass `item`, and that `item` and `fixeditem` define method `m2`, and that neither `container` nor `vase` define `m2`. Now, since `container` inherits `m2` from `item`, it might seem that `vase` should inherit `m2` from `container` and thus from `item`. However, this is not the case; since the `m2` defined in `fixeditem` overrides the one defined in `item`, `vase` inherits the `m2` from `fixeditem` rather than the one from `item`. Hence, the rule, fully stated, is: the inherited property in the case of multiple inheritance is that property of the earliest (leftmost) superclass in the object's superclass list that is not overridden by a subsequent superclass.

Property Lists

A property list takes on this form:

property-definition [property-list]

(This is a formal way of saying that you can string together any number of property definitions in a property list, one after the other.) A property definition looks like this:

identifier = *data-item*

Note that no semicolon comes after the property definition. A semicolon terminates the entire object definition, not individual property definitions.

Property Data Items

A data item can be a number, a double-quoted string, a single-quoted string, a list, an object, `nil`, or `true`.

Numbers

A number is just a string of digits. The default base is decimal; you can also enter octal and hexadecimal numbers. An octal number simply starts with a leading zero, so `035` is an octal number having the decimal value 29. A hexadecimal number starts with `0x`, as in `0x3a9`.

Numbers can vary from -2147483647 to 2147483647 (decimal), inclusive. Only integers are allowed; numbers cannot have a decimal point or a fractional part.

Double-quoted Strings

A double-quoted string is an arbitrary string of characters enclosed in double quotation marks, such as,

```
"This is a double-quoted string."
```

Stretches of whitespace in such strings are compressed to single spaces. This includes newlines; double-quoted strings can go on past the end of a line, and keep going for several lines. For example, this is perfectly legal:

```
"This is a string
that goes on for

several lines."
```

Note that TADS converts all of that blank space, including the blank lines, down to a single space between each word. Thus, you can enter your text without having to worry about formatting it; TADS will do all the work of filling up each output line when your program runs.

Sometimes, you will want to format your output in a specific way, overriding the standard output formatting. Since TADS converts all whitespace in your strings (including newlines) to spaces, you have to specify any special formatting you want explicitly. TADS provides several special character sequences you can use to obtain these effects.

`\t` Tab to the next stop. Tab stops are every four spaces. This is useful if you want stretches of blank space within a line.

`\n` A newline. Ends the current line, and skips to a new one. Note that repeating this sequence has no effect, since the output formatter ignores redundant `\n` sequences; this is normally convenient, because it means you don't have to worry about several newline sequences being displayed by different objects filling the display with unwanted blank lines. If you want to force one or more blank lines, use `\b`.

`\b` Ends the current line, and then outputs a blank line.

`\"` A double quote.

`\\` A backslash.

`\<` A left angle bracket. This is not required to produce a single angle bracket (although it will do so), but you will need to use this on second and subsequent angle brackets in a stream of contiguous angle brackets, to prevent TADS from thinking you want to embed an expression in your string (see below). Hence, if you want to display `<<<<<`, you will have to type this in your program: `"<\<\<\<\<"`.

`\^` (That is, a backslash followed by a circumflex or "up arrow" or "hat," entered with the shifted "6" key on most keyboards.) This sequence causes the next character in the output to be capitalized. This sequence is generally used just before invoking a function or method that will display text that will be at the start of a sentence. By using this sequence prior to the text, you ensure that the first letter of the sentence is capitalized, even if the function or method displaying the text doesn't know it should be capitalized. For example: `"\^<< obj.adesc >> is sitting on the table."` Note that the `caps()` built-in function has the same effect.

`\`*space* (That is, a backslash followed by a space.) A quoted space. This is useful in certain cases to achieve special formatting effects, because it overrides the output formatter's suppression of multiple spaces, and it also suppresses the double-space that follows punctuation marks such as periods. For example, if you want to print someone's name with a middle initial, you don't want two spaces after the initial; you would thus enter: `"Michael J.\ Roberts"`.

`\(` Begin highlighting. Text after the `\(` is displayed with a special attribute, which varies by system; on some machines it will appear

in a different color than ordinary text, while on others it may appear with a different typeface. On some systems it may have no effect. This is intended to be used for emphasizing text the same way that boldface or italics would emphasize text in a book. Note that highlighting does not "nest"; that is, if a \(sequence occurs while highlighting is already active, the second sequence has no additional effect.

\) — End highlighting. Text after the \) without the special highlighting attribute. Note that a single \) sequence turns off highlighting, no matter how many \(sequences preceded it.

There is no specific limit on the length of a double-quote string, although the compiler may impose some limits depending on the context in which the string appears.

A double-quoted string is always displayed whenever evaluated. When a double-quoted string appears in code, executing that code results in displaying the string. Likewise, when a property has a double-quoted string as its value, the string is displayed whenever the property is evaluated.

Embedding Expressions in Strings

TADS has a convenient feature which allows you to embed expressions in strings, without actually stopping the string. Any time TADS encounters two consecutive left angle brackets in a string, it will momentarily stop the string, and evaluate what comes after the angle brackets as an expression whose value is to be displayed. The expression ends, and the string resumes, when two right angle brackets are encountered. This makes property definitions that need to look up another property very convenient to define; for example,

```
itsHere = "<< self.thedesc >> is here."
```

The statement above is equivalent to this more verbose version:

```
itsHere =
{
  self.thedesc; " is here.";
}
```

An embedded expression can evaluate to a number, a single-quoted string, or a double-quoted string. Other datatypes are not allowed. A string may have any number of embedded expressions within it. An embedded expression may only contain an expression; statements (such as `if`) are not allowed, and semicolons may *not* be used to separate multiple expressions—use commas instead.

This feature has certain limitations. It is illegal to use a string in an embedded expression which itself has an embedded expression. This feature can be used only in double-quoted strings; single-quoted strings cannot contain embedded expressions.

Because some game authors may wish to use two or more consecutive left angle brackets in their strings, the special sequence '\<' is provided. See the section on special character sequences, above, for more information.

Single-quoted Strings

Single-quoted strings are essentially the same as double-quoted strings in appearance, except, of course, that they are enclosed in single quote marks. The sequence \' is available to produce a single quote in the string itself.

Single-quoted strings do not display when evaluated; instead, they are treated as values. Certain built-in functions are available to manipulate these strings, such as `say`, which displays such a string value.

Note also that all vocabulary words appear as single-quoted strings (or lists of single-quoted strings).

Lists

A list is an aggregate of other data items. It is entered as a set of data items enclosed in square brackets (note that these square brackets are the actual punctutation, not indicators of optionality):

`[` *data-list* `]`

A *data-list* can be nothing at all, in which case the list is empty, as in

```
[]
```

Or, it can be one or more data items (numbers, strings, objects, lists). Generally, it is only useful to construct lists with the same datatype for all members, but this is not required by the language.

Examples of lists:

```
[ 1 2 3 ]
[ 'hello' 'goodbye' ]
[ [1 2 3] [4 5 6] [7 8 9] ]
[ vase goldSkull pedestal ]
```

List elements need not be constants, unless the entire list needs to be a constant. For example, a list used as a property value must be a constant, so all of the elements must be constants (hence, local variables and non-constant expressions cannot be used). However, a list used in code, such as a list assigned to a local variable, can contain non-constant expressions within the list elements. For example, the following function dynamically constructs a list from three expressions:

```
f:  function(x, y, z)
{
  return([x+1 y+1 z+1]);
}
```

The non-constant list elements are legal because the list construction appears in code. Note that the routine above has the same effect as this code:

```
f:  function(x, y, z)
{
  return((([] + (x+1)) + (y+1)) + (z+1));
}
```

The second example starts with an empty list, then adds an element whose value is `(x+1)`, then adds a second element whose value is `(y+1)`, and so on. The two examples construct exactly the same list. However, you should use the first construct where possible, because it is much more efficient: the first example constructs only a single list, whereas the second must construct *four* lists, one at a time. The first example is faster and consumes less memory at execution time.

---

nil and true

Two special datatypes, `nil` and `true`, are pre-defined. These are generally used as "truth values"; `nil` means false and `true` means true. For example, `1 > 3` evaluates to `nil`, whereas `1 < 3` is `true`. In addition, `nil` means the absence of a value; taking the `car` of an empty list returns `nil`, as does evaluating a property of an object when the property is neither defined nor inherited by the object.

For truth values, you should use `nil` and `true` rather than numeric equivalents, since `nil` and `true` provide much more explicit self-documentation of your program.

Expressions

A property definition can contain an expression rather than a simple constant. When a property value is an expression, the expression must be enclosed in parentheses. The expression is evaluated each time the property is evaluated. An example of using an expression for a property value:

```
exprObj:  object
  x = 1
  y = 2
  z = (self.x + self.y)
;
```

Whenever `exprObj.z` is evaluated, the sum of the current values of `exprObj.y` and `exprObj.z` is returned.

Note that a property defined in this manner can take parameters, just like methods (described below). For example:

```
exprObj2:  object
  w(a, b) = (a * b)
;
```

When `exprObj2.w` is evaluated, it requires two parameters, which are multiplied together to give the value of the property. For example, evaluating `exprObj2.w(3, 4)` produces a value of `12`.

Note that the property `z` above could have been written without the "`self.`" prefix on `x` and `y`:

```
z = (x + y)
```

This is because properties, if used without an object name, are assumed to belong to the current `self` object.

Methods

Code can be used in a property definition just like any other datatype; when you associate code with an object, the code is called a *method*. When the method is evaluated, the code is executed; the return value is the value of the method, but often it is useful for the code to produce side effects as well.

Method code is enclosed in braces, { and }. Anything valid in a function may appear in code associated with a method. The method may have local variables, and it can receive arguments. If the method has arguments, the argument list (which is identical to a function's argument list) appears after the method name. For example:

```
obj1:  object
  f(x) =
  {
    return(x + 1);
  }
;
```

When such a method is evaluated, the argument list is specified after the method name, as in:

```
f1:  function
{
  say(obj1.f(123));
}
```

Functions

A function definition has this form:

identifier: `function` *[* `(` *argument-list* `)` *]*
{
  function-body
}

The *argument-list* is optional; functions need not take arguments. If the function does take one or more arguments, the list looks like this:

identifier [, argument-list]

The identifiers may be used just like local variables within the function.

The punctuation in the above syntactic description may be a little much, so an example might be helpful:

```
addlist:  function(list)                        // add up the numbers in the list
{
  local sum, count, i;
  i := 1;                                     // index the first item in the list
  sum := 0;                                        // initialize the sum to zero
  count := length(list);                   // get the number of items in the list
  while (i < count)                           // as long as there's more to do...
  {
    sum := sum + list[i];                                    // add next element
    i := i + 1;                             // go on to the next list element
  }
  return(sum);
}
```

Functions with Variable Argument Lists

It is possible to define a function that takes a variable number of arguments. In the function definition, you can specify a minimum number of arguments that are always passed to the function (which can be no arguments at all, if you wish), and then specify that more arguments can optionally follow. This is done with the "ellipsis" token, "...", as the final "argument" in the function's argument list. For example, to define a function that takes any number of arguments:

```
f:  function(...)
{
}
```

To define a function that always takes at least one argument, but could take additional arguments:

```
g:  function(fmt, ...)
{
}
```

In a function taking a variable number of arguments, you can determine how many arguments were actually passed to the function by inspecting the pseudo-variable `argcount`. This pseudo-variable's value is simply the number of arguments to the current function.

To retrieve an argument, use the `getarg(`*argnum*`)` built-in function. The argument *argnum* is the number of the argument you want to retrieve; `getarg(1)` returns the first argument, `getarg(2)`

returns the second, and so forth. Note that in a function which has some explicit arguments, followed by an ellipsis, `getarg(1)` still returns the first argument, even though it has a name in the argument list. For example, in the function `g` above, `getarg(1)` returns the value of the argument `fmt`.

Another built-in function, `datatype(`*value*`)`, can be used to determine the datatype of an argument. See the description of the `datatype` function later in this chapter for more information.

As an example, the function below displays any number of values.

```
displist:  function(...)
{
  local i;
  for (i := 1 ; i <= argcount ; i++)
  {
    say(getarg(i));
    " ";
  }
  "\n";
}
```

Forward-Declaration of Functions

An alternative form of the `function` statement allows you to forward-declare a name as a function, without actually defining the function. The format of a forward declaration is:

identifier: `function;`

Note that this does *not* define the function; it merely tells the compiler that a function definition for the specified *identifier* will appear later in the file. This reserves the *identifier* for use as a function name, and prevents the compiler from assuming the *identifier* refers to an object.

Forward declarations are not necessary except where `setdaemon()` and the like will be used. When a function is actually called, the syntax of the call tells TADS that the name refers to a function even when TADS hasn't seen the function definition yet. In `setdaemon()` and similar calls, though, no special syntax is present to tell TADS what the identifier refers to, so the compiler assumes it will refer to an object.

Writing Code

In this section we cover the details of the language that goes inside functions and methods. Function and method code consists of a series of statements; the statements are executed sequentially in the

order they appear in the source program. The following pages provide details about the statements that may be used.

Each statement in TADS code is terminated with a semicolon.

---

local

At the start of a function or property definition, you can define local variables for the current code block. This is done with a statement such as this:

`local` *identifier-list* ;

The *identifier-list* has the form:

identifier [initializer] [, identifier-list]

An *initializer*, which is optional, has the form:

`:=` *expression*

where the *expression* is any valid expression, which can contain arguments to the function or method, as well as any local variables defined prior to the local variable being initialized with the expression. The expression is evaluated, and the resulting value is assigned to the local variable prior to evaluating the next initializer, if any, and prior to executing the first statement after the `local` declaration. Local variables with initializers and local variables without initializers can be freely intermixed in a single statement; any local variables without initailizers are automatically set to `nil` by the run-time system.

The identifiers defined in this fashion are visible only inside the function in which the `local` statement appears. Furthermore, the local statement supersedes any global meaning of the identifiers within the function.

A `local` statement can occur in any block (that is, any statements grouped together with braces), and a single block can have multiple `local` statements; all `local` statements in a block must precede the first executable statement of the block (that is, the only thing that can occur before a `local` statement is another `local` statement.

An example of declaring local variables, using multiple `local` statements, and using initializers is below.

```
f:  function(a, b)
{
  local i, j;                                          /* no initializers */
  local k := 1, m, n := 2;            /* some with initializers, some without */
  local q := 5*k, r := m + q;           /* OK to use q after it's initialized */
  for (i := 1 ; i < q ; i++)
  {
    local x, y;                        /* locals can be at start of any block */
    say(i);
```

```
    }
}
```

Expressions

TADS expressions are entered in algebraic notation. Operators have different meanings on different datatypes. The basic list follows.

`&` — Takes "address" of a function or property. The `&` operator *must* be immediately followed by the name of a property or the name of a function. You can assign this value to a local variable or a property, or you can pass it to a function or method. The address value can then be used to call the function or property. See "Indirect Function and Method Calls" later in this chapter. (Note: older versions of TADS used the `#` operator to take the address of a function. This operator is still understood by the system, but we recommend that you use `&` in new code instead.)

`.` — Takes a property of an object. On the left of the dot is an expression that evaluates to an object, and on the right is a property name. If the property is a method that requires arguments, these arguments are listed in parentheses after the property name, just as with function arguments. A local variable or an expression enclosed in parentheses may appear on the right hand side; the local or expression must evaluate to a property pointer.

`[]` — List indexing; applied to the right side of a list-valued expression. An expression which evaluates to a number must appear between the square brackets. If the index (i.e., the value between the square brackets) is n, the nth element of the list is returned by this operator. The first element in the list is number 1. For example, `['one' 'two' 'three'][1+1]` evaluates to the string `'two'`. Note that a run-time error results if n is outside the bounds of the list; that is, n is less than 1 or greater than the number of elements in the list (as returned by the `length()` built-in function).

`++` — Increment; adds one to a numeric variable or property. The expression to which `++` is applied must be suitable for assignment (i.e., it must be something that you can use on the left-hand side of the assignment operator, "`:=`"). The `++` operator can be either a prefix or postfix operator—that is, it can be placed either before

or after its operand. If it's placed before the operand, as in `++i`, the operand is incremented, and the value of the expression is the value of the operand *after* the increment. If it's placed after the operand, as in `i++`, the value of the expression is the value of the operand *before* the increment. So, if the variable `i` has the value `3`, `++i` has the value `4`, while `i++` has the value `3`. Think about it by reading the expression left-to-right, and noting that the value of the expression is the value of `i` *when you read it*, and the increment is applied when you read the `++` operator. So, with `++i`, you first read the increment operator, which adds 1 to `i`, then you note the value of `i`, which by now is `4`. With `i++`, you first read `i` and note its value, which is still `3`, then you see the `++` operator, leaving `4` in `i`.

`--` Decrement; subtracts one from a numeric variable or property. This acts exactly like the `++` operator, except that it decrements its operand rather than incrementing it.

`not` Logical negation. Turns `true` into `nil` and vice versa. This operator can only be applied to the values `true` and `nil`.

`-` Arithmetic negation. As a unary prefix operator (that is, an operator that precedes its single operand), the minus sign negates the number following it.

`*` Numeric multiplication. Multiplies the numbers on either side of the asterisk together.

`/` Numeric division. The number on the left of the slash is divided by the number on the right. Note that TADS numbers are all integers; the remainder of the division is discarded. Hence, `7/2` has a value of `3`.

`+` Adds numbers, concatenates lists, concatenates strings. If both the left and right are numbers, the result is numeric addition. If one or the other is a list, but not both, the non-list is added as the last element of the list. If both operands are lists, the items from the list on the right side are appended to the list on the left; for example,

`[1 2 3] + [4 5]` yields the list `[1 2 3 4 5]`. If both are strings, the right string is concatenated to the left string. Other datatypes are illegal.

`-` Subtracts numbers, removes items from lists. If both operands are numbers, the right operand is subtracted from the left. If the left element is a list, the right item is removed from the list if it appears in the list on the left (nothing happens if not); if the right item is also a list, each element from the right list that appears in the left list is removed from the left list. Other datatypes are illegal.

`=` Equality. The datatypes on either side must be the same, but can't be lists. One exception: anything can be compared for equality to `nil`. Evaluates to `true` if the items are the same, `nil` otherwise.

`<>` Inequality. True if the operands on both sides are not the same (although the datatype of each operand must be the same).

`>` Greater than. Like the other comparison operators, this may only be applied to numbers and strings. Evaluates to `true` or `nil`. Note that, when applied to strings, the comparison is based on the collation sequence of the character set of the computer you are using, such as ASCII.

`<` Less than.

`>=` Greater than or equal to.

`<=` Less than or equal to.

`and` Logical product: both the left and right sides must be `true`, in which case the value is `true`; otherwise, the value is `nil`. Note that if the left operand is `nil`, the right operand is not evaluated at all, since the value of the expression will be `nil` regardless of the value of the right operand.

`or` Logical sum: one or the other of the left or right operators must be `true` for the value to be `true`; otherwise the value is `nil`. Note

that if the left operand is `true`, the right operand is not evaluated at all, since the value of the expression will be `true` regardless of the value of the right operand.

`? :` Conditional. This is a *tertiary* operator; that is, it takes three operands, of the form *cond* `?` *true-expr* `:` *false-expr*. First, the *cond* is evaluated; if it evaluates to `true` or a non-zero number, then the *true-expr* is evaluated, and its value is the value of the entire conditional expression. If the *cond* is false, the *false-expr* is evaluated, and its value is the value of the entire expression. Note that either *true-expr* or *false-expr* is evaluated—not both. This allows you to use this operator when the expressions have side effects, such as displaying strings.

`:=` Assignment. The variable or property on the left of the operator is assigned the value on the right side. This is the lowest priority operator, and operates right to left, unlike the other operators. Hence, `a := b := 3` assigns the value `3` to `b`, then to `a`. Note that an assignment is an expression, which returns the assigned value; hence, the value of `(a := 3) + 4` is `7`, with the side effect that the variable `a` has been set to `3`.

`+=` Add and assign. This is simply short-hand notation. The expression `a += b` has exactly the same effect as the expression `a := a + b`, except that the `+=` operator makes the expression easier to write. The value of the expression is `a + b`, which is the same as the value of `a` after the assignment.

`-=` Subtract and assign. This is short-hand notation. The expression `a -= b` has the same effect as `a := a - b`.

`*=` Multiply and assign. The expression `a *= b` has the same effect as `a := a * b`.

`/=` Divide and assign. The expression `a /= b` has the same effect as `a := a / b`.

`,` Conjunction. The comma operator simply allows you to start a new expression within an expression. On the left hand side is an

expression, and on the right hand side is another expression. The two expressions are independent; the comma operator performs no computation on either expression. The value of a pair of expressions separated by a comma is the value of the second (right-hand) expression. The comma operator is useful mostly in contexts where an expression is required, but you wish to evaluate several otherwise independent expressions for their side effects. For example, in the initialization portion of a `for` statement, you often wish to initialize several variables; you can do this by separating the initializations with commas. See the description `for` statement later in this chapter. *Note:* the comma operator is different from a comma that appears in a function's or method's argument list. In the argument list, the comma simply separates different expressions that are the arguments; in a normal expression, however, the comma separates parts of the same expression.

The operators are shown above in their order of evaluation, from first to last. Hence, conjunctions are done last, after all higher-precedence operators have been evaluated. Note that there are some groups of operators in the list above; for example, all of the comparison operators are grouped together. This indicates that these operators have the same precedence. Operators of the same precedence associate left to right; for example, `3-4+5` is evaluated as `(3-4)+5`. The exception is the assignment operator, which groups right to left; that is, in the expression `a := b := 2`, `b` is first set to `2`, then `a` is set to `b`.

You can use parentheses to force a different order of evaluation of your expression.

Assignments

An assignment is not a special type of statement; it is merely an expression which uses an assignment operator:

item `:=` *expression*;

The *item* is a local variable, a list element, or an object's property. When assigning to a property of an object, the object reference may itself be an expression. Likewise, when assigning to a list element, the list and index may both be expressions. Note, however, that lists cannot be expanded by assigning to an element past the end of the list; the index must refer to an existing member of the list to be replaced. A few examples of assignments:

```
a:  function(b, c)
{
  local d, e, lst;
  d := b + c;                                    // assign a local variable
  obj3.prop1 := 20;                          // assign obj3's property prop1
  e := obj3;                           // assign an object to a local variable
```

```
  e.prop1 := obj2;                // assign obj3's property prop1 the object obj2
  e.prop1.prop2 := 20;              // assign obj2's property prop2 the number 20
  fun1(3).prop3 := 1 + d;          // assign prop3 of the object returned by fun1
  lst := [1 2 3 4 5];                      // set up a local variable with a list
  lst[3] := 9;                                          // lst is now [1 2 9 4 5]
  lst[5] := 10;                                       // and it's [1 2 9 4 10] now
  /* lst[6] := 7 would be illegal - the list is only 5 elements long */
}
```

Function Calls

A function call is simply an expression involving a call to a function:

function-name(*[argument-list]*);

The *function-name* is the name of a function defined elsewhere. The argument list, if provided, is passed to the function for its parameters. (Naturally, the arguments passed to a function should match in number those defined in the function's definition.) Each parameter can be an arbitrary expression, and the individual arguments are separated by commas.

Whether there's an argument list or not, the parentheses are required; they tell the compiler that you wish to call the function. (You may wonder where you would ever want to use a function's name other than when calling the function. A few special built-in functions, such as `setdaemon`, which will be discussed in detail later, allow you to specify a function that is to be called eventually, but not right away. In these cases, you need to be able to refer to a function without actually calling it. In these cases, you use the function's name without parentheses.)

If the function returns a value, the value is discarded in this form of the call. A function called in this manner is invoked for its side effects rather than its return value.

An example:

```
showlist(a+100, 'The list is:  ', list1);
```

Indirect Function and Method Calls

You can call a function or method "indirectly"; that is, you can use a pointer to a function or method to call the function or method. This can sometimes be useful in setting up a general routine which calls other routines based on parameters passed into it.

To call a function with a function pointer, simply use an expression yielding a function pointer, within parentheses, where you'd normally use a function name. For example:

```
g:  function(a, b, c)
{
  return(a + b + c);
```

```
}

f:  function
{
  local fptr, x;
  fptr := &g;                                    /* get address of function g */
  x := (fptr)(1, 2, 3);                         /* call function with pointer */
}
```

A property pointer is used in essentially the same way.

```
f:  function(actor, obj)
{
  local propPtr := &doTake;                 /* get pointer to doTake property */
  obj.(propPtr)(actor);                      /* call property through pointer */
}
```

For example, suppose you wish to implement a way of asking characters in a game about various objects in the game. One way you could do this is by defining a general "ask" routine that takes the character and the object as arguments. Set things up so that each object defines a property saying what each actor knows about that object. Then, each actor specifies via a "property pointer" which property to evaluate in an object to find out what the actor knows about the object. If an object doesn't define this property, the actor doesn't know anything about that object.

So, our general "ask" routine is very simple (or concise, anyway):

```
ask:  function(actor, obj)
{
  local propPtr;

  /* find out what property to evaluate in the object */
  propPtr := actor.askPropPtr;

  /* see if it's defined in the object */
  if (defined(obj, propPtr))
  {
    /* it is defined - call the property indirectly */
    obj.(propPtr);
  }
  else
  {
    /* it's not defined - use default message */
    actor.dontKnow;
  }
}
```

Now, each actor simply has to define a property that each object uses to specify what the actor

knows about the object, and place the address of this property the actor's `askPropPtr` property. The actor also needs to define the default message in the `dontKnow` property. Here's an example:

```
joe:  Actor
  sdesc = "joe"
  noun = 'joe'
  dontKnow = "Joe just scratches his head and shrugs."
  askJoe = "You probably don't want to get Joe started on his life story."
  askPropPtr = askJoe
;
```

Finally, each object must define an appropriate `askJoe` property if Joe knows anything about that object. Likewise, it will define other properties for what other actors know about it. This way, all of the information about an object, including what the various characters in the game know about it, can be kept with the object itself. In addition, the general "ask" routine is extremely simple. The overall concept behind the mechanism is somewhat complicated, but the finished product is very simple and easy to use and expand.

---

return

A function can return a value to its caller by using a statement such as:

`return` *[expression]*;

With or without the *expression*, execution of the function is terminated and the caller resumes execution where it left off. If the *expression* is provided, the caller receives the expression as the function's value.

Here's an example of a function that computes the sum of the elements of a list, and returns the sum as the value of the function.

```
listsum:  function(lst)
{
  local i, sum, len := length(lst);
  for (i := 1, sum := 0 ; i <= len ; i++)
    sum += lst[i];
  return(sum);
}
```

---

if

The general form of the conditional in TADS is:

`if (` *expression* `)` *statement*
[`else` *statement]*

Here and elsewhere, a *statement* can be either a single statement or a series of statements enclosed in braces. The *expression* should evaulate to either a number, in which case zero counts as false and anything else counts as true, or to a truth value, `true` or `nil`.

Note that the optional `else` clause is grouped with the most recent `if` statement when `if` statements are nested. For example,

```
if (self.islit)
  if (film.location = Me)
    "Oops!  You've exposed the film!";
else
  "It's dark in here.";
```

The author of this code obviously intended the `else` clause to go with the first `if`, but remember that an `else` goes with the *most recent* `if`, so it actually is grouped with the second `if` statement. This problem can be easily rectified by using braces to make the grouping explicit:

```
if (self.islit)
{
  if (film.location = Me)
  {
    "Oops!  You've exposed the film!";
  }
}
else
{
  "It's dark in here.";
}
```

---

switch

The `switch` statement lets you set up the equivalent of a large `if-else` tree, but is considerably easier to read and is more efficient to execute. A `switch` statement allows you to test a particular value against several alternatives, and execute a group of statements accordingly.

The form of the `switch` statement is:

```
switch ( expression )
{
  [ case-list ]
  [ default-clause ]
}
```

The form of the *case-list* is:

```
case constant-expression :
  [ statements ]
```

[case-list]

The form of the *default-clause* is:

`default:`
  [statements]

In the diagrams, *statements* means that zero or more statements can follow a `case` or `default`. You do not need to supply any `case` labels at all, and the `default` is also optional.

The *expression* evaluates to a number, string, object, list, or `true` or `nil`. The value of the expression is then tested against each `case` value. If the value matches one of the `case` values, the statements following the matching `case` are executed. If the value does not match any `case` value, and a `default` case is defined, the statements following the `default` are executed. If the value does not match any `case` value, and there is no `default` case, the entire `switch` statement is skipped, and execution resumes following the closing brace of the `switch`.

Note that execution is *not* interrupted when another `case` is enountered. Instead, it just continues into the statements following the `case` label. If you wish to stop executing statements in the `switch` at the end of the statements for a single `case`, you must use the `break` statement.

The `break` statement has a special meaning within a `switch` statement: it indicates that execution should break out of the `switch` statement and resume following the closing brace of the `switch`.

Here's an example of a `switch` statement.

```
f:  function(x)
{
  switch(x)
  {
  case 1:
    "x is one";
    break;
  case 2:
  case 3:
    "x is either 2 or 3";
    break;
  case 4:
    "x is 4";
  case 5:
    "x is either 4 or 5";
  case 6:
    "x is 4, 5, or 6";
    break;
  case 7:
    "x is 7";
    break;
  default:
```

```
      "x is not in 1 through 7";
    }
  }
```

---

while

The `while` statement defines a loop: a set of statements that is executed repeatedly as long as a certain condition is true.

`while (` *expression* `)` *statement*

As with the `if` statement, the *statement* may be a single statement or a set of statements enclosed in braces. The *expression* should be a number (in which case 0 is false and anything else is true), or a truth value (`true` or `nil`).

The *expression* is evaluated *before* the first time through the loop; if the *expression* is false at that time, the statement or statements in the loop are skipped. Otherwise, the statement or statements are executed once, and the *expression* is evaluated again; if the *expression* is still true, the loop executes one more time and the cycle is repeated. Once the *expression* is false, execution resumes at the next statement after the loop.

---

do-while

The `do-while` statement defines a slightly different type of loop than the `while` statement. This type of loop also executes until a controlling expression becomes false (0 or `nil`), but evaluates the controlling expression *after* each iteration of the loop. This ensures that the loop is executed at least once, since the expression isn't tested for the first time until after the first iteration of the loop.

The general form of this statement is:

`do` *statement* `while (` *expression* `);`

The *statement* may be a single statement or a set of statements enclosed in braces. The expression should be a number (in which case 0 is false and anything else is true), or a truth value (`true` or `nil`).

---

for

The `for` statement defines a very powerful and general type of loop. You can always use `while` to construct any loop that you can construct with `for`, but the `for` statement is often a much more compact and readable notation for the same effect.

The general form of this statement is:

`for (` *init-expr* `;` *cond-expr* `;` *reinit-expr* `)` *statement*

As with other looping constructs, the *statement* can be either a single statement, or a block of statements enclosed in braces.

The first expression, *init-expr*, is the "initialization expression." This expression is evaluated once, before the first iteration of the loop. It is used to initialize the variables involved in the loop.

The second expression, *cond-expr*, is the condition of the loop. It serves the same purpose as the controlling expression of a `while` statement. Before each iteration of the loop, the *cond-expr* is evaluated. If the value is true (a non-zero number, or `true`), the body of the loop is executed; otherwise, the loop is terminated, and execution resumes at the statement following the loop body. Note that, like the `while` statement's controlling expression, the *cond-expr* of a `for` statement is evaluated prior to the first time through the loop (but after the *init-expr* has been evaluated), so a `for` loop will execute zero times if the *cond-expr* is false prior to the first iteration.

The third expression, *reinit-expr*, is the "re-initialization expression." This expression is evaluated *after* each iteration of the loop. Its value is ignored; the only purpose of this expression is to change the loop variables as necessary for the next iteration of the loop. Usually, the re-initialization expression will increment a counter or perform some similar function.

Any or all of the three expressions may be omitted. Omitting the expression condition is equivalent to using `true` as the expression condition; hence, a loop that starts "`for ( ;; )`" will iterate forever (or until a `break` statement is executed within the loop). A `for` statement that omits the initialization and re-initialization expressions is the same as a `while` loop.

Here's an example of using a `for` statement. This function implements a simple loop that computes the sum of the elements of a list.

```
sumlist:  function(lst)
{
  local len := length(lst), sum, i;
  for (sum := 0, i := 1 ; i <= len ; i++)
    sum += lst[i];
}
```

Note that an equivalent loop could be written with an empty loop body, by performing the summation in the re-initialization expression. We could also move the initialization of `len` within the initialization expression of the loop.

```
sumlist:  function(lst)
{
  local len, sum, i;
  for (len := length(lst), sum := 0, i := 1 ; i <= len ;
       sum += lst[i], i++);
}
```

---

break

A program can get out of a loop early using the `break` statement:

```
break;
```

This is useful for terminating a loop at a midpoint. Execution resumes at the statement immediately following the innermost loop in which the break appears.

The `break` statement also is used to exit a `switch` statement. In a `switch` statement, a `break` causes execution to resume at the statement following the closing brace of the `switch` statement.

---

continue

The `continue` statement does roughly the opposite of the `break` statement; it resumes execution back at the start of the innermost loop in which it appears. The `continue` statement may be used in `for`, `while`, and `do-while` loops.

In a `for` loop, `continue` causes execution to resume at the re-initiailzation step. That is, the third expression (if present) in the `for` statement is evaluated, then the second expression (if present) is evaluated; if the second expression's value is non-nil or the second expression isn't present, execution resumes at the first statement within the statement block following the `for`, otherwise at the next statement following the block.

---

goto

The `goto` statement is used to transfer control unconditionally to another point within the same function or method. The target of a `goto` is a label; a label is defined by placing its name, followed by a colon (`:`), preceding a statement.

Note that labels have function- or method-scope; that is, they are visible within the entire function or method in which they are defined. This is different from local variables, which are visible only within the *block* (the group of statements enclosed in braces) in which they are defined. Labels are not visible outside the function or method in which they are defined.

An example of using `goto`:

```
f:  function
{
  while (true)
  {
    for (x := 1 ; x < 5 ; x++)
    {
      /* do some stuff */
      if (myfunc(3) < 0)                          /* did an error result?  */
        goto exitfunc;                                 /* error - quit now */
      /* do some more stuff */
    }
  }
```

```
  /* come here if something goes wrong */
exitfunc:  ;
}
```

This use of `goto` avoids the need for testing a flag in the outer (`while`) loop, which makes the code a little simpler and easier to understand.

The `goto` statement is widely considered among civilized computer scientists to be an evil and malevolent feature of ancient and unusable languages, and the esteem of TADS within serious computer language design circles has undoubtedly been fatally injured by the inclusion of this construct. So, you may wish to use this statement sparingly, if at all, especially if you're hoping to impress a civilized computer scientist with your coding efforts. However, many software engineers look upon `goto` as a highly useful statement when in the hands of a seasoned professional, and scoff at the blanket indictment by the more-elegant-than-thou academic establishment, most of whom probably haven't written a line of code since *their* TA's were chastising them for using `goto` in their Pascal programs, excepting perhaps some algorithms written in pseudo-code that always end in "the rest is left as an exercise for the reader" anyway. The author of TADS doesn't wish to take sides in this heated controversy, but hopes that both camps will be pleased, by gaining either the utility of using `goto` with wild abandon or the sense of virtue of knowing they could have used it but overcame the unclean temptation. With TADS, the choice is yours.

---

pass

A method can decide to inherit the behavior of its parent class by using the `pass` statement:

`pass` *method-name*;

The *method-name* must be the same as the name of the method that the `pass` statement occurs in. When a `pass` statement is executed, the method that would have been inherited if the object had not overridden it is called with the same arguments, if any, as the method called in the first place. The `self` object is unchanged; that is, the superclass method is run, but `self` is the object that was originally sent the message being passed.

---

exit and abort

During processing of a player's command, the game can terminate the normal sequence of events and return to the get another command from the player in two different ways. The `abort` statement stops all processing, and gets the next command; it is normally used by "system" functions, such as saving the game, that should not count as a turn, and therefore shouldn't run any daemons or fuses. The `exit` statement, on the other hand, skips everything up to the fuses and daemons; `exit` is used when an actor wishes to stop further processing, and other similar cases.

Note that messages to objects scheduled with the `notify()` function are treated the same as other daemons and fuses, so these are also skipped by the `abort` statement.

askdo and askio

These statements interrupt processing of a turn like `abort`, skipping fuses and daemons, but allow the user to enter more information. The `askdo` command asks the user to enter a direct object; the system displays a message asking the user for an object. The user can either enter an object in response to the request, or can simply type a new command. For example, if the verb is "take," executing an `askdo` command will cause the system to display "What do you want to take?" and wait for an object or a new command. The `askio` command is similar, but it takes a preposition as a parameter; this preposition is added to the command, and the system prompts the player for an indirect object. For example, if the verb is "unlock," and the following command is executed:

```
askio(withPrep);
```

then the system displays "What do you want to unlock it with?" and prompts the player for an object or a new command. The `askio` command can only be used when a direct object is already present.

In either case, if the player responds to the request with an object, the command is tried again from the start. If the player types a new command, the command that resulted in the `askdo` or `askio` is discarded. Note that control never comes back to the statement after the `askdo` or `askio` command, regardless of user input; in either case, command processing starts from the top.

Note that, in some cases, the system won't actually ask the player for a new object. Instead, the system will attempt to find a default object, using exactly the same mechanisms that it uses to find default objects normally (see the section on the parser's default object mechanisms in chapter four).

self

When code associated with a property is being executed, a special object is defined, called `self`. This special object refers to the object whose property is being evaulated. This may not sound too useful, but consider the case of an object whose superclass defines a property which refers to other properties of the object:

```
class book:  object
  description =
  {
    "The book is << self.color >>.";
  }
;

redbook:  book
  color = "red"
;

bluebook:  book
```

```
  color = "blue"
;
```

In this example, the general object, `book`, knows how to describe a book given its color. The books that are defined, the `redbook` and `bluebook` objects, take advantage of this by simply defining their color, and letting the description property of their superclass be used to describe them. So, when you attempt to evaluate `redbook.description`, you get

```
The book is red.
```

---

inherited

A special pseudo-object called `inherited` allows you to call a method in the current `self` object's superclass. This pseudo-object is similar to the `pass` statement, but much more useful in several ways. First, with `inherited`, you can simply *call* the superclass method, and regain control when it returns; with `pass`, the current method never regains control. Second, you can use `inherited` in an expression, so any value returned by the superclass method can be determined and used by the current method. Third, you can pass arguments to the property invoked with the `inherited` pseudo-object.

You can use `inherited` in an expression anywhere that you can use `self`.

Here is an example of using `inherited`.

```
myclass:  object
  sdesc = "myclass"
  prop1(a, b) =
  {
    "This is myclass's prop1.  self = << self.sdesc >>,
    a = << a >>, and b = << b >>.\n";
    return(123);
  }
;

myobj:  myclass
  sdesc = "myobj"
  prop1(d, e, f) =
  {
    local x;
    "This is myobj's prop1.  self = << self.sdesc >>,
    d = << d >>, e = << e >>, and f = << f >>.\n";
    x := inherited.prop1(d, f) * 2;
    "Back in myobj's prop1.  x = << x >>\n";
  }
;
```

When you call `myobj.prop1(1, 2, 3)`, the following will be displayed:

```
This is myobj's prop1.  self = myobj, d = 1, e = 2, and f = 3.
This is myclass's prop1.  self = myobj, a = 1, and b = 3.
Back in myobj's prop1.  x = 246.
```

Note one feature of `inherited` that is the same as `pass`: the `self` object that is in effect while the superclass method is being executed is the *same* as the `self` object in the calling (subclass) method. This makes `inherited` very different from calling the superclass method directly (i.e., by using the superclass object's name in place of `inherited`).

---

argcount

The pseudo-variable `argcount` returns the number of arguments to the current function. This can be used for functions that take a variable number of arguments to learn the number of arguments that need to be processed. Note that `argcount` isn't really a variable, so you can't assign a value to it, but otherwise you can use it as though it were an ordinary variable.

Built-in Functions

The system has a set of built-in functions to facilitate writing programs. The functions are described in this section. They operate just like ordinary functions, with the exception that built-in functions which don't take arguments don't require parentheses; this simplifies coding of some special cases.

---

askfile

Call: `askfile(`*prompt*`)`

This function asks the user to enter a filename, in a system-dependent manner. The system's standard file dialog, if the computer has one, will be used; otherwise, the user may simply be prompted to type a filename. The *prompt* (a single-quoted string value) may or may not be used, depending on system conventions; for systems without any defined standard file dialog, it will be displayed to prompt the user for a filename.

This function is primarily useful for operations such as saving and restoring games which require that the user enters a system filename.

---

caps

Call: `caps()`

Forces the next non-space character to be displayed to be capitalized. This is useful for formatting output when it is not known in advance whether an item will be displayed at the start of a sentence or not.

Note that displaying the sequence "`\^`" has the same effect as calling `caps()`.

Do not confuse this function with the `upper(`*string*`)` built-in function, which converts all of the letters in a string to upper-case. The `caps()` function takes no arguments, and affects only the next character output.

---

car

Call: `car(`*list*`)`

Returns the first element of a list, or `nil` if the list is empty.

Note that the same value can be retrieved with the expression *list*`[1]`, which uses the list indexing operator to retrieve the first element of *list*. The primary difference between using `car()` and the list indexing operator is the style of your program; using `car()` and `cdr()` to decompose a list, you can iterate or recurse until you run out of list to process, at which time `car()` will return `nil`. With the list indexing operator, however, you need to know in advance how many elements are in the list; this information can be learned with the `length()` built-in function. The choice of one of these methods over another is a matter of personal preference.

---

cdr

Call: `cdr(`*list*`)`

Returns the end of a list; that is, the list of everything after the list's `car()`. With `car()` and `cdr()` you can decompose a list into its individual elements.

---

cvtnum

Call: `cvtnum(`*string*`)`

This function converts a string that contains the text version of a number into a numeric value. For example, `cvtnum('1234')` returns the number `1234`. Note that the special strings `'true'` and `'nil'` are also accepted by the function, and are converted to the logical values `true` and `nil`, respectively.

---

cvtstr

Call: `cvtstr(`*value*`)`

This function converts a numeric or logical value into its string representation. For example, `cvtstr(1234)` returns the string `'1234'`, `cvtstr(true)` returns the string `'true'`, and `cvtstr(nil)` returns the string `'nil'`.

---

datatype

Call: `datatype(`*value*`)`

This function returns the type of a value. It is useful primarily with functions that take a variable number of arguments, but could also be useful for inspecting lists whose contents vary.

This function returns a numeric value based on the datatype.

1	Number
2	Object
3	String
5	`nil`
7	List
8	`true`
10	Function Pointer
13	Property Pointer

---

defined

Call: `defined(`*object, propPointer*`)`

This function allows you to determine if a property is defined or inherited by an object, or if the object doesn't have any value for the property. This function returns `true` if the the object has a definition for the property (either explicitly in the object, or inherited from a superclass), or `nil` if the object doesn't have any definition for the property.

Note that the *propPointer* argument must be a property *pointer*. You can obtain a property pointer using the `&` operator with a property name. For example, to determine if the player's current location has an `ldesc` property defined, you would do this:

```
x := defined(Me.location, &ldesc);
```

In this example, `x` is `true` if the current location has an `ldesc` property defined, and `nil` otherwise.

---

find

Call: `find(`*value, target*`)`

If *value* is a list; the function returns the offset (starting at 1 for the first element) in the list of the *target* item within the *value* list. If the *target* is not found, `nil` is returned. For example, `find([4 5 6], 5)` returns 2.

If *value* is a string, in which case *target* must also be a string, the function will return the offset within the string *value* of the substring *target*, or `nil` if the substring is not found. The offset of the first character in the *target* string is 1. For example, `find('abcdefghij', 'cde')` returns 3.

---

firstobj

Call: `firstobj()`

Alternative Call: `firstobj(`*class*`)`

This function is used in conjunction with `nextobj(`*object*`)` to loop over all non-class objects in the game. The `firstobj()` function returns the first non-class object, and `nextobj(`*object*`)` is used to retrieve subsequent objects in a loop. The order of the objects returned by these functions is arbitrary, but calling `firstobj()` and then sequentially calling `nextobj(`*object*`)` until `nil` is returned will guarantee that each non-class object is retrieved exactly once.

These functions are useful primarily during initialization to construct lists of all objects satisfying certain search criteria; these lists can be used later in the game to expedite searches for objects with such criteria. For example, code that determines if a room is dark will have to always check to see if a light-providing object, such as a lamp or a burning candle, is present. It will be faster during game play to check only those objects known to be light-providing than to check all objects in the game; to accomplish this, we could set a property called `islamp` to `true` in every potential light-providing object, then construct a list of all such objects during initialization with the example code below.

```
getlamps:  function
{
  local obj, l;
  l := [];
  obj := firstobj();
  while(obj <> nil)
  {
    if (obj.islamp) l := l + obj;
    obj := nextobj(obj);
  }
  global.lamplist := l;
}
```

After initialization, then, it is only necessary to check for the presence of one of the objects in the list `global.lamplist`, rather than checking all objects in the game, to determine if a room is lit or not.

Note that `firstobj()` will return `nil` if the game has no non-class objects. Likewise, `nextobj(`*object*`)` returns `nil` when the last non-class object has been retrieved.

The alternative form of this function, with a single argument giving a class object, allows you to restrict the objects returned by `firstobj()` and `nextobj()` to those that are subclasses of a particular class. This can save a great deal of time by ignoring objects that are not important to your search. For the example above, you could make the loop execute much more quickly by rewriting it as follows.

```
getlamps:  function
{
  local obj, l;
  l := [];
  obj := firstobj(lampitem);
  while(obj <> nil)
  {
    l := l + obj;
    obj := nextobj(obj, lampitem);
  }
  global.lamplist := l;
}
```

Note that the test for `obj.islamp` is no longer necessary, because only objects that are subclasses of the class `lampitem` will be used in the loop. By iterating over a much smaller set of objects, the loop will execute substantially faster.

---

getarg

Call: `getarg(`*argnumber*`)`

This function returns the argument given by *argnumber*, which is a number from 1 to the number of arguments to the function (which can be learned with the pseudo-variable `argcount`). `getarg(1)` returns the first argument to the current function, `getarg(2)` returns the second argument, and so forth, up to `getarg(argcount)`. This function can be used to retrieve the arguments to functions taking a variable number of arguments.

---

incturn

Call: `incturn()`

Increments the turn counter. Normally, a daemon is present to call this function once per player command. The turn counter is used to time the execution of fuses, so the user must call `incturn()` once per turn.

This function is not called automatically by the system, since the turn counter should not be incremented after certain events. For example, most system commands, such as saving a game, should not count as a turn. This function is provided so that the game program can decide when to increment the counter.

input

Call: `input()`

This function stops and waits for the user to enter a line of text (terminated by the return key and edited as normal for command lines), and returns a string containing the text the user enters. This function does not display any prompt, so it is up to the game program to prompt the user before calling this function.

As a stylistic point, this function should normally be avoided except for special system functions, since the game will present a more consistent interface if the command line is used for most player input. One possible use is given by the example below. (Note that this code is somewhat simplified; in an actual game, you would also want to call `setscore()` at the appropriate points, and provide other useful feedback.)

```
die:  function
{
  "*** You have died ***
  \bDo you wish to RESTART, RESTORE, or QUIT? >";
  while (true)
  {
    local response;
    response := upper(input());
    if (response = 'RESTART') restart();
    else if (response = 'RESTORE')
    {
      response := askfile();
      if (restore(response)) "Failed.  ";
      else abort;
    }
    else if (response = 'QUIT')
    {
      quit();
      abort;
    }
    else "Please enter RESTORE, RESTART, or QUIT: >";
  }
}
```

isclass

Call: `isclass(`*object, class*`)`

This function determines if an object is a subclass of a given class. It returns `true` if *object* is a subclass of *class*, `nil` otherwise. Note that the function scans the entire class tree, so *class* need not

be in the immediate superclass list for *object*, but can be a superclass of a superclass of *object*, or a superclass of a superclass of a superclass, and so on.

This function should be used to determine if an object is a member of a class, rather than using a special property value that the object inherits from the class. Using `isclass` is more efficient than using a special inherited property value, because the property value doesn't need to be stored, and because it is much faster to scan the inheritance tree than to check each object in the tree for the property value.

---

length

Call: `length(`*item*`)`

If the *item* is a string, `length()` returns the number of characters in the string. If *item* is a list, `length()` returns the number of elements in the list.

---

logging

Call: `logging(`*value*`)`

If the *value* is a string, it specifies the name of an operating system file which will be created (or truncated to zero length if it already exists) and to which subsequent information displayed on the screen is duplicated. That is, a complete log of the play will be copied to the file for later inspection.

If the *value* is `nil`, it indicates that a log file, if in effect, should be closed and logging halted.

Logging is automatically terminated when the player quits the game, but other operations (including saving, restoring, and restarting a game) do not affect logging.

This function has no return value.

---

lower

Call: `lower(`*string*`)`

This function returns a copy of *string* with all of the upper-case letters converted to lower-case. See also the `upper(`*string*`)` function.

---

nextobj

Call: `nextobj(`*object*`)`

Alternative Call: `nextobj(`*object, class*`)`

This function is used to retrieve the next object in a search started with `firstobj()`. The *object* is the value returned by the call to `firstobj()` or by the previous `nextobj(`*object*`)` call. The next non-class object after *object* is returned, and `nil` is returned if *object* was the last non-class object.

The alternative form, with a second argument giving a class object, returns the next object that is a subclass of the given class. This can be used to restrict the search to a particular class of objects.

For an example of the use of this function, see the description of the `firstobj()` built-in function.

---

notify

Call: `notify(`*object*`, &`*message*`,` *turns*`)`

This function sets up to send the *message* to the *object* after the given number of *turns* has elapsed, or, if *turns* is zero, after each turn. Note the `&` before the *message* parameter; this is required to indicate that the *message* is to be stored rather than being sent immediately.

The *message* is simply the name of a method defined by the *object*. The method receives no parameters when it is called.

There is a limit of 100 simultaneously pending notifications.

With a non-zero number of *turns*, the `notify()` function is similar to the `setfuse()` function, in that it waits a given number of turns and then sends the specified *message* to the given *object*. With *turns* set to zero, `notify` is similar to `setdaemon()`, in that the *message* is sent to the *object* after each turn. Note that the `notify()` function is generally the better function to use, since most fuses and daemons are directed at objects, and it is better to keep all code that affects the object in the object's definition, rather than move some into a separate function.

See also the `unnotify()` function, which cancels the effect of the `notify()` function.

An example of the `notify()` function is shown below. In the example, we define two objects: a bomb, and a button on the bomb. When the button is pushed, it calls `notify()` to specify that the `explode` method in the object `bomb` is to be called in three turns.

```
bomb:  item
  location = bombroom
  sdesc = "bomb"
  noun = 'bomb'
  ldesc =
  {
    "The bomb seems to have a small button marked
    \"detonator\" on it.  ";
    if (self.isActive) "It's ticking loudly.  ";
  }
  explode =
  {
    "The bomb explodes!  ";
```

```
    self.moveInto(nil);
  }
;

bombButton:  buttonItem
  location = bomb
  sdesc = "detonator button"
  adjective = 'detonator'
  doPush(actor) =
  {
    "The bomb starts ticking.  ";
    notify(bomb, &explode, 3);                    // Bomb explodes in three turns
    bomb.isActive := true;
  }
;
```

---

proptype

Call: `proptype(`*object, propPointer*`)`

This function determines the datatype of a property of an object, without actually evaluating the property. This can be useful if you want to find out what kind of value the property has, without activating any side effects of evaluating the property (for example, displaying a double-quoted string contained by the property).

The return value is a number, and the possible values are a superset of those returned by the `datatype()` built-in function.

1	Number
2	Object
3	String
5	`nil`
6	Code (executable statements enclosed in braces)
7	List
8	`true`
9	Double-quoted string
10	Function Pointer
13	Property Pointer

Note that the *propPointer* argument must be a property pointer. You can get a property pointer by using the `&` operator on a property name. For example, to determine the type of the `north` property of the player's current location, you would do this:

```
x := proptype(Me.location, &north);
```

Note that using `proptype()` is very different from using the `datatype()` built-in function on the value of the property, because `proptype()` does *not* evaluate the property in order to determine its type. This has several implications. First, and most importantly, it allows you to avoid side effects of evaluating the property (such as displaying a double-quoted string value) if you don't desire the side effects. Second, it means that you will find out if a property contains code, but you will *not* learn what type of value (if any) the code will return.

---

quit

Call: `quit()`

Terminates the game. Actually, this function flags that the game is to be ended the next time a command will be parsed, so normal return codes should be passed back when using `quit()`. Hence, after calling `quit()`, you should always immediately execute an `abort` statement. See the example given in the description of the `input()` built-in function for an illustration of the use of `quit()`.

---

rand

Call: `rand(`*upper-limit*`)`

This function returns a random number from 1 to the *upper-limit* given. (The *upper-limit* must be a positive number.) See also the discussion of `randomize()`, below.

---

randomize

Call: `randomize()`

This function "seeds" the random number generator with a new value. The value is implementation-defined, and comes from such a source as the system clock. Until `randomize()` is called, the random number generator will always produce a fixed sequence of random numbers; once randomized, however, the sequence is unpredictable. For testing purposes, you can leave the call to `randomize()` out of your program, because the game will always run repeatably. Once the game is working, you can simply put a call to `randomize()` in your `init()` function in order to make the game unpredictable for your players.

---

remdaemon

Call: `remdaemon(`*function*`, `*value*`)`

Removes a daemon set previously with `setdaemon()`. The daemon will no longer be called. Note that if a given function has been set as a daemon multiple times, a call to `remdaemon()` only removes the first such daemon; hence, you must call `remdaemon()` as many times as you called `setdaemon()` on a particular function if you wish to cancel all calls to it. Note also that the *value* given must match the *value* used in the original call to `setdaemon()`; this allows you to set a daemon several times, with different contexts (the context being given by the *value*), and selectively remove the daemons later.

---

remfuse

Call: `remfuse(`*function*`, `*value*`)`

See also the `setfuse()` built-in function. This function removes a fuse previously set, or does nothing if the named function has not been set as a fuse. The fuse will not be executed if removed prior to burning down. Note that if a function has been set as a fuse multiple times, `remfuse()` will remove only the first such fuse set; hence, you must call `remfuse()` on a function as many times as you called `setfuse()` on the function if you wish to cancel all calls to it. Note also that the *value* given must match the *value* used in the original call to `setdaemon()`; this allows you to set a fuse several times, with different contexts (the context being given by the *value*), and selectively remove the fuses later.

---

restart

Call: `restart()`

This function starts the game over again from the beginning. It does not return.

---

restore

Call: `restore(`*filename*`)`

The *filename* is a string specifying an operating system file which was created with the `save()` function. The state of the previously saved game is restored, so that the players can continue where they left off.

If successful, `nil` is returned; otherwise, an error occurred restoring the game from the file. Note that the game to be restored must have been saved by the identical version of the TADS software *and* of the game itself. The system will refuse to restore a game saved by a different version of the TADS software, or by a different version of your game program (as determined by a timestamp stored in the game by the compiler).

---

save

Call: `save(`*filename*`)`

The *filename* is a string specifying an operating system file to be used to save the current status of the game. Everything about the game is saved in the file, so that the exact point in the game can be later restored with the `restore()` function.

If successful, `nil` is returned; otherwise, an error occurred saving the file.

---

say

Call: `say(`*value*`)`

The *value* can be either a number or a string. If it's a number, its decimal value is printed. A string is displayed as it is.

There is no return value.

---

setdaemon

Call: `setdaemon(`*function*`, `*value*`)`

Sets a daemon. Each turn, all outstanding daemons are called once, after all player command processing is completed. The *value* given is passed to the *function* as its sole parameter each time the function is called.

Note: the function must be known when the compiler reads this statement, which means the function must have been defined or at least called prior to the `setdaemon()` call in the source file. You can forward-declare a function by using a statement like this:

```
test:  function;
```

Note that the actual definition of the function `test` will appear later in the file; the statement above only declares it as a function so it can be used in the `setdaemon()` or similar call.

At most 100 daemons can be running simultaneously; a run-time error will result if a call to `setdaemon()` is made when 100 daemons are already running. There is no return value.

See also the `notify()` function.

---

setfuse

Call: `setfuse(`*function*`, `*time*`, `*value*`)`

Sets a fuse of the specified length. The *function* is the identifier naming a function. The *time* is a positive number specifying the number of turns before the fuse burns down. After the specified number of turns, *function* is called with *value* as its sole parameter.

Note: the function must be known as a function when the compiler reads this statement, which means the function must have been defined or at least called prior to the `setfuse()` call in the source file. See the note in the description of `setdaemon()`, above, for more information.

There is limit of 100 simultaneously pending fuses; a run-time error results if `setfuse()` is called when 100 fuses are already pending. There is no return value from `setfuse()`.

Note that fuses differ from daemons in that a fuse is only called once, at a time determined when the fuse is set; once called, it is automatically removed from the list of pending fuses. Daemons, on the other hand, are called once per turn until explicitly removed.

---

setit

Call: `setit(`*object*`)`

Sets the antecedant of "it," as used in a player's command, to the specified object. This is useful if a complaint refers to an object which will draw the player's attention; consider the following dialogue:

```
>go north
The metal door is closed.
>open it
There is a padlock preventing you from opening the door.
>unlock it
What do you want to unlock the padlock with?
>the key
The padlock swings open.
```

In this example, `setit()` has been used each time a complaint refers to an object which the user might wish to manipulate due to the complaint.

There is no return value.

---

setscore

Call: `setscore(`*score*`, `*turns*`)`

Alternative Call: `setscore(`*stringval*`)`

This function is used to inform the system of the current score and turn count. On versions of TADS that have a status line, this function updates the values on the status line. On versions of TADS without a status line, this call is ignored.

Your game should call this function whenever the score or turncounter change, so that the status line is always kept up to date. Note that this means that it should be called after using `restore()` to load an old game, and before calling `restart()` to start over from the beginning, in addition to being called whenever the player earns points or completes a turn. Generally, if you use the function `incscore()` defined in `adv.t` to add points to the score, and you use a standard turn counter function (such as the `turncount()` function defined in `adv.t`), you will not have to worry about calling `setscore()` too often.

The alternative form of the `setscore()` call allows you to display any text string in the score/turn counter area of the status line. In older versions of TADS, only the first form could be used, and the status line display was fixed with the score/turn counter format. In new versions, however, you can place whatever string you want on the right portion of the status line, simply by passing the string you wish to display to the `setscore()` function. You could, for example, display the time of day (within your game), or you could display the player's rank, experience, or other statistics rather than the score. The text string used in `setscore()` is displayed right-justified on the status line.

Note that several routines in `adv.t` (about four in all) call the numeric form of `setscore()`. If you use the string version, you should change the `setscore()` calls in `adv.t` to call a routine that you provide that displays the status line in the format you prefer.

---

setversion

Call: `setversion(`*version*`)`

Note: This function is no longer needed, although TADS continues to support it for compatibility with existing games written with older versions of TADS.

In older version of TADS, this function was used to identify the version of your game that created a saved game file. It was necessary to use `setversion` so that a saved position file created by one version of your game couldn't accidentally be used with another version of your game, because the information in a saved position file is only usable by the same version of a game that created it. The compiler now automatically generates a unique version identifier whenever your game is compiled, eliminating the need for you to do this manually. You no longer need to call this function, and it has no effect if you do.

---

substr

Call: `substr(`*string*`, `*offset*`, `*length*`)`

This function returns the substring of *string* starting at the character position given by *offset* (the first character's position is 1) and containing up to *length* characters. For example, `substr('abcdefghi', 4, 3)` returns the string `'def'`. If the given *offset* is past the end of *string*, a null string is returned; hence, `substr('abcdef', 10, 3)` returns a null string. (Note that a null string, specified by two single quotes, `''`, is *not* the same as `nil`.) If the *string* is too short to satisfy the requested *length*, the substring from the given *offset* to the end of the *string* is returned; for example, `substr( 'abcdefg', 4, 10 )` returns the string `'defg'`.

---

undo

Call: `undo()`

This function undoes all changes made since the last "savepoint" in the undo records. A savepoint is entered at the beginning of each turn automatically by the player command parser. Calling `undo()` once restores the game to its state at the very beginning of the current turn. Calling `undo()` again returns it to the state at the beginning of the previous turn, and so forth.

You should call `undo()` once in any situation where you only want to undo the current turn. For example, in a routine that gives the player options after being killed by an action taken on the current turn, you'd only want to call `undo()` once if the player elects to undo the action that killed him, since it's just the current turn that needs to be taken back.

You should call `undo()` twice in your verb that implements an "undo" command, if you have one in your game. The first call undoes the current turn (the "undo" command itself), and the second call undoes the previous turn—this is appropriate behavior, since the "undo" command should back up one turn. The default `undoVerb` defined in `adv.t` implements exactly this behavior.

The `undo()` function returns `true` if the undo operation was successful, or `nil` if no more undo information is available. A return value of `nil` means either that the player has already undone turns all the way back to the beginning of the game, or that he has undone turns as far back as the undo records go. The default undo area size will hold about a hundred turns in a typical game, but the actual number of turns that can be undone will vary by game, depending on the amount of work that needs to be undone. The undo mechanism will always keep as many of the most recent turns as possible, discarding the oldest turns in the undo log when it runs out of space.

---

unnotify

Call: `unnotify(`*object*`, &`*message*`)`

This function cancels the effect of a previous call to `notify()` with the same *object* and *message*. Note that the `&` before *message* is required.

See also the `notify()` function.

---

upper

Call: `upper(`*string*`)`

This function returns a copy of *string* with all of the lower-case letters converted to upper-case. Do not confuse this function with the `caps()` built-in function, which affects only the next character displayed.

---

yorn

Call: `yorn()`

Waits for the user to type a character, and returns `1` if the player typed "Y", `0` if the player typed "N", and `-1` for anything else. A suitable prompt should be issued prior to calling this function.

Our life is frittered away by detail. . .
Simplify, simplify.

— HENRY DAVID THOREAU, *Where I Lived, and What I Lived For* (1854)

The statements was interesting but tough.

— MARK TWAIN, *Adventures of Huckleberry Finn* (1884)

Chapter Six

Getting Started with TADS

The previous chapters have covered a lot of ground: the language, the parser interface, and even the general principles of object-oriented programming. Because TADS provides such a rich set of options, it's often difficult when you're first getting started with TADS to figure out the best way to do something.

This chapter is intended to help you through the process of writing a game with TADS. We'll assume that you have a basic idea of how the TADS language works.

Designing and Implementing

Your author has been involved in the design and implementation of several text adventure games. The game development process presented here reflects the experience gained from those projects.

The first step is designing the game. Using a system like TADS, which makes it very easy to get started with a game, you may be tempted to jump right in and start programming your game, but you should instead turn off your computer and get out a pad of paper.

We did not follow this advice with *Deep Space Drifter*. After formulating the basic plot of the game, and mapping out the portion that takes place on the space station (roughly the first half of the game), we started implementation. We had a basic idea of the second half of the game, but it wasn't even mapped.

Implementation went well for a while, but as we got further along, we started to run into details in the first half of the game that were dependent upon details from the undesigned second half. We improvised some details, and left others for later. As we did this, a strange thing happened: we started to realize that there were holes in the plot, and weird little inconsistencies that hadn't occurred to us until we needed to think about details. As a result, we started to change our basic ideas about the second half, which led to even more inconsistencies and plot holes. It was like digging

in sand, and before long we decided to throw out the entire original plan for the second half and start over.

However, we had so much time and effort invested in the space station that we didn't want to throw it away. Instead, we tried to design a new second half that fit in with the existing first half. From this point on, the battle was lost. We went through a series of essentially unrelated plots for the game, trying to fit each new plot to an even larger set of existing implementation. We'd plan a little, implement it, then discover that the plan wasn't working and would have to go—but the programming work we did would have to stay. The swamp, the cave maze, and the shuttle represented so much work that we couldn't contemplate throwing them away, so whatever we came up with had to include them somehow; for a brief time, we were actually going to make the swamp a "Swamp Simulator" because it was the only way we could make it fit.

In the end, we were totally sick of writing *Deep Space Drifter*, but refused to let the project die before it was complete for psychological reasons. To me, this attitude shows through in the last half of the game; I think there's a room on the planet whose description is something like this: "This room is very boring; you can leave to the north." In fact, I think the entire game reflects its history: the space station is full of things to do, it has some nice running jokes, and it's stylistically consistent. The planet, on the other hand, has an empty, barren feel; it's spread out and there's not much to do. The only parts that are interesting are essentially unrelated to each other and to the story in general, a reflection of having been forced into the game whether they belonged or not.

I'm not saying that *Deep Space Drifter* is a bad game—I like the space station a lot, and the puzzles on the planet are very elaborate and elegant. But the game has some serious flaws, most of which I attribute to the long, chaotic process of design and implementation.

Ditch Day Drifter, on the other hand, went through a much shorter and smoother design and implementation process, and I think a better game resulted. Admittedly, *Ditch* is a much smaller game than *Deep*, but it's clear to me that the same design process could have been applied *Deep*, and that doing so would have resulted in a much better game.

Before any programming work started on *Ditch*, I had a complete map of the game and descriptions of all the objects and chracters in the game, including the key elements of their behavior. This made implementation very smooth and easy, because I could simply go through the lists of rooms and objects and implement each. I didn't have to make any design decisions during implementation, which eliminated the desire to change design decisions that had already been made. The game took only a few weeks to implement, and came out fairly well.

If you resist the temptation to start programming before the design is complete, your game's implementation will go much faster, the game will turn out much better, and most importantly, the chances that you complete your game will be much higher.

Designing a Game

Before you start implementing your game, you should have a list of all of the locations, objects, and

characters in the game. For each location, object, and character in the game (to which we'll refer collectively as "game elements" henceforth), you should have a description of the item and its key behavior. Basically, you should know enough about your game that you can go through the game and play it completely through, command by command. You don't need to know *all* of the details of the game at this point, but you should know all of the details along the "minimal path" through the game (the series of commands that a player would type to go through the game from start to finish, if no mistakes are made and the player never needs to back up). The remaining details do not affect the plot, so you will be able to make these up as you go without creating plot holes or inconsistencies.

The complete map and object list is the *result* of the design process. To get there, you will probably go through many steps, adding detail as you go, until you have a complete design. The process detailed here is a good way to design a game, but you may find other ways that suit you better.

Plot

Most adventure games have a basic plot framework that controls the overall flow of the game. The plot is a good place to start your design.

An adventure game's plot isn't usually as elaborate as the plot of a non-interactive form of fiction, such as a book or a movie, because you have much less control over the details of the main character's actions. This makes plot design much more difficult, but you should give the player as much control as you can, since adventure games are always more satisfying when they give players a greater sense of control. Adventure game plots, therefore, should simply be a framework that provides the general direction of action in the game, and specifies only the important events.

You should start off with a basic background and goal. Where does the game take place? When? Who is the main character? What must be accomplished by the time the game is over?

For example, in *Ditch Day Drifter*, the location is the Caltech campus (or perhaps Caltech in some parallel universe, since the game isn't a strictly accurate simulation of the real Caltech), and the time is the present, generally, and Ditch Day, in particular. The main character is a Caltech underclassman. The goal is to solve the Ditch Day "stack" that the senior across the hall has created.

You can continue by identifying the major sub-goals that must be reached in order to reach the overall goal of the game. In *Ditch*, the major sub-goals are to find each treasure listed in the note that describes the stack.

Filling in the details of the plot can proceed by "working backwards" from the overall goal to the major sub-goals, then backwards to the smaller goals that must be reached for each sub-goal, and so on.

I would advise against designing a game with major sub-goals such as "the player must find the five rings of power." When the sub-goals are abstract like this, they don't suggest anything about what you might need to do to accomplish them. On the other hand, if a sub-goal is something concrete and specific like "the player must destroy the radio broadcasting tower," the minor sub-goals come almost automatically: you need to break into the compound where the broadcasting tower is located,

and you'll need some sort of explosive, and you may need to turn off power to the tower to avoid being electrocuted. But if you can turn off the power, why do you need to blow it up? Maybe because an attendant will come and turn the power back on after a few turns. Now we have another puzzle: dealing with the attendant. Sub-goals that are abstract, such as finding five rings of power, aren't nearly as helpful in getting to the next level of detail. Concrete sub-goals can often provide a seed that your imagination can almost automatically expand into a full set of plot elements.

Setting

The basic plot will provide an overall setting for the game, and the plot elements will usually suggest a specific set of large-scale locations. So, you'll start off with a "low-resolution" map simply by designing the basic plot.

Next, you can start filling in the specific rooms. At this point, you'll probably find that you start to fill in the details of the plot at the same time as you're filling in the details of the setting. Many locations in the setting will offer natural puzzles: if you have a bank, you'll probably have a safe, for which you'll need to find the combination; maybe you'll need to figure out how to defeat the alarm system, or perhaps you'll need something to distract the teller; maybe you'll need to find the key to the safety deposit box. If you have an airport, you'll probably need to figure out how to get past the overly sensitive metal detectors, or you'll need to know someone's itinerary, or you'll need to find some airline tickets, or you'll need to find a badge or key or combination to get into a restricted area.

Whatever type of location you're designing, the location will often suggest a series of puzzles to be solved. As you design the solutions to the puzzles, you'll be adding detail to the plot, which will in turn give you new locations to develop.

As you flesh out the setting, you may be tempted to add enormous amounts of space to your locations. For example, if you're building an airport, you may find yourself putting in dozens of gates, each pretty much the same as all the others. While the added space may make the game setting more like a real airport, it can often harm the game's playability. Remember, you're designing a game, not an airport—the most important thing is making the game fun to play, not "real."

The main problem with adding lots of essentially unused space to a game is that it tends to make the level of detail throughout the game less consistent. You should make an effort to keep the level of detail as consistent as possible throughout the game, to avoid annoying and confusing the player. If you have a few rooms with a great deal of detail (such as lots of objects that can be examined and manipulated), the player will come to expect that level of detail in other rooms. The dozens of almost identical airport gates will not have anything interesting to do.

Objects and Characters

You'll probably end up designing most of the game in the course of mapping out the setting and plot. As you develop plot elements, you should make notes of the objects and characters that are

involved. You'll probably find it easiest to note objects and characters on the map, where they'll be placed.

For each object and character, you should make notes about its relevance to the game, and what it does. You should think carefully about what other objects might be used for the same purpose, and what other uses an object might have. For example, if you have a door that you need to break down, and you intend to provide an axe for this purpose, you should consider two things: What else might the axe reasonably be expected to destory? And what else might reasonably be expected to be able to break down the door?

Because a player could reasonably expect an axe to have many uses, and because many objects should be able to break down a door, you should be very careful about creating puzzles that involve brute force and powerful tools of general utility. Explosives, flammable liquids, weapons, and heavy tools are all likely to frustrate the player when they don't remove just about any physical obstacle. On the other hand, it would be very satisfying if you took the trouble to fully realize the axe and the door by allowing the axe to break down everything that's reasonable, and allowing the door to be broken down by other reasonable objects.

Implementation

Once you have the game fully designed, you're ready to start implementing. In this section, we'll show how to implement the basic types of objects.

Start off by drawing the map, and making annotations on the map describing the essential details of the plot. This should include placement of the major objects in the game, and descriptions of the important actions the player must perform.

As an example, we'll implement a game that takes place in a small airport. Our airport will have a terminal area, a concourse, and a gate area. We'll also have a plane parked at one of the gates. The terminal area will have a ticket counter, and a metal detector leading into the concourse. In the concourse, there will be a snack bar, and a locked door leading off into a security area. The gate area will have a couple of gates, plus a locked maintenance room. Here's the basic map we'll be implementing.

This map conveniently has a couple of locked areas, which we can use for puzzles. In addition, we can probably find some use for the metal detector, since it will prevent the player from carrying any objects through it. The plane can also be a puzzle, since you'd normally need a ticket to board a plane. Plus, the cockpit should be restricted to airline personnel.

Let's make the goal of the airport segment be getting out of the airport. The player will start off in the main terminal area, but won't be able to go outside the terminal—we'll make up some excuse, such as heavy traffic that always pushes the player back into the terminal, to prevent exiting that way. (If this were an actual game, we'd probably have more game outside the airport, so we wouldn't use such an artificial boundary as having heavy traffic that pushes the player back in. For this example, though, we want to keep things fairly small.) The only other obvious way to get out of the airport is to fly out on a plane; so, let's make the goal be to fly the plane.

To take the plane out of the airport, the player will have to get into the cockpit. (We'll implement this example up to the point that the player makes it into the cockpit; in a full game, we'd go on to let the player fly the plane somewhere else.) Now, only the pilot can go into the cockpit—the flight attendant wouldn't let a passenger into the cockpit. So, we'll need some way to get past the flight attendant. One way would be to create a diversion that distracts the flight attendant long enough

to slip by; for this example, though, we'll require the player to find a pilot's uniform.

Where would the pilot's uniform be? The pilot's lounge would be a logical place; we'll put a suitcase in the pilot's lounge that contains a uniform. Fortunately, the lounge is behind a locked door, which creates a secondary puzzle. To get into the security area that contains the pilot's lounge, the player needs a magnetic ID card to put into a slot outside the security area.

We'll put the ID card out in the open, on the ticket counter in the terminal area. However, we'll make it impossible to carry the ID card past the metal detector—the card will set off the metal detector, and the security guard will confiscate it (and place it back on the counter so that the player can try again). To get the card by the metal detector, you'll have to turn off the power to the metal detector.

The power switch will be in the maintenance room, which is locked with a key. We'll leave the key with some other maintenance items in the plane's bathroom—we'll also leave a pail, sponge, and garbage bag, so that it's clear by association that the key is probably for the maintenance room.

To get into the plane's bathroom, you'll need a ticket to board the plane. We'll make the ticket fairly simple to find: we'll leave it hidden inside a newspaper in the snack bar. As soon as you pick up the newspaper, the ticket will fall out.

So, that's about the whole game: you go to the snack bar, pick up the newspaper, and find the ticket. You take the ticket and board the plane, then go to the plane's bathroom and get the key. Take the key to the maintenance room, unlock the door, enter, and turn off the power to the metal detector. Go back to the ticket counter, pick up the ID card, go to the security door, put the magnetic card in the slot, and enter the security area. Go to the pilot's lounge, get the pilot's uniform out of the suitcase, and wear it. Go to the plane, and stroll right past the flight attendant and into the cockpit.

We should draw a new map now, which includes annotations for the main objects and actions that make up the game.

Implementing the Map

The first step is to convert the skeleton of your map into the beginnings of your game program. At this point, don't worry about entering all of the detailed behavior of your map, but just build the basic rooms and connections between rooms. This will allow you to get a prototype of the game running quickly, so you can walk around it and see how it feels.

Start off by creating a source file for your game, and including the base definition files:

```
#include <adv.t>
#include <std.t>
```

Now, for each room in your game, make an entry that gives the room's name (`sdesc`, or short description) and long description (`ldesc`), and the other rooms that are connected to this room. Here's how to implement the first few rooms from the sample map above. For the time being, we

won't worry too much about making the long descriptions complete; we can always flesh those out later.

```
terminal:  room
  sdesc = "Terminal"
  ldesc = "You are in the airport's main terminal.  To the east,
  you see some ticket counters; to the north is the main concourse."
  east = ticketCounter
  north = securityGate
;

ticketCounter:  room
  sdesc = "Ticket Counter"
  ldesc = "You are in the ticket counter area.  Ticket counters
  line the north wall; so many people are waiting in line that
  you're sure you'll never manage to get to an agent.  The main
  terminal is back to the west."
  west = terminal
;

securityGate:  room
  sdesc = "Security Gate"
  ldesc = "You are at the security gate leading into the main
  concourse and boarding gate areas.  The concourse lies
  to the north, through a metal detector.  The terminal is
  back to the south."
  north = concourse
  south = terminal
;

concourse:  room
  sdesc = "Concourse"
  ldesc = "You are in a long hallway connecting the terminal
  buliding (which lies to the south) to the boarding gates (which
  are to the north).  To the east is a snack bar, and a door leads
  west.  Next to the door on the west in a small slot that looks
  like it accepts magnetic ID cards to operate the door lock."
  north = gateArea
  south = securityGate
  east = snackBar
  west = securityArea
;
```

The other rooms are implemented in the same manner. For now, we're not worrying about the items contained in the rooms, or the other people around, or even the locked doors. We'll just implement all the rooms so we can walk through the map and try it out.

Implementing the items

The next step is to implement the basic objects that make up the game. As with the rooms, don't worry about the complex behavior of some of the objects at this point; just go through and write basic object definitions for the main items in the game.

Items have different properties than rooms. The basic properties of an item are its name (`sdesc`), long description (`ldesc`), vocabulary words (`noun` and possibly `adjective`), and container (`location`, which may be either a room or another item). If the item can be carried by the player, the object will be of class `item`; if not, it will be of class `fixeditem`. Some items may be of different classes; for example, if you want to make an object that can contain other objects (such as the pail), make it a `container`. If you want to be able to put objects on top of another object, use a `surface` object. You can make something both a `container` or `surface` and a `fixeditem` if you want.

Here are some of the basic object definitions for our sample game.

```
counter:  fixeditem, surface
  location = ticketCounter
  noun = 'counter'
  adjective = 'ticket'
  sdesc = "ticket counter"
;

IDcard:  item
  location = counter
  noun = 'card'
  adjective = 'id' 'identification'
  sdesc = "ID card"
  adesc = "an ID card"
;

newspaper:  readable
  location = snackBar
  noun = 'newspaper' 'paper'
  adjective = 'news'
  sdesc = "newspaper"
  ldesc = "It's today's copy of USA YESTERDAY."
  readdesc = "You read a few articles, and promptly become
  depressed.  The federal deficit just went up by another
  twenty billion dollars, but it's all \"off budget,\"
  so it doesn't really count.  There's another White House scandal
  involving illegal arms sales, money laundering through Italian banks,
  Congressional Pages; several high-ranking federal arts critics have
  already resigned in disgrace.  The economy had yet another downturn,
  but the President says he's confident that the recovery is \"just
```

```
  around the corner and picking up steam.\""
;

cardslot:  fixeditem
  location = concourse
  noun = 'slot'
  adjective = 'card'
  sdesc = "card slot"
  ldesc = "The slot appears to accept special ID cards with
  magnetic encoding.  If you had an appropriate ID card, you could
  put it in the slot to open the door."
;

suitcase:  openable
  isopen = nil
  location = pilotsLounge
  noun = 'suitcase'
  sdesc = "suitcase"
;

uniform:  clothingItem
  location = suitcase
  noun = 'uniform'
  adjective = 'pilot' 'pilot\'s'
  sdesc = "pilot's uniform"
  ldesc = "It's a uniform for an Untied Airlines pilot.  It's
  a little large for you, but you could probably wear it."
;
```

The rest of the objects are implemented in much the same way. In implementing the basic objects, the main properties you need to fill in are `location`, so the object appears somewhere in the game; `noun`, so the player can refer to the object; and `sdesc`, so the system knows what to call the object when it mentions the object in messages. It's also important to choose the appropriate class for each item, so that you can get the correct basic behavior of the object without any additional work. At some point, you should go through `adv.t` and familiarize yourself with the classes defined there, so you know what you can get from `adv.t` classes without any work. See Appendix A for full details on `adv.t`.

Implementing Special Behavior

The next step is to implement all of the special behavior that really makes the game work. For example, let's implement the simple mechanism that lets the player find the airline ticket upon picking up the newspaper. To do this, all we need to do is add a `doTake` method to the `newspaper` object (we'll just show the new `doTake` method below; the rest of the object is the same as shown above):

```
doTake(actor) =
{
  if (not self.foundTicket)
  {
    "As you pick up the paper, an airline ticket that was
    inside falls to the floor.";
    ticket.moveInto(actor.location);
    self.foundTicket := true;
  }
  pass doTake;
}
```

This method runs whenever the player takes the newspaper. The first thing we do is check to make sure that the airline ticket hasn't already been found; if not, we display a message that it's been found, move the ticket into the game, and note that it's been found. Finally, we continue with the default `doTake` action that the `newspaper` object inherited from its superclass (in this case, `readable`) by using the `pass` statement.

Note that the airline ticket itself should be defined with no location, because it's not anywhere at all when the game first starts:

```
ticket:  item
  noun = 'ticket'
  adjective = 'airline'
  sdesc = "airline ticket"
  ldesc = "It's a one-way ticket to New York, in class C
  (the \"C\" probably stands for \"Cattle\")."
;
```

As another example, let's implement the puzzle the keeps the player out of the security area until the ID card is used to unlock the door. First, we must prevent movement from the concourse into the security area. For this, we'll change the concourse room definition, and add a door object.

```
concourse:  room
  sdesc = "Concourse"
  ldesc = "You are in a long hallway connecting the terminal
  building (which lies to the south) to the boarding gates (which
  are to the north).  To the east is a snack bar, and a door leads
  west.  Next to the door on the west is a small slot that looks
  like it accepts magnetic ID cards to operate the door lock."
  north = gateArea
  south = securityGate
  east = snackBar
  west =
  {
    if (securityDoor.isopen) return(securityArea);
```

```
    else
    {
      "The door is closed and locked.";
      return(nil);
    }
  }
;

securityDoor:  fixeditem
  location = concourse
  noun = 'door'
  sdesc = "door"
  isopen = nil
  ldesc =
  {
    "The door has a label reading SECURITY AREA--AUTHORIZED
    PERSONNEL ONLY. ";
    if (self.isopen) "The door is open, which isn't very secure.";
    else "The door is securely closed.";
  }
  verDoOpen(actor) =
  {
    "The door is securely locked.";
  }
  verDoUnlock(actor) =
  {
    "You should examine the slot if you want to unlock the door.";
  }
;
```

The door is really just a decoration item; the various methods we implement are just to inform the user that the door can't be operated directly. The real work is done by the card slot; we'll add some new behavior to that object now.

```
cardslot:  fixeditem
  location = concourse
  noun = 'slot'
  adjective = 'card'
  sdesc = "card slot"
  ldesc = "The slot appears to accept special ID cards with
  magnetic encoding.  If you had an appropriate ID card, you could
  put it in the slot to open the door."
  verIoPutIn(actor) =
  {
  }
```

```
    ioPutIn(actor, dobj) =
    {
      if (dobj = IDcard)
      {
        if (securityDoor.isopen)
          "You put the card in the slot; nothing happens,
          so you remove it.";
        else
        {
          "You put the card in the slot.  There's a click, and
          the security door pops open!  You remove the card.";
          securityDoor.isopen := true;
        }
      }
      else
        "That doesn't seem to fit in the slot.";
    }
  ;
```

The new behavior is that the player can put the ID card in the slot. When this is done, the `ioPutIn` method runs, with the ID card as the direct object (the `dobj` parameter). This method opens the door, if it's not already open.

Most of the remaining puzzles in this sample game are implemented in a similar fashion. You'll need to implement a couple of actors: one for the flight attendant, and one for the guard at the metal detector. In addition, you'll need a few more locked doors and special items.

We won't go any further into the other puzzles, because the next chapter describes in much greater detail how to implement these and much more.

Where to go from here

Before expanding this sample game by adding more areas, you could flesh out the game by adding lots of detail to what we've implemented so far. Most of the added material would probably be irrelevant to the plot, but it could make the game more interesting and more fun to play. Keep in mind that you're writing a game, not a real-world simulator; try to concentrate on things that make the game more fun to play.

Some items that would enhance the airport: Add lots of incredibly overpriced and extremely dubious snack items at the snack bar. Generate random messages over the public address system once in a while, asking various people to pick up the white courtesy telephone and warning travellers not to park in the red zone. Implement other effects for other power switches in the maintenance room: what happens when you turn off power to the snack bar, or the ticket counter, or the PA system, or

the automatic doors? Add lots of strange people milling about in the airport; make some of them actively bother the player, such as various fringe religious and political groups trying to hand out literature (you'll want to write some wacky literature for them to distribute).

Alternatively, you could expand this game by providing some place to fly to. The player could go on to crash the plane into a remote mountain or desert island, and explore that. Or, you could take the plane to several other cities and explore.

Of course, you'll probably have the most fun if you start with your own ideas and write a whole new game. The hard part is always finding the right idea; once you have the premise for your game, you will probably be surprised at how quickly you can build a map and start implementing. Refer to the examples in this chapter to help you get started. As your game starts to take shape, and you want to add more ambitious features, the examples of advanced TADS programming techniques shown in the next chapter should be helpful.

In creating, the only hard thing's to begin; a grass-blade's no easier to make than an oak.

— JAMES RUSSELL LOWELL, *A Fable for Critics* (1848)

Chapter Seven

Advanced TADS Techniques

So far, we've concentrated on the basic aspects of writing a game with TADS. Once you get started on a game, you will probably have lots of ideas for plot elements in your game. Many parts of an adventure game can be implemented very easily with TADS, simply by piecing together the basic classes from `adv.t` and filling in descriptions, locations, and so forth. However, some of the things you will want to implement won't be that easy, and you will need to do some programming.

This chapter is intended to help you realize your more complex ideas by showing how to implement a number of sophisticated plot elements. We've tried to show examples of the most common difficult game features, so there's a good chance that you'll find an example in this chapter that's very similar to what you're trying to do. Even if you don't find a close match to the feature you have in mind, reading this chapter will show you many of the more subtle points of TADS programming, which should give you a better idea of how to implement your ideas.

Creating your own Verbs

It's always best to try to limit the number of verbs a player needs to use to complete your game, because it helps prevent your adventure from becoming a game of "guess the verb." Choosing the right sentence to express an idea for a command should never be a challenge—if it is, your game will quickly become uninteresting.

However, that being said, you will sometimes want to include situations in your games that call for verbs that aren't in the set defined in `adv.t`. In addition, there's nothing wrong with making your game understand obscure verbs and phrasings that are *alternatives* to one of the more common verbs in `adv.t`. Furthermore, some games include "magic words" that are effectively special verbs; since the player learns these within the game, obscurity is not a problem.

The way you add a verb depends on how the verb is used. The simplest kind of verb is used by itself—the player doesn't include any objects in a command with the verb. For example, "north"

is a verb that doesn't take any objects; magic words generally fall into this category as well. To implement a verb with no objects, all you have to do is create a `deepverb` object that includes a `verb` vocabulary property, an `sdesc` property, and an `action(actor)` method. (The `deepverb` class is defined in `adv.t`; see Appendix A for details.) The `action` method is called when the verb is used with no objects. As an example, let's implement a verb for a magic word; when the magic word is used, the player will be magically transported from the cave to the shack, or the shack to the cave.

```
magicVerb:  deepverb
  verb = 'xyzzy'
  sdesc = "xyzzy"
  action(actor) =
  {
    if (actor.location = cave)
    {
      "Out of nowhere, a huge cloud of orange smoke
      fills the cave.  When it clears, you suddenly
      realize that you're in the shack!";
      actor.moveInto(shack);
    }
    else if (actor.location = shack)
    {
      "You suddenly feel very disoriented, and the
      room seems to be spinning all around you.  As you
      gradually regain your balance, you realize that
      you're now in the cave!";
      actor.moveInto(cave);
    }
    else
      "Nothing happens.";
  }
;
```

For a verb that takes a direct object, the verb object itself is fairly simple; all you need to do is define `verb` and `sdesc` properties, plus the special property `doAction`. The `doAction` property defines a string that the system uses to form the names of two new properties. As an example, we'll implement the verb "smell," which takes as its direct object the item to be smelled.

```
smellVerb:  deepverb
  verb = 'smell'
  sdesc = "smell"
  doAction = 'Smell'
;
```

When the system sees this definition, it creates two new properties: `verDoSmell` and `doSmell`. Now, for any object that the player can smell, you add the `verDoSmell` and `doSmell` methods. For

example, we'll implement a flower (notice that we deftly bypass the opportunity for a cheap joke here at the expense of good taste by choosing an object with a non-offensive odor):

```
flower:  item
  noun = 'flower' 'rose'
  adjective = 'red'
  sdesc = "red rose"
  ldesc = "It's a red rose."
  verDoSmell(actor) =
  {
  }
  doSmell(actor) =
  {
    "A rose by any other name...";
  }
;
```

If the `verDoSmell` method isn't defined for an object, trying to smell it will cause the system to display the message "I don't know how to smell the *object*." Since "smell" is a verb that you may want to allow for every object in your game, you may want to modify the definition of `thing` in `adv.t` to provide a better default message for this verb. You could add these lines to the definition of `thing` in `adv.t`:

```
verDoSmell(actor) =
{
}
doSmell(actor) =
{
  "It smells like any other <<self.sdesc>>.";
}
```

If you do this, all you need to do is override the `doSmell` method for any object with a non-default odor.

Adding a verb that takes two objects (a direct object plus an indirect object) is similar. Rather than defining a `doAction` property, you define an `ioAction(`*preposition*`)` property. For example, suppose you want to implement a verb "pry *direct-object* with *indirect-object*":

```
pryVerb:  deepverb
  verb = 'pry'
  sdesc = "pry"
  ioAction(withPrep) = 'PryWith'
  prepDefault = withPrep
;
```

This causes the system to create the properties `verIoPryWith`, `verDoPryWith`, and `ioPryWith`, which the system will call in that order (`verDoPryWith` is called for the direct object, and the other

two are called for the indirect object.) Note that we've also defined a default preposition; while this isn't required, it allows the system to be smarter by guessing what the player meant when some words are left out. For example, if the only object present that can be used as an indirect object in the "pry" command is a crow bar, the system can provide the "with the crow bar" phrase by default when the player simply types "pry the door."

Whether or not you add verbs to the basic set in `adv.t`, it's a good idea to document your verbs. In the instructions for your game, you should provide a list of all of the verbs that are needed to complete the game. If players gets stuck, this list can be helpful in convincing them that they're not fighting the parser.

Complex Room Interconnections

The basic travel routines used by `adv.t` make it easy to implement a basic map: all you have to do in your definitions is to provide properties with names like `north` and `south`, and set them to point to rooms, simply by naming the room:

```
kitchen:  room
  north = hallway
  east = porch
;
```

This is fine for most situations, but frequently you'll want something to happen during travel that's more complicated than simply moving the player to the given room. For example, if you have a very long hallway, you may want a special message to be displayed when travelling to indicate that the walk was especially long. Or, you may want to set off a trap when the player takes a certain exit. Or, you may want to prevent travel in a particular direction until a door has been opened or some other obstacle has been removed.

The basic trick to implementing special travel effects like these is to realize that the direction properties of a room aren't limited to simple room pointers—you can write a method instead. As long as you follow a few simple rules, you can do whatever you want in this method. Basically, if you want normal travel to occur after the method is finished, the method should return a room object which indicates the destination of the travel. If you don't want anything more to happen after the method is finished, the method should return `nil`. The following sections show how to use direction methods to implement a variety of special travel effects.

Display a Message During Travel

One of the simplest special effects is displaying a message during travel. For example, suppose you have a stairway leading down, but the stairs collapse while the player is descending them, dropping

the player unceremoniously into the room below (with no harm done). All you really want to do here is to display a message about the stairs collapsing, then complete the travel as normal.

```
livingRoom:  room
  sdesc = "Living Room"
  ldesc = "You're in the living room.  A dark stairway leads
  down."
  down =
  {
    "You start down the stairs, which creak and moan and
    wobble uncertainly beneath you.  You stop for a moment
    to steady yourself, then continue down, when the entire
    stairway suddenly breaks away from the wall and crashes
    to the floor below.  You land in a heap of splintered
    wood.  After a few moments, everything settles, and you
    manage to get up and brush yourself off, apparently
    uninjured.\b";

    return(cellar);
  }
;
```

The "\b" is at the end of the message to separate the message about the stairway's collapse from the descriptive text that accompanies normal travel into a new room. Since the method just returns a room object, travel will proceed as normal after the message has been displayed.

Simple Obstacles

The collapsing stairway of the previous example will need some additional special consideration at the other end, because the player clearly can't go back up the stairs once they've collapsed. The simplest case is to disallow travel altogether, which is simple: just make an `up` method that displays an appropriate message, then returns `nil` to indicate that no travel is possible in this direction.

```
cellar:  room
  sdesc = "Cellar"
  ldesc = "You're in the cellar.  A huge pile of broken pieces of
  wood that was once a stairway fills half the room.  "
  up =
  {
    "The stairway is in no condition to be climbed.  ";
    return(nil);
  }
;
```

Since the method returns `nil`, no further travel processing takes place after the message is displayed.

Note that we should add an object for the broken stairway, just for completeness, so that the player can mention the stairs in a command. We'll make it a `fixeditem` since it can't be moved or otherwise manipulated.

```
brokenStairs:  fixeditem
  location = cellar
  noun = 'stairs' 'stairway' 'pile' 'wood'
  adjective = 'broken'
  sdesc = "pile of wood"
  ldesc = "It's a huge pile of wood that used to be a stairway.
  You won't have much luck trying to climb it.  "
;
```

Adaptive Room Descriptions

The stairway examples shown above could be enhanced to make it possible to enter the cellar by some other passage, which would mean that the stairway could be seen before it collapses. If this were done, it would be necessary for the cellar's room description, as well as the `up` method, to be sensitive to whether the stairway had collapsed yet.

Making a room description that adapts to the state of the room is done by turning `ldesc` into a method. Just as special travel effects can be achieved by making a direction property into a method, special room descriptions can be programmed by making `ldesc` a method.

Since the stairway can be seen before it's collapsed, we'll introduce a new object for the working stairs. Note that this object should be climbable, so we'll define `verDoClimb` and `doClimb` methods for it. Note that the method `actor.travelTo(`*location*`)` is the exact same method that is called when the player travels normally by typing a direction command (such as "up"). Also note that the `location` property of the `brokenStairs` object should initially not be set (which will make it `nil`), since the broken stairway is not part of the game initially in the new configuration.

```
workingStairs:  fixeditem
  location = cellar
  noun = 'stairs' 'stairway'
  sdesc = "stairs"
  ldesc = "The stairway leads up.  It looks extremely
  rickety.  "
  verDoClimb(actor) = {}
  doClimb(actor) =
  {
    actor.travelTo(livingRoom);
  }
;
```

Now we'll change the `cellar` object's `up` method, as well as its `ldesc` property, to reflect the current status of the stairs. We'll detect whether the stairs have collapsed yet by checking the location of the `workingStairs` object: if it's in the cellar, we'll know the stairs haven't collapsed yet, and if it's not anywhere (that is, if it's location is `nil`), we'll know that the stairs have collapsed. (This object switch will be implemented below, in the `livingRoom` object's `down` method.) We'll use this detection technique to make both the `ldesc` and the `up` method sensitive to the status of the stairway.

```
cellar:  room
  sdesc = "Cellar"
  ldesc =
  {
    "You're in the cellar.  ";
    if (workingStairs.location = nil)
      "A huge pile of broken pieces of wood that was
      once a stairway fills half the room.  ";
    else
      "A suspicious-looking stairway leads up.  ";
  }
  up =
  {
    if (workingStairs.location = nil)
    {
      "The stairway is in no condition to be climbed.  ";
      return(nil);
    }
    else
    {
      "The stairs creak and moan and sway, but you
      somehow manage to climb them safely.\b";
      return(livingRoom);
    }
  }
;
```

We'll also have to make a small change to the `down` method in the `livingRoom` object, to replace the `workingStairs` object with the `brokenStairs` object. Here's the new `down` method, which will remove the `workingStairs` object from the game by moving it to `nil`, and put the `brokenStairs` object in its place.

```
livingRoom:  room
  sdesc = "Living Room"
  ldesc = "You're in the living room.  A dark stairway leads
  down."
  down =
```

```
  {
    if (workingStairs.location = nil)
    {
      "You quickly notice that the stairs have
      collapsed, so there's no way down.  ";
      return(nil);
    }
    else
    {
      "You start down the stairs, which creak and moan and
      wobble uncertainly beneath you.  You stop for a moment
      to steady yourself, then continue down, when the entire
      stairway suddenly breaks away from the wall and crashes
      to the floor below.  You land in a heap of splintered
      wood.  After a few moments, everything settles, and you
      manage to get up and brush yourself off, apparently
      uninjured.\b";

      workingStairs.moveInto(nil);               /* replace workingStairs...  */
      brokenStairs.moveInto(cellar);                /* ...  with brokenStairs */
      return(cellar);
    }
  }
;
```

Doors

One common type of obstacle is a door. Doors can be implemented using exactly the same mechanisms shown above for the stairway: in the directional properties for the rooms involving the door, check the status of the door object (generally the `isopen` property), and either return a room object if the door is open, or display a message and return `nil` if the door is closed. This type of mechanism is used in `DITCH.T` for its doors.

However, `adv.t` provides a set of classes that makes it easier to implement doors. The classes are based on a general class called `obstacle` (which you may wish to extend into your own specialized classes to implement different types of obstacles). The basic class is `doorway`, and an additional class called `lockableDoorway` makes it easy to implement a door that can be locked and unlocked with a particular key object.

By using the `doorway` and `lockedDoorway` classes, you can implement doors with no programming, because the basic travel routines will do the necessary work based on properties defined in your doorway objects.

The first thing to note about doors is that they have an unusual property: a door exists in two rooms, because it connects the two rooms. Unfortunately, TADS doesn't have any good way of putting a single object in multiple locations. To work around this problem, you should make *two* objects for each actual door in your game: one for each side. Fortunately, the `doorway` class makes it easy to keep the states of the two objects synchronized: for example, when you open one side of the door, the other side is automatically opened.

Start by implementing the two sides of the door. Make a `doorway` object for each, filling in the normal properties for any object (`location`, `sdesc`, `noun`). Put one `doorway` object in each room to be connected. Now add the special properties for a doorway. Set `otherside` to the other door object making up the pair of doors; this is how the door's internal routines know what other door to keep synchronized. Set `doordest` to the room object that's reached by travelling through the door.

Then, all you need to do for the rooms containing the doors is to set the direction properties to the doors themselves, rather than the rooms reached through the doors. Here's an example, showing the two room objects and the two door objects.

```
porch:  room
  sdesc = "Porch"
  ldesc = "You're on a porch outside a huge run-down wooden house
  that was probably painted white in the distant past,
  but which is gray and faded now.  A door (which is
  << porchDoor.isopen ?  "open" :  "closed" >>) leads west."
  west = porchDoor                              /* use door for the direction */
;

frontHall:  room
  sdesc = "Front Hall"
  ldesc = "You're in the spacious front hallway of the old house.
  The paint on the walls is all peeling, and a thick veil
  of spiderwebs hangs down from the ceiling.  A door
  (which is <<hallDoor.isopen ?  "open" :  "closed">>)
  leads east."
  east = hallDoor
;

porchDoor:  doorway
  location = porch
  noun = 'door'
  sdesc = "door"
  otherside = hallDoor
  doordest = frontHall
;

hallDoor:  doorway
```

```
    location = frontHall
    noun = 'door'
    sdesc = "door"
    otherside = porchDoor
    doordest = porch
;
```

Note that there's no need to define `ldesc` for a doorway, unless you want to. The default `ldesc` defined in the `doorway` object displays a message saying simply "It's open" or "It's closed" or "It's closed and locked," depending on the door's state. If you want to override `ldesc`, you can use the `isopen` and `islocked` properties to determine the state of the door to be displayed in your custom message.

Some more properties of the `doorway` class are worth mentioning. If the door is not locked, the door will be opened *automatically* when the player tries to travel through it (such as by going west from the porch in the example above). This makes the game more convenient, because it's not necessary to type "open the door" when that's the obvious thing to do. However, in cases where you don't want a door to be automatically opened, you can simply add `noAutoOpen = true` to the `doorway` object's properties; this requires the player to explicitly open the door, even when it's not locked.

If you want to require a key to lock and unlock the door, use the `lockableDoorway` class instead of `doorway`. Then, add the `mykey` property, setting it to the object which serves as the key (this object should be of class `keyItem`). If you don't set the `mykey` property, the door can be locked and unlocked without any key; that is, the player simply types "lock door" and "unlock door" to lock and unlock it.

Note that there's no requirement that the two sides of a door be identical. You can take advantage of this in certain situations. For example, if you want to have a door that needs a key from the outside, but doesn't need any key from the inside (like most doors you'd find in a house), make both sides `lockableDoorway` objects, but only set the `mykey` property in the object that serves as the outer side of the door.

Vehicles

Vehicles are rooms that move. Basically, a vehicle is an extension to the idea of complex room interconnections; rather than having a room interconnection that merely displays a message, or sometimes disallows travel, a vehicle has a room interconnection that moves around. Usually, a vehicle will also have some other behavior, such as controls that operate the vehicle, or a viewscreen that shows the area outside the vehicle.

The first type of vehicle we'll address is the fully-enclosed variety, such as a subway train or an elevator. We'll address vehicles that are also objects in their own right, such as rubber rafts, in a later section ("Nested Room Vehicles" below). A fully-enclosed vehicle really just amounts to a

room that has a direction property that changes, depending on where the vehicle is supposed to be. The vehicle doesn't have a `location` property, because it's just an ordinary room—it's not an object in other rooms.

As an example, let's implement an elevator that travels between two floors. The elevator will have "up" and "down" buttons to activate it, doors, and a display indicating which level it's on.

To make the elevator more realistic, we'll arrange for it to spend a couple of turns in transit as it moves between floors. To do this, we'll use a "notifier" function that's set when a button is used to start the elevator on its way. We'll also use a flag to indicate when the elevator is in transit, so that pushing the buttons while the elevator is moving will have no effect.

The doors will be simpler than normal doors, since the user can't directly manipulate them. They'll open and close automatically.

```
elevDoors:  fixeditem
  location = elevRoom
  noun = 'door' 'doors'
  adjective = 'elevator' 'sliding'
  sdesc = "elevator doors"
  ldesc = "They're a standard pair of sliding elevator doors.
  They're currently <<self.isopen ?  "open" :  "closed">>.  "
  isopen = true
  verDoOpen(actor) =
  {
    "The doors are automatic; you can't open them yourself.  ";
  }
  verDoClose(actor) =
  {
    "The doors are automatic; you can't close them yourself.  ";
  }
;
```

The buttons are fairly simple: they use the basic `buttonitem` class. We'll set a property indicating the destination of the elevator when that button is pushed—the "up" button will take the elevator to the upper floor, and the "down" button will take the elevator to the lower floor. The `doPush` method checks to see if the elevator is already moving, and then to see if the elevator is already at the destination of this button.

To make less work for ourselves, we'll define a class for the two elevator buttons, then make two objects based on the class.

```
class elevButton:  buttonitem
  location = elevRoom
  doPush(actor) =
  {
    /* ignore if already in motion or already at destination */
```

```
    if (elevRoom.isActive or elevRoom.curfloor = self.destination)
      "\"Click.\"";
    else
    {
      "The doors close, and the elevator starts to move.";
      elevRoom.isActive := true;
      elevDoors.isopen := nil;
      elevRoom.counter := 0;
      elevRoom.curfloor := self.destination;
      notify(elevRoom, &moveDaemon, 0);
    }
  }
;

elevUpButton:  elevButton
  sdesc = "up button"
  adjective = 'up'
  destination = floor2                    /* where to go when button is pushed */
;

elevDownButton:  elevButton
  sdesc = "down button"
  adjective = 'down'
  destination = floor1
;
```

You'll recall that `notify` is a built-in function that sets up a "daemon," which is a routine that you want the system to call after each turn. The call to `notify` in the example above tells the TADS run-time system to call the method `moveDaemon` in the object `elevRoom` after every turn (the third argument is `0`, which means that the method should be called after each turn; if it had been non-zero, it would have specified a number of turns to wait before calling the method once).

The elevator object needs to be sensitive to its current location with its `exit` and `north` parameter. It also needs to be sensitive to the state of the doors. The daemon (that is started with the `notify` call in the buttons) controls the travel of the elevator.

```
elevRoom:  room
  sdesc = "Elevator"
  ldesc = "You're inside a small elevator.  Sliding doors (which are
  currently <<elevDoors.isopen ?  "open" :  "closed">>) lead north."
  out = (self.north)
  north =
  {
    if (elevDoors.isopen) return(self.curfloor);
    else
```

```
    {
      "The doors are closed.";
      return(nil);
    }
  }
  curfloor = floor1                                  /* start off on lower floor */
  moveDaemon =
  {
    self.counter++;
    switch(self.counter)
    {
    case 1:
    case 2:
    case 3:
      "\bThe elevator continues to travel slowly.";
      break;

    case 4:
      "\bThe elevator stops, you hear a bell make a
      \"ding\" sound, and the doors slide open.  ";
      elevDoors.isopen := true;
      self.isActive := nil;
      unnotify(elevRoom, &moveDaemon);
      break;
    }
  }
;
```

The "\b" sequences before the messages in the `moveDaemon` method are there because the routine runs as a daemon. This means it's hard to predict exactly what will have been displayed just prior to the method's execution, so it's safest to put in a blank line to make sure that the daemon's messages are set apart from the previous message. You should generally put in a blank line before the first message displayed by any daemon.

Note that you'll have to do a small amount of extra work to finish this example. First, you'll have to define the rooms `floor1` and `floor2`. These are extremely simple if the only way to reach them is through the elevator, because you don't have to worry about whether the elevator is present or not. If you want to be more sophisticated, and make the floors reachable without taking the elevator (for example, with a stairway), you'll have to do two things: first, you'll have to put some doors on each floor, and make sure they're closed while the elevator isn't on that floor; and second, you'll probably want to make some way to call the elevator to the current floor.

Here's a general class for making the doors on each floor. We'll make the door's `isopen` property simply check to see if the inner doors are open, and if so, whether the elevator is on the current floor (which is the door's location). We'll do the same thing for the `verDoOpen` and `verDoClose` methods

that we did with the `elevDoors` object.

```
class outerElevDoor:  fixeditem
  sdesc = "elevator door"
  ldesc = "They're <<self.isopen ?  "open" :  "closed">>.  "
  isopen = (elevDoors.isopen and elevRoom.curfloor = self.location)
  noun = 'door' 'doors'
  adjective = 'elevator'
  verDoOpen(actor) =
  {
    "The doors are automatic; you can't open them yourself.  ";
  }
  verDoClose(actor) =
  {
    "The doors are automatic; you can't close them yourself.  ";
  }
;
```

Now you just need to add an object of this class for each floor; the only thing you need to set in each object is its `location` property. Then, on each floor, make the `south` property sensitive to the state of the doors, in the same way as the `north` property in the `elevRoom` object is sensitive to the state of the inner doors.

The other addition you'll want to make is to add a call button on each floor. This is probably accomplished most easily by simply using the existing daemon that runs the elevator, and adding new buttons on the floors, outside the elevator, that activate the elevator. Here's a basic class that can be used to make these buttons.

```
class outerButton:  buttonitem
  sdesc = "elevator call button"
  adjective = 'elevator' 'call'
  doPush(actor) =
  {
    /* ignore if elevator is already here */
    if (elevRoom.curfloor = self.location and elevDoors.isopen)
      "Your incredible powers of observation suddenly inform
      you that the elevator is already here.  ";
    else
    {
      "\"Click.\"";
      if (not elevRoom.isActive) notify(elevRoom, &moveDaemon, 0);
      elevRoom.isActive := true;
      elevDoors.isopen := nil;
      elevRoom.counter := 0;
      elevRoom.curfloor := self.location;
    }
```

```
    }
```

You'll notice if you try to run this example that there's a minor problem: the daemon that moves the elevator around displays messages as though the player were on the elevator. By using the same daemon to move the elevator around with the call buttons, we'll have to change the messages so that they're sensitive to whether the player is on the elevator or not. For motion messages, the player should only get the messages if the player is on the elevator. Messages concerning the doors opening, on the other hand, should be seen if the player is either on the elevator or at the location at which the elevator is arriving. The new `moveDaemon` method (for the `elevRoom` object) is shown below (the rest of the `elevRoom` object stays the same).

```
moveDaemon =
{
  self.counter++;
  switch(self.counter)
  {
  case 1:
  case 2:
  case 3:
    if (Me.location = self)
      "\bThe elevator continues to travel slowly.";
    break;

  case 4:
    if (Me.location = self)
      "\bThe elevator stops, you hear a bell make a
      \"ding\" sound, and the doors slide open.  ";
    else if (Me.location = self.curfloor)
      "\bYou hear a bell make a \"ding\" sound,
      and the elevator doors slide open.  ";
    elevDoors.isopen := true;
    self.isActive := nil;
    unnotify(elevRoom, &moveDaemon);
    break;
  }
}
```

There are many more enhancements that you could make to the elevator control daemon to make the elevator more realistic. First, it would be fairly easy to add more floors, simply by adding more buttons. Second, it would be nice to provide some sort of indication of where the elevator is currently. In addition, real elevators use a somewhat different control algorithm. A real elevator usually has a current direction; it travels in that direction, stopping at each floor that's been selected with a call button at the floor or with the matching button in the elevator. When it gets to the top or bottom floor, all of the selected buttons inside the elevator are cleared (i.e., de-selected). The elevator then reverses direction if necessary, servicing called floors. This type of algorithm would be a little more work to implement, and is left as an exercise to the reader.

Chairs and other Nested Rooms

A "nested room" is a room inside another room. In most cases, nested rooms are also objects in their own right. The principal feature of a nested room is that it isn't enclosed. Typical examples of nested rooms are chairs and beds: these are objects within a room, but they're also "rooms" in the sense that the player can be located in them.

Implementing a nested room in TADS is fairly simple, because `adv.t` defines a class called `nestedroom` that does most of the work. Furthermore, the classes `chairitem` and `beditem` let you implement the most common nested room objects with very little work.

One feature of nested rooms that is worth mentioning is that the player can not necessarily reach all of the objects in the enclosing room from a nested room. By default, *none* of the objects in the enclosing room are reachable from a nested room. However, you can easily make any objects reachable from the nested room by setting its `reachable` property to a list containing all the reachable objects in the enclosing room. If you want to set it to everything in the enclosing room, simply set it as follows:

```
reachable = (self.location.contents)
```

This says that everything contained in the enclosing room is reachable.

Nested Room Vehicles

Sometimes, you'll want to make a nested room that's also a vehicle. For example, you might want to implement a rubber raft that the player can carry around, but also inflate and use as a vehicle. Fortunately, there's an `adv.t` class called `vehicle` that serves this function. The `vehicle` class is a carryable object that can be boarded as a vehicle.

Let's implement an inflatable rubber raft. The first thing we need to do is build a standard `vehicle` object, but we'll customize it so that it can only be boarded when it's inflated. We'll furthermore make it so that it can only be inflated when it's not being carried, and we'll make it impossible to take when it's inflated.

```
raft:  vehicle
  location = startroom
  sdesc = "inflatable rubber raft"
  noun = 'raft'
  adjective = 'inflatable' 'rubber'
  isinflated = nil
  ldesc = "It's an inflatable rubber raft.  Currently,
  it's <<self.isinflated ?  "inflated" :  "not inflated">>.  "
  verDoTake(actor) =
  {
```

```
    if (self.isinflated) "You'll have to deflate it first.  ";
    else pass doTake;
  }
  verDoInflateWith(actor, iobj) =
  {
    if (self.isinflated) "It's already inflated!";
  }
  doInflateWith(actor, iobj) =
  {
    if (self.isIn(actor)) "You'll have to drop it first.  ";
    else
    {
      "With some work, you manage to inflate the raft.  ";
      self.isinflated := true;
    }
  }
  verDoDeflate(actor) =
  {
    if (not self.isinflated) "It's as deflated as it can get.";
  }
  doDeflate(actor) =
  {
    "You let the air out, and the raft collapses to a
    compact pile of rubber.";
    self.isinflated := nil;
  }
  doBoard(actor) =
  {
    if (self.isinflated) pass doBoard;
    else "You'll have to inflate it first.";
  }
;
```

Note that we'll have to define a couple of verbs, and it would be convenient to add an object for the pump as well.

```
inflateVerb:  deepverb
  sdesc = "inflate"
  verb = 'inflate' 'blow up'
  ioAction(withPrep) = 'InflateWith'
  prepDefault = withPrep
;


deflateVerb:  deepverb
```

```
  sdesc = "deflate"
  verb = 'deflate'
  doAction = 'Deflate'
;

pump:  item
  sdesc = "pump"
  location = startroom
  noun = 'pump'
  verIoInflateWith(actor) = {}
  ioInflateWith(actor, dobj) =
  {
    dobj.doInflateWith(actor, self);
  }
;
```

Once all of this is done, making the raft move around is very similar to making a fully-enclosed vehicle move around. The only real difference is that the raft will always have a location. As with most vehicles of this type, you'll want to make it impossible for the player to get out of the raft while it's in motion. To do this, you'll need to override the `doUnboard` and `out` methods. For this example, let's assume that the river (where the raft will travel) will be composed of a series of rooms, each of which is of class `riverRoom`. So, whenever the raft is in a `riverRoom`, we'll prevent the player from getting out. To do this, we'll add the methods below to the `raft` object. Note that these methods will inherit the corresponding `vehicle` methods when the raft is not floating down the river.

```
doUnboard(actor) =
{
  if (isclass(self.location, riverRoom))
    "Please keep your hands and arms inside the raft
    at all times while the raft is in motion.";
  else pass doUnboard;
}
out =
{
  if (isclass(self.location, riverRoom))
    "You can't get out until you've landed the raft.";
    return(nil);
  else pass out;
}
```

The only thing left is to move the raft around. For this example, we'll set things up as follows: each (non-river) room which borders on a river will have a property, `toRiver`, set to point to the bordering river room. Each river room which has a landing will have a property, `toLand`, set to point to the bordering non-river room. Each river room will also have a property, `downRiver`, set to the next river room downstream from the current room. We'll introduce two new verbs: "launch"

and "land." When the player is in the raft, "launch raft" will set the raft in motion down the river (if it's on land), and "land raft" will land (if there's a landing nearby). The new verbs are below.

```
launchVerb:  deepverb
  sdesc = "launch"
  verb = 'launch'
  doAction = 'Lanuch'
;


landVerb:  deepverb
  sdesc = "land"
  verb = 'land'
  doAction = 'Land'
;
```

Now we'll add some more methods to the `raft` object: the verb handling methods for the new verbs, and a daemon that causes the raft to drift downstream while it's in the river.

```
verDoLaunch(actor) = {}
doLaunch(actor) =
{
  if (isclass(self.location, riverRoom))
    "You're already afloat, if you didn't notice.";
  else if (self.location.toRiver = nil)
    "There doesn't appear to be a suitable waterway here.";
  else if (Me.location != self)
    "You'll have to get in the raft first.";
  else
  {
    "The raft drifts gently out into the river.";
    notify(self, &moveDaemon, 0);
    self.counter := 0;
    self.moveInto(self.location.toRiver);
  }
}
verDoLand(actor) = {}
doLand(actor) =
{
  if (not isclass(self.location, riverRoom))
    "You're already fully landed.";
  else if (self.location.toLand = nil)
    "There's no suitable landing here.";
  else
  {
    "You steer the raft up onto the shore.";
```

```
    unnotify(self, &moveDaemon);
    self.moveInto(self.location.toLand);
  }
}
moveDaemon =
{
  "\bThe raft continues to float downstream.";
  self.counter++;
  if (self.counter > 1)
  {
    self.counter := 0;
    if (self.location.downRiver = nil)
    {
      "The raft comes to the end of the river, and lands.";
      self.moveInfo(self.location.toLand);
      unnotify(self, &moveDaemon);
    }
    else
    {
      self.moveInto(self.location.downRiver);
      self.location.riverDesc;
    }
  }
}
```

Note that we'll expect each river room to have a property, `riverDesc`, which displays a message when the raft drifts into that room. The `moveDaemon` method will keep the raft in each river room for two turns, then move the raft to the next river room, calling `riverDesc` to note the entry. When the raft comes to the end of the river, the method will automatically land the raft; this means that the last river room must have a non-nil `toLand` property. You could alternatively put in a waterfall or other special effect when reaching the end of the river.

To build a river, all you have to do is define a series of rooms of class `riverRoom`, and point the `downRiver` property in each to the next room downriver. Landings are built by setting the `toRiver` and `toLand` properties of the landing room and corresponding river room, respectively, to point to each other.

Hiding Objects

Hiding objects is fairly simple, thanks to a set of classes defined in `adv.t` that make certain kinds of hiding automatic.

The `adv.t` classes make it possible to hide objects under or behind other objects, and to set up an

object so that its contents are found only when the object is searched. The basic hider classes are `underHider`, `behindHider`, and `searchHider`. These classes are for the objects doing the hiding; the hidden objects are hidden inside these objects. For the objects being hidden, instead of setting their `location` properties, set the properties `underLoc`, `behindLoc`, or `searchLoc` to point to the respective classes. All hidden objects must be of class `hiddenItem`.

For example, to hide a key under a bed, make the bed an `underHider` object, and set the `underLoc` property of the key to point to the bed.

```
bed:  beditem, underHider
  noun = 'bed'
  location = startroom
  sdesc = "bed"
;


key:  item, hiddenItem
  noun = 'key'
  sdesc = "key"
  underLoc = bed
;
```

The `behindHider` and `searchHider` objects work the same way, but you should use the `behindLoc` and `searchLoc` properties, respectively, instead of `underLoc`.

Note that the `initSearch` function, defined in `adv.t`, must be called during initialization (usually, by your `preinit` function) to set up the hidden objects. This routine sets up special contents lists for the various `hider` objects. This routine only considers hidden objects when they're of class `hiddenItem`, which is why you must use this class for all of your hidden objects.

Note that the ease with which you can hide objects shouldn't be taken as a license to hide objects with wild abandon. From a game design viewpoint, hidden objects are almost always poor puzzles. If you do hide an object, make sure that you provide some sort of clue that something is hidden there. Otherwise, the player must tediously look behind and under every object in the game. As with most puzzles, it's easy to make a really hard game—it's harder to make a fun game.

Collections of Objects

Each item that a player can manipulate in a TADS game generally corresponds to an object defined in the game program. To allow the player to refer to each object individually, your game program must define a unique set of vocabulary words for each object. For example, if you have two books in your game, the books must have at least one distinct adjective each, so that the player can specify in a command which book he wants to manipulate.

This style of game construction, in which each object must have a unique set of vocabulary, imposes certain limitations when you design your game. Sometimes you will find that you want to have a large collection of similar objects that are not distinguished from one another. Fortunately, there are a number of ways to implement object collections; the particular mechanism you choose will vary, and depends on the type of situation you want to create. This section provides examples of several different effects you can create with collections of objects.

Object Collections as Decorations

Suppose you are creating a book store. You will want to stock the store with a large number of books—but you won't want to create a huge number of individual book objects, not only because it would take too much time and effort, but also because most of the books won't be relevant to the game. There will usually be one or two important books, and many unimportant ones.

In this type of situation, one way to work within the limitations of TADS is to create a single object representing the collection of unimportant books. Since most of the books are not important, they can be grouped together into this single object, which is set up so that it tells the player about its lack of importance whenever the player tries to manipulate it. Furthermore, it can serve as a pointer to any important similar objects.

As an example, we'll implement a collection of books in a book store. A single object will represent the collection of unimportant books. When the player looks at the collection of books, though, it will point out the presence of a couple of important books.

```
manybooks:  fixeditem
  sdesc = "books"
  adesc = "a number of books"
  noun = 'book' 'books'
  location = bookstore
  ldesc =
  {
    "The bookstore has a huge stock of books, ranging from
    scientific titles to the latest collection of
    \"Wytcome and Wyse\" cartoons.  ";
    if (not self.isseen)
    {
      "One title in particular catches your attention:
      \"Liquid Nitrogen:  Our Frigid Friend\".  ";
      ln2book.moveInto(bookstore);
      self.isseen := true;
    }
  }
  verDoRead(actor) =
  {
    "Although you'd really like to sit down and start reading,
```

```
    you know from painful experience that the neo-fascist staff
    have been cracking down on browsing lately.  ";
  }
  verDoTake(actor) =
  {
    "Unfortunately, you know you could never afford all of these
    books.  ";
  }
;
```

Note that we set the `adesc` property to a non-default value. This is because the default `adesc`, which would display "a" followed by the object's `sdesc`, would result in the strange message "a books" in this case.

To help the player avoid wasting a lot of time trying to manipulate the books, we put in some special messages that let the player know that the books can't be manipulated. When you create a decoration object, it's always a good idea to think of all of the things that the player might want to do with it, and provide messages that make it clear that the object isn't important to the game. In this case, we've made it clear that the collection of books can't be read or taken.

The most important definition, though, is the `ldesc` property. This property not only provides a general description of the collection of books, but also points out the single important book to the player when the collection of books is examined for the first time. (The `isseen` property is used to determine if the `ldesc` has been displayed before. The first time the `ldesc` property is executed, `isseen` is set to `true`.) In addition to displaying a message for the player about the special book, it moves the special book into the bookstore.

Selecting from an Object Collection

One way you may wish to extend this basic idea is to provide a collection of books from which the player can take a single book, seemingly at random. The first time a player takes a book from the collection, he gets one book; the next time, he gets another book; and so forth. This is easy to implement by overriding the `doTake` method of the collection of books: rather than moving the collection itself into the player's inventory, the method will choose a book from a list of individual book objects, and move that book into the player's inventory.

```
manybooks2:  fixeditem
  sdesc = "books"
  adesc = "a number of books"
  noun = 'book' 'books'
  location = bookstore
  ldesc =
  {
    "The bookstore has a huge stock of books, ranging from
    scientific titles to the latest collection of
```

```
    \"Wytcome and Wyse\" cartoons.  You
    may want to just try picking up a book.  ";
  }
  verDoTake(actor) =
  {
    if (length(self.booklist) = 0)
      "You don't see anything else you're interested in.  ";
  }
  doTake(actor) =
  {
    local selection;

    selection = self.booklist[1];
    self.booklist := cdr(self.booklist);
    selection.moveInto(actor);

    "You look through the books, and choose << selection.thedesc >>.  ";
  }
  booklist = [ln2book chembook cartoonbook]
;
```

The property `booklist` contains a list of books that we have pre-defined. Each time the player takes a book, the `doTake` method will take the first element out of the `booklist` property, and act as though the player had picked up that object. This way, the player can just type "take book," and the game will seem to select a book for the player. The `verDoTake` property makes sure there's something left in the list, and displays an appropriate message if not. Note that we've written the `ldesc` message so that it's clear to the player that he should attempt to take a book.

Selecting from Indistinguishable Objects

Another variation on this theme involves a collection of indistinguishable objects. For example, suppose we want to implement a matchbook which contains a number of matches. The player should be able to take a match out of the matchbook in order to light it.

Since TADS requires every object to have unique vocabulary, though, we'll have to impose some limits on what the player can do. In particular, we'll only allow the player to have a single match at any given time—all of the other matches must be in the matchbook.

In terms of implementation, this means that we'll only need a single object to represent a match. This object will have two possible states: existent or non-existent. We'll use the `location` property to indicate the state; if the `location` is `nil`, the object is non-existent, otherwise it's in the game. When the match is non-existent, the player can take a match out of the matchbook—which brings the match into existence by moving it into the player's inventory. As long as the match is in the game, the player cannot take another match out of the matchbook. Once the match is burned away, though, it will be removed from the game, and the player can take another match.

The matchbook will have a count of available matches. When the player attempts to take a match, we'll first check to see if there are any matches left; if so, we'll check to make sure that the match object doesn't already exist in the game.

```
matchbook:  item
  location = bookstore
  noun = 'matchbook'
  sdesc = "matchbook"
  matchcount = 4                        /* number of matches left in matchbook */
  ldesc =
  {
    if (self.matchcount > 0)
      "The matchbook contains << self.matchcount >> match<<
      self.matchcount = 1 ?  "." :  "es.  " >>";
    else
      "The matchbook is empty.";
  }
;


match:  item
  noun = 'match'
  adjective = 'single'
  sdesc = "single match"
  ldesc =
  {
    if (self.isBurning)
      "It's currently lit.";
    else
      "It's an ordinary safety match.";
  }
  verDoLight(actor) =
  {
    if (self.isBurning) "It's already lit!";
  }
  burnFuse =
  {
    "\bThe match burns down.  You drop it, and it disappears
    in a cloud of ash.";
    self.isBurning := nil;
    self.moveInto(nil);                         /* match is now non-existent */
  }
  doLight(actor) =
  {
    "The match starts burning.";
```

```
    self.isBurning := true;
    notify(self, &burnFuse, 2);
  }
;


fakematch:  fixeditem
  noun = 'match'
  adjective = 'bound'
  sdesc = "bound match"
  location = matchbook
  verDoTake(actor) =
  {
    if (matchbook.matchcount = 0)
      "The matchbook is empty.";
    else if (match.location <> nil)
      "You'll have to use the one you already took first.";
  }
  verDoLight(actor) =
  {
    "You'll have to remove it from the matchbook first.";
  }
  doTake(actor) =
  {
    "You tear a match out of the matchbook.";
    match.moveInto(actor);                    /* move a match into inventory */
    matchbook.matchcount--;                             /* one less match */
  }
;
```

The `fakematch` object lets the player refer to the match in a command, even when the `match` object doesn't exist in the game (i.e., its `location` is `nil`). The `fakematch` has `verDoTake` and `doTake` methods that let the player take a match out of the matchbook.

Note one minor flaw in this implementation: the player might not have the match, but still can't take another until the first is destroyed. This could be confusing. For example, the player may take a match, then later drop it because his hands are full, then move to a different room. As far as the player is concerned, no match is present—he left the match in another room, and may not even remember where. However, he won't be able to take another match, because the first match is still in the game. There's no easy way to fix this. You could arbitrarily move the match into the player's inventory, even though it is in some other room; this might be less confusing for some players, but it may be more confusing for others who are aware that there is a match in another room—this "other" match would be gone after the single match object is moved into the player's inventory.

Money

Money does not fit very naturally into a TADS game for a number of reasons. If you do want to use money in your game, it can generally be implemented using techniques similar to those we used for the matches in the matchbook. The only changes for money are that some new verbs will probably be added, and you will probably want to be able to add more money to the player's holdings.

The necessary new verbs will depend on your game. For example, you may want "pay" and "buy" verbs for purchasing items. The main reason you'll want to use these new verbs is that there's no easy way for the player to refer to an amount of money. So, you'll have to arrange a protocol for purchasing objects. For example, make a location that serves as a store, and make a shopkeeper actor. To purchase an object, the player tells the shopkeeper to give him an object. The shopkeeper responds by telling the player the price of the object. The player responds with "pay shopkeeper"; this deducts the price of the object, and gives the player the object.

If your game contains a non-obvious protocol like this, it is very important that you document it for the player. You can provide an explanation in the instructions for your game, but since most people will start playing your game immediately without reading any accompanying instructions, it would be far better to put the explanation into the game itself. There are a number of ways you could do this; you could, for example, simply display an explanation of the necessary commands when the player first enters the shop. Alternatively, you could provide instructions as error messages; when the player attempts to take an object in the shop, you could display a message: "Please direct your inquiries to the shopkeeper; for example, type 'shopkeeper, give me the axe.'"

An even better approach would be to avoid situations like this altogether. If you can't make it fit easily and naturally into your game without resorting to special sequences of commands, you should probably find another way to accomplish the same thing. Remember, absolute realism is not important in a text adventure game—the only important thing is to make the game fun to play.

Non-Player Characters

One of the most effective techniques for giving a game depth and making it involving is to include non-player characters. Early adventure games often seemed empty and static; the player just wandered alone through a vast maze of caves. More modern games use non-player characters (which we'll refer to as "actors" from now on), who move through the game, interacting with the player and doing things on their own. When done well, actors add a whole new dimension to a game, and make the setting seem alive and real.

Actors are one of the most complicated parts of a TADS game, because they operate on more levels than other objects. First, there's player interaction: an actor normally can interact with the player, by talking (the player can tell an actor to do things, and can ask an actor questions), and by direct action (attacking, for example, or exchanging objects). There are a great many ways a player and

an actor can interact, and TADS lets you implement as much or as little interactive capability into an actor as you wish. Second, actors often do things on their own. Sometimes, this means that the actor carries out a pre-scripted set of actions. Other times, the actor just reacts to the player; for example, an actor might spend most of the game following the player around, making amusing comments from time to time. Most actors, though, are combinations of these, reacting to the player most of the time, and carrying out pre-scripted actions in response to certain events. Third, some actors have a "memory," and have knowledge of various parts of the game. Certain things an actor does may depend on what a player has done previously.

Fortunately, all of these basic elements of an actor are implemented pretty easily. You can write the code for a simple actor without too much difficulty. Even better, once you have a simple actor working, fleshing out the actor is just a matter of adding to the basic framework of the actor.

To implement an actor, you need to write two main pieces of code that aren't associated with most other types of objects. First is the `actorAction` method. This method handles most of the actor's interaction with the player; it essentially defines the limits of the actor's ability to interact. Second is the actor's "daemon." You'll recall from previous chapters that a daemon is a function or method that's called after every turn (that is, the run-time system calls a daemon each time it finishes processing a player command). The actor daemon is what allows the actor to go about its business; since it is called after each turn, the actor can perform an action of its own on each turn.

If you've played *Ditch Day Drifter,* you'll recall a couple of actors in that game of differing degrees of interaction and activity. One fairly limited actor is the guard in front of the steam tunnels: the guard does very little apart from blocking your way into the tunnel. The guard's interaction consists mostly of accepting objects; you can attempt to bribe the guard with money, for example, or give him something to drink. The guard's daemon also does very little; it just displays a randomly-chosen message (from a set of several pre-determined messages) on each turn. A more versatile actor is Lloyd, the insurance robot. Lloyd can accept objects (you can buy the insurance policy), and you can also talk to him to a limited extent (you can ask about the policy). Like the guard, Lloyd displays a randomly-chosen message on most turns. In addition, Lloyd follows the player from room to room. Lloyd even reacts to certain events and places, and has a memory of sorts: once Lloyd has paid out an insurance claim, he will remember not to pay out the claim again.

Let's look at an example of implementing an actor. We'll start with a simple "puzzle actor," similar to the guard in *Ditch.* This is a very common element of adventure games: a person that is blocking the way to some goal. In this case, we'll implement a receptionist who's blocking the way to an office you want to enter. As long as the receptionist is awake, you won't be allowed to enter the office.

```
receptionist:  Actor
  noun = 'receptionist'
  sdesc = "receptionist"
  isawake = true
  ldesc =
  {
    "The receptionist is a very, shall we say, sturdy-looking
    woman, with a tall beehive hair-do and thick glasses.  She
    reminds you of your third-grade teacher, who was overly
```

```
    concerned with discipline.";
    if (self.isawake)
      "She eyes you impatiently as she goes through some papers.";
    else
      "She's slumped over the desk, fast asleep.";
  }
  location = lobby
  actorDesc =
  {
    if (self.isawake)
      "A receptionist is sitting at the desk next to the
      door, watching you suspiciously.";
    else
      "A receptionist is slumped over the desk next to
      the door.";
  }
  actorDaemon =
  {
    if (self.location <> Me.location or not self.isawake) return;
    switch(rand(5))
    {
    case 1:
      "The receptionist sharpens some pencils.";
      break;
    case 2:
      "The receptionist goes through some personal mail, holding
      each letter up to the light and attempting to read the
      contents.";
      break;
    case 3:
      "The receptionist looks through the personnel files.";
      break;
    case 4:
      "The receptionist answers the phone and immediately
      puts the caller on hold, cackling to herself fiendishly.";
      break;
    case 5:
      "The receptionist shuffles some papers.";
      break;
    }
  }
;
```

```
lobby:  room
  enterRoom(actor) =
  {
    if (not self.isseen)
      notify(receptionist, &actorDaemon, 0);
    pass enterRoom;
  }
  sdesc = "Lobby"
  ldesc = "You're in a large lobby, decorated with expensive-looking
  abstract pastel oil paintings and lots of gleaming chrome.
  A large receptionist's desk sits next to a door leading
  into an office to the east.  The exit is to the west."
  west = outsideBuilding
  out = outsideBuilding
  east =
  {
    if (receptionist.isawake)
    {
      "You start to stroll past the receptionist nonchalantly,
      trying to ignore her as though you actually belong here,
      but she's wise to that trick:  she leaps up, and with
      surprising force throws you back from the doorway.
      Satisfied that you're not going through, she sits back
      down and returns to her work.";
      return(nil);
    }
    else
      return(office);
  }
;
```

Notice that the implementation of this puzzle looks very much like that of a door, or any other obstacle, except that the obstacle itself is somewhat more complicated. The actual work of preventing the player from moving past the receptionist, though, is exactly the same as for any other obstacle: in the direction method, we check to see if the receptionist is still awake, and if so, we refuse to let the player past by returning `nil` (after displaying an appropriate message, of course).

The main features that are special for an actor are the `actorDesc` and `actorDaemon` methods. The `actorDesc` is a special property that you should define for any actor; this method is called by the general `room` code that displays the long description of the room. After the description of the room and its contents, the `room` long description routine will call the `actorDesc` method of each actor in the room (except the player actor, `Me`). The `actorDesc` method should simply display an appopriate message to the effect that the actor is present; it's similar to the `ldesc` method for the actor, but it usually contains less detail, since it's more to inform the player that the actor is present than to provide a detailed desription of the actor. It should mention that the actor is present and what the

actor is doing.

The `actorDaemon` is nothing special—you can call this method whatever you want. Most actors will have something like the `actorDaemon`, though. This is the daemon that makes the actor seem alive by doing something after every turn. For our simple receptionist actor, which doesn't really do anything of its own accord, this method simply displays a random message each turn. Note that the method returns immediately if the player (the `Me` object) is not present in the same room as the receptionist; it clearly wouldn't make any sense to display a message about what the receptionist is doing when the receptionist is not present.

Note that `actorDaemon` is *not* started automatically for an actor. Instead, you must explicitly activate it. In this case, we've chosen to activate the `actorDaemon` routine by calling `notify()` the first time the player enters the lobby. The `enterRoom` method of the `lobby` object checks to see if the room has been entered before (by checking to see if the `isseen` property of the current room has been set—this is always automatically set by the `room` code after the room has been seen for the first time); if the room has never been entered, we call `notify()` to start `actorDaemon` running. Note that `enterRoom` finishes by using `pass` to run the `enterRoom` method that would normally be inherited from the `room` class.

Following the Player

There are several main types of wandering actors. The first type follows the player around, like Lloyd the insurance robot in *Ditch Day Drifter*. This first type is fairly simple, because all you have to do is set up a daemon that checks to see if the actor and the player are in the same room, and if not, move the actor to the player's location.

The big change between a stationary actor, such as the receptionist in the earlier example, and a wandering actor is in the actor's daemon. Instead of just displaying a random message, as did the receptionist, the actor daemon will actually move the actor around.

Here's a basic `actorDaemon` method that will make the actor follow the player.

```
actorDaemon =
{
  if (self.location = Me.location)
  {
    switch(rand(5))
    {
    case 1:
      "Lloyd hums one of his favorite insurance songs.";
      break;
    /* etc.  with other random messages...  */
    }
  }
  else
  {
```

```
      self.moveInto(Me.location);
      "Lloyd rolls into the room, and checks to make sure
      the area is safe.";
    }
  }
```

Note that this routine must be set up to run every turn with the `notify()` function, as explained in the receptionist example.

This new version of the method still displays a random message if the actor is in the same room with the player, but now it has some new code that is executed when the player is somewhere else. The new code moves the actor to the player's location, and displays a message informing the player that the actor has entered the room.

In a real game, you'll probably further extend this type of daemon to make the actor do special things in certain locations. For example, in *Ditch Day Drifter*, there's some extra code that displays special messages when Lloyd enters certain improbable locations, such as squeezing through the north-south crawl or climbing the rope.

Moving on a "Track"

The second type of wandering actor moves around on a fixed path, or "track." This type of actor is not much harder to implement than an actor which follows the player; rather than using the player's location in the actor motion daemon, you make a list of locations that the actor visits in sequence, and the actor motion daemon takes locations out of this list.

One new factor you'll have to keep in mind in implementing this type of motion daemon is that the actor could be entering the actor's location, leaving the actor's location, or neither. The motion message will have to check for each of these situations.

For an example, we'll show how to implement a robot similar to the vacuuming robot in *Deep Space Drifter* that moves around between several rooms in a fixed pattern.

```
vacRobot:  Actor
  sdesc = "domestic robot"
  noun = 'robot'
  adjective = 'domestic'
  location = stationMain
  tracklist = ['southeast' bedroomEast 'west' bathroom 'west'
      bedroomWest 'northeast' stationMain 'north'
      stationKitchen 'south' stationMain]
  trackpos = 1
  moveCounter = 0
  actorDaemon =
  {
    if (not self.isActive) return;                /* do nothing if turned off */
```

```
    self.moveCounter++;
    if (self.moveCounter = 3)           /* move after 3 turns in one location */
    {
      self.moveCounter := 0;
      if (self.location = Me.location)
        "The robot moves off to the <<self.tracklist[self.trackpos]>>.";
      self.moveInto(self.tracklist[self.trackpos + 1]);
      if (self.location = Me.location)
        "A domestic robot rolls into the room and starts noisily
        moving around the room vacuuming and dusting.";

      /* move to next position on track; loop back to start if necessary */
      self.trackpos += 2;
      if (self.trackpos > length(self.tracklist)) self.trackpos := 1;
    }
    else
    {
      /* we're not moving this turn, so display activity message */
      if (self.location = Me.location)
        "The domestic robot continues darting around the room
        vacuuming.";
    }
  }
;
```

The list of rooms that the robot visits is specified in the `tracklist` property. This is a strange-looking list, because it contains both objects and (single-quoted) strings. The way this works is that the list elements are always inspected in pairs: the first item in a pair is a string that will be displayed to indicate the direction that the robot will leave the current room; the second item in the pair is the room to be visited next after the current room. So, the first pair contains the string `'southeast'` and the object `bedroomEast`: the robot will leave to the southeast, and move to the east bedroom.

The `actorDaemon` still does all the work, but it's a little more complicated than in the previous examples. First, it checks to see if the robot is active; if not, it doesn't go any further. Next, it increments a counter that tracks how long the robot has been in the current room; when it gets to three turns, it's time to move on, otherwise the robot stays where it is.

If the robot is moving, the daemon first resets the move counter to zero. Next, it checks to see if the player is in the current location; if so, the daemon displays a message telling the player that the robot is leaving, and indicates which direction (the direction is determined by looking at the current element of the `tracklist` list, as described above). Next, the robot is actually moved to the new location (again, determined by looking in the `tracklist` list). Next, if the player is in the *new* location, the daemon displays a message saying that the robot has entered the room. Finally, we increment the track position counter property `trackpos` (by 2, since the list entries are in pairs).

If the position counter has moved beyond the end of the list, we reset it to 1.

If the robot is not moving during this turn, the daemon will check to see if the actor is in the same location as the player, and display a message if so. Note that the single fixed message in the example could easily be replaced by a randomly-chosen message, as we did with the earlier examples.

Random Wandering

The third type of wandering actor moves through the game at random. This type of actor wandering is somewhat more complex than the earlier types.

The main trick in implementing a randomly-wandering actor is that we must avoid evaluating direction properties when they don't contain a fixed object. This is because we could accidentally display spurious messages, and possibly trigger side effects (such as killing the player or making other changes to the game state) if we were to evaluate a direction property that was actually a method. For example, think back to the earlier example with the stairway that collapses when used—you probably wouldn't want to display the collapse message and destroy the stairs when one of your wandering actors encountered the room.

The best way to avoid such unintended side effects is to avoid using any direction property that isn't a simple object. TADS provides a special built-in function called `proptype` that lets you determine the type of a property without actually evaluating it. The `proptype` function will tell you whether a property is a simple object or a method.

Other than the mechanism for choosing a new room, randomly-wandering actors are essentially the same as actors on a track. You generate messages in the same manner, checking to see if the actor is leaving or entering a room occupied by the player.

```
actorDaemon =
{
  local dir, dirnum;
  local tries;
  local newloc;
  for (tries := 1 ; tries < 50 ; tries++)
  {
    dirnum := rand(6);
    dir := [&north &south &east &west &up &down][dirnum];
    if (proptype(self.location, dir) = 2 /* object */)
    {
      newloc := self.location.(dir);
      if (not isclass(newloc, room)) continue;
      if (self.location = Me.location)
        "The robot leaves to the <<
        ['north' 'south' 'east' 'west' 'up' 'down'][dirnum]>>.";
      self.moveInto(newloc);
      if (self.location = Me.location)
```

```
            "A domestic robot enters the room and starts vacuuming
            noisily.";
        }
    }
}
```

The line that sets the `dir` local variable probably needs a little explanation, because it's a tricky bit of TADS coding. The first thing is a list of property addresses—these aren't properties of any object, but rather just references to properties that you can later use to get an actual property of an object. We take this list, which consists of six elements, then choose a single element at random by indexing into the list with a randomly-chosen number from 1 to 6. This leaves the `dir` local variable with a pointer to a property that we can use later. Note that you could include northeast, southeast, northwest, and southwest in the list if you want; we left them out to make the example a little bit easier to read.

Next, we use the `proptype()` built-in function to determine if the selected property (contained in `dir`) of the actor's current location contains an object. If it does not, we continue looping.

If the selected property of the current location does in fact contain an object, we first get that location into the local variable `newloc`. Then, we make one more check on the new location: we ensure that `newloc` actually contains an object of class `room`. This is because it is possible for a room direction connection property to contain an object of class `obstacle`, such as a door (see the section on implementing doors earlier in this chapter). If it's a door or other obstacle, we want to treat it the same as though the direction property didn't contain an object at all—so we'll just continue with the loop, ignoring this choice of a direction.

Once we have an object of class `room` in `newloc`, we proceed to move the actor the same way we moved the actor on a track. Note that we use the same indexing trick to display the direction that the actor is exiting.

In case you're wondering why the example uses a "tries" counter and stops after 50 times through the loop: this is simply a bit of bullet-proofing to keep the daemon from going into an infinite loop. It's quite possible, especially while you're developing your game, that the robot may encounter some rooms from which there is no escape. Remember that the robot can only use exits which are explicitly set to objects; if you create a room with nothing but exits that are methods, the robot will be unable to find a suitable exit. If we didn't include the loop counter and give up after a while, the loop would run forever in such rooms, leaving the game stuck.

In a real game, you may want to add some special-case code that allows a randomly-wandering actor to use certain exits that would normally be off-limits because they're methods. One way of accomplishing this is to add a special check to the daemon to test if the actor is in such a special location, and if so, to make it go to a particular destination. Another way is to put additional properties in some rooms (such as `robotEast`) that specify directions accessible only to the robot; in the daemon, you'd check for the presence of these special robot directions first, falling back on the normal direction list only if the special directions are not present.

One modification you may wish to make to this type of totally random wandering is to restrict the actor to a particular set of rooms. There are a number of ways to accomplish this. The simplest

is to create a class of rooms that the actor is allowed to occupy, and always check upon choosing a new room that the room is a member of the class.

If you want to change the allowed rooms dynamically as the game runs, use a condition rather than a class. Instead of checking to see if the room is a member of a class, check to see if the condition is true. For example, if you want to create an actor that only enters dark rooms, you would check upon choosing a room to make sure the room is not lit.

More Complex Actor Scripts

For some games, you may wish your actors to exhibit even more complex autonomous behavior than we've shown so far. One type of common extension is to make the actor perform some action when in a certain room or when a certain set of conditions is met. For example, you may wish to implement a thief that steals any treasure left lying around in a maze room. To implement this, we'd first need to create a class of rooms for mazes; let's call this new class `mazeroom`:

```
mazeroom:  room
;
```

Similarly, we'll create a class for any object that counts as a treasure:

```
treasure:  item
;
```

Note that the only purpose of these two classes is to serve as a flag, so we can determine if an object is considered a treasure or if a room is part of a maze. These classes don't actually implement any new behavior of their own.

Next, we'd add a check in the thief's daemon that looks for treasure objects when the thief is in a room in a maze. We'll add this code to the end of the same daemon we used for the randomly wandering actor above in the previous section; we won't reproduce that entire routine here. This code is added after the `for` loop that comprised the bulk of that daemon.

```
if (isclass(self.location, mazeroom))
{
  /* check contents of current room */
  local cont := self.location.contents;
  local len := length(cont);
  local found := nil;
  local i;
  for (i := 1 ; i <= len ; ++i)
  {
    if (isclass(cont[i], treasure))
    {
      found := true;
      cont[i].moveInto(self);
```

```
    }
  }
  if (found and Me.location = self.location)
    "You notice the thief has helped himself to some items.";
}
```

This is a fairly complicated example of making your actors do something special when certain conditions are met. A simpler example is Lloyd's attempt to sell insurance to the sleeping guard in *Ditch Day Drifter*, which is simply a message that's displayed by Lloyd's movement daemon when Lloyd enters the room with the guard.

You may want some actors to have very complicated scripts that do more than just move the actors around. You may want to implement an actor which goes about his business, doing things in some of the locations he visits. The best way to do this is probably by making your actor a "state machine." One easy way to implement a state machine is to use a number to represent the current state; on each turn, you increment the state number, and use a `switch` statement to take the appropriate action depending on the new state.

For example, suppose you want to implement an actor that enters a building and walks to the study. If someone else is in the study, he'll just wait impatiently for the other person to leave; once he's alone in the study, he'll produce a key and open a safe, conveniently dropping the key near the safe. He'll then take something out of the safe and leave the building.

We'll represent the set of states with numbers. State 1 will be outside the building; when in state 1, we'll enter the building. State 2 will be inside the front hall, and we'll move to the study. In state 3 we'll check to see if we're alone; if not, we'll just stay in state 3 and act impatient. Once we're alone, we'll produce the key and open the safe. In state 4, we'll take the object out of the safe and exit the room, moving back to the front hall. In state 5, we'll leave the building.

The following methods can be used to accomplish this.

```
actorMessage(msg) =
{
  if (Me.location = self.location)
  {
    "\b";
    say(msg);
  }
}
actorMove(newloc, todir, fromdir) =
{
  if (Me.location = self.location)
    "^<<self.thedesc>> leaves to the <<self.todir>>.  ";
  self.moveInto(newloc);
  if (Me.location = self.location)
    "^<<self.thedesc>> enters the room from the <<self.fromdir>>.  ";
}
```

```
actorState = 1
actorDaemon =
{
  switch(self.actorState++)
  {
  case 1:
    self.actorMove(frontHall, 'north', 'south');
    break;
  case 2:
    self.actorMove(study, 'east', 'west');
    break;
  case 3:
    if (self.location = Me.location and not Me.ishidden)
    {
      "Jack crosses his arms and looks at his watch.";
      self.actorState := 3;                              /* remain in state 3 */
    }
    else
    {
      actorMessage('Jack looks around, and is satisfied that
      he is alone.  He searches his pockets, and produces a
      key, which he inserts into the safe and opens it.  He
      nervously removes something from the safe; he doesn\'t
      seem to notice when the key drops to the ground.');
      safeKey.moveInto(self.location);
    }
    break;
  case 4:
    self.actorMove(frontHall, 'west', 'east');
    break;
  case 5:
    self.actorMove(outsideBuilding, 'south', 'north');
    break;
  }
}
```

Note that the `actorMove` and `actorMessage` methods that we implemented provide a convenient mechanism for displaying messages conditionally if the player is present.

Naturally, you'd want to extend the end of this script to give Jack something to do (or somewhere to disappear to) at the end of his appearance. Using this basic technique, you can implement essentially arbitrarily complicated actors. Note that it's easy to jump to a different state rather than progress to the next state—just assign the state variable (`actorState` in this case) to the new state you want to enter on the next turn. This allows your actor to behave intelligently: rather than just following a script without reference to what's going on in the game, the actor can take different

actions depending on the conditions around him. In our simple example above, the only sensitivity to game conditions is that the actor waits to open the safe until he's alone (or thinks he is); you can easily test for much more complex conditions, though.

In deciding how to implement your actors, you should consider the things you actor does most frequently. If your actor mostly moves around on a track, and occasionally does something special in a particular location or when a particular set of conditions is true, then implementing your actor using the techniques for moving on a track is easiest; just check for your special conditions before or after moving. On the other hand, if your actor does something special in many locations and on most turns, using a state machine is easiest.

Talking to Actors

One of the main areas of interaction with actors in adventure games is asking the actors questions. TADS is a bit limited when it comes to artificial intelligence; unfortunately, TADS doesn't make it possible to ask actors general questions such as "what do you think about the role of the media in the upcoming election?" or "why is the sky blue?" Instead, questions addressed to actors in a TADS game are limited to asking about particular objects in the game. Furthermore, all the player can do is ask about an object in general—the player can't ask "why is the fuse outside the tram?", but is limited to "ask lloyd about the fuse."

Another type of interaction is to tell an actor something. As with questions, TADS is too limited to allow a player to tell an actor anything very specific. All that a player can say is something like "tell lloyd about the scroll."

As a game designer, these limitations make your job fairly straightforward. Implementing the "ask" and "tell" routines is essentially the same as implementing any other verb method. Here's an example of a `doAskAbout` method for an actor.

```
verDoAskAbout(actor, iobj) =
{
}
doAskAbout(actor, iobj) =
{
  switch(iobj)                /* indirect object is what is being asked about */
  {
  case insurancePolicy:
    if (insurancePolicy.isbought)
      "\"It's very complicated, but let me assure you,
      it's a very good policy.\"";
    else
      "\"It's a great policy for only a dollar!\"";
    break;
  default:
    "\"I don't know much about that.\"";
```

```
    }
  }
```

You could add more cases as needed to add to Lloyd's knowledge.

Note one strange twist here: the `actor` parameter to the methods is the actor that asked the question—that is, the player, or `Me`. The actor who's being asked is `self`, since the actor is the direct object of the "ask" verb. Note also that we had to implement an empty `verDoAskAbout` method so that the actor can be asked questions at all; by default, all objects inherit a generic error message for `verDoAskAbout`.

You can implement behavior for telling an actor about an object in much the same way, by writing `verDoTellAbout` and `doTellAbout` methods for the actor. Likewise, you can make it possible for the player to give objects to the actor by implementing `verIoGiveTo` and `ioGiveTo` methods for the actor. Note that the `ioGiveTo` method should move the direct object into the actor's inventory (by executing `dobj.moveInto(self)`) if the object is accepted.

Taking Objects from Actors

By default, actors don't let the player take objects that the actor is carrying. This is because all of the routines that try to move an object will call the `verGrab(actor)` method of all of the containers of the object. By default, an actor's `verGrab` method will display a message indicating that the actor won't give up the object so easily; since `verGrab` is called during the validation phase of parsing, displaying a message is sufficient to prevent the command from proceeding.

If you want to allow the player to take objects from an actor, override the actor's `verGrab` method. You could make the method do nothing at all, in which case the actor will allow any object to be taken; or it can display a message only on certain objects, which will prevent only those objects from being taken.

Commanding an Actor

The player can issue a command to an actor by typing something like "lloyd, go north." When this happens, the TADS parser processes the command in almost exactly the same way as a normal command, except that the actor of the command is changed from the default `Me` (the player's actor) to the actor indicated in the command. (In fact, it really is exactly the same processing; when no actor is typed, it's as though the player had typed "me, go north.")

Since most all processing occurs relative to the `actor` parameter to the verb-handling methods (such as `verDoTake(actor)` and `doDrop(actor)`), you don't need to make very many special provisions for commands directed to actors. This type of processing is done almost entirely automatically by the system.

However, in most cases, you don't actually *want* the player to be able to boss your actors around. Most of your actors will have a mind of their own, and many will even be presenting obstacles to the player.

To make it possible for you to control whether the player can control your actors, and to what extent, the TADS parser will always call the `actorAction` method of the selected actor prior to allowing any further processing to proceed. If this routine executes an `exit` statement, the command will go no further, preventing the player from controlling your actor. Since the method receives all of the information on the current command (the verb, direct object, preposition, and indirect object) as parameters, it can choose to allow or disallow particular commands. For allowed commands, do nothing; for disallowed commands, display an appropriate message and then execute an `exit` statement.

At this very moment, we have the necessary techniques, both material and psychological, to create a full and satisfying life for everyone.

— BURRHUS FREDERIC SKINNER, *Walden Two* (1948)

Appendix A

Adventure Definitions

The TADS run-time system provides a great deal of support for text adventure games, but it doesn't pre-define very much specific to adventures. To make the game author's job easier, the file `adv.t` provides definitions for a great many basic objects and classes that underlie most adventure games. These classes are very general, and most games will customize them further, but they vastly reduce the amount of work required to write a game. This section describes these classes, what they do, and how to use them to write a game.

Note that in addition to everything listed in the following pages, `adv.t` also defines the articles "a," "an," and "the."

These definitions become part of your game when you include the standard adventure definitions file `adv.t` in your source file.

Objects, Classes, and Functions

This section lists the objects and functions defined in the standard adventure definitions file `adv.t`.

The descriptions of the objects indicate the object's superclass or superclasses; for complete information on an object, you need to refer to the description of its superclasses as well. The descriptions also indicate the main properties defined in the object; in customizing the object, you will generally need to override one or more of these properties. For example, most descendants of `thing` will specify an `sdesc` to provide the object with a name.

The function descriptions provide information on how to call the functions, what they do, and what they return.

For clarity, the objects are described in inheritance order; that is, the lower-level (parent) classes are described first, then the descendants of those classes, then the descendants of those, and so on. (This is the same order in which they appear in `adv.t`.) However, this can make it difficult to find the description of a particular object, thus the following index.

---

itemcnt

`itemcnt: function(`*list* `)`: Returns a count of the "listable" objects in *list*. An object is listable (that is, it shows up in a room's description) if its `isListed` property is `true`. This function is useful for determining how many objects (if any) will be listed in a room's description.

listcont

`listcont: function(`*obj*`)`: This function displays the contents of an object, separated by commas. The `thedesc` properties of the contents are used. It is up to the caller to provide the introduction to the list (usually something to the effect of "The box contains" is displayed before calling `listcont`) and finishing the sentence (usually by displaying a period). An object is listed only if its `isListed` property is `true`.

listfixedcontcont

`listfixedcontcont: function(`*obj*`)`: List the contents of the contents of any `fixeditem` objects in the `contents` list of the object *obj*. This routine makes sure that all objects that can be taken are listed somewhere in a room's description. This routine recurses down the contents tree, following each branch until either something has been listed or the branch ends without anything being listable. This routine displays a complete sentence, so no introductory or closing text is needed.

listcontcont

`listcontcont: function(`*obj*`)`: This function lists the contents of the contents of an object. It displays full sentences, so no introductory or closing text is required. Any item in the `contents` list of the object *obj* whose `contentsVisible` property is `true` has its contents listed. An Object whose `isqcontainer` or `isqsurface` property is `true` will not have its contents listed.

turncount

`turncount: function(`*parm*`)`: This function can be used as a daemon (normally set up in the `init` function) to update the turn counter after each turn. This routine increments `global.turnsofar`, and then calls `setscore` to update the status line with the new turn count.

addweight

`addweight: function(`*list*`)`: Adds the weights of the objects in *list* and returns the sum. The weight of an object is given by its `weight` property. This routine includes the weights of all of the contents of each object, and the weights of their contents, and so forth.

addbulk

`addbulk: function(`*list*`)`: This function returns the sum of the bulks (given by the `bulk` property) of each object in *list*. The value returned includes only the bulk of each object in the list, and *not* of

the contents of the objects, as it is assumed that an object does not change in size when something is put inside it. You can easily change this assumption for special objects (such as a bag that stretches as things are put inside) by writing an appropriate `bulk` method for that object.

---

incscore

`incscore: function(`*amount*`)`: Adds *amount* to the total score, and updates the status line to reflect the new score. The total score is kept in `global.score`. Always use this routine rather than changing `global.score` directly, since this routine ensures that the status line is updated with the new value.

---

initSearch

`initSearch: function`. Initializes the containers of objects with a `searchLoc`, `underLoc`, and `behindLoc` by setting up `searchCont`, `underCont`, and `behindCont` lists, respectively. You should call this function once in your `preinit` (or `init`, if you prefer) function to ensure that the underable, behindable, and searchable objects are set up correctly.

---

nestedroom

`nestedroom: room`. A special kind of room that is inside another room; chairs and some types of vehicles, such as inflatable rafts, fall into this category. Note that a room can be within another room without being a `nestedroom`, simply by setting its `location` property to another room. The `nestedroom` is different from an ordinary room, though, in that it's an "open" room; that is, when inside it, the actor is still really inside the enclosing room for purposes of descriptions. Hence, the player sees "Laboratory, in the chair." In addition, a `nestedroom` is an object in its own right, visible to the player; for example, a chair is an object in a room in addition to being a room itself.

---

chairitem

`chairitem: fixeditem, nestedroom, surface`. Acts like a chair: actors can sit on the object. While sitting on the object, an actor can't go anywhere until standing up, and can only reach objects that are on the chair and in the chair's `reachable` list. By default, nothing is in the `reachable` list. Note that there is no real distinction made between chairs and beds, so you can sit or lie on either; the only difference is the message displayed describing the situation.

---

beditem

`beditem: chairitem`. This object is the same as a `chairitem`, except that the player is described as lying on, rather than sitting in, the object.

thing

`thing: object`. The basic class for objects in a game. The property `contents` is a list that specifies what is in the object; this property is automatically set up by the system after the game is compiled to contain a list of all objects that have this object as their `location` property. The `contents` property is kept consistent with the `location` properties of referenced objects by the `moveInto` method; always use `moveInto` rather than directly setting a `location` property for this reason. The `adesc` method displays the name of the object with an indefinite article; the default is to display "a" followed by the `sdesc`, but objects that need a different indefinite article (such as "an" or "some") should override this method. Likewise, `thedesc` displays the name with a definite article; by default, `thedesc` displays "the" followed by the object's `sdesc`. The `sdesc` simply displays the object's name ("short description") without any articles. The `ldesc` is the long description, normally displayed when the object is examined by the player; by default, the `ldesc` displays "It looks like an ordinary `sdesc`." The `isIn(`*object*`)` method returns `true` if the object's location is the specified *object* or the object's `location` is an object whose `contentsVisible` property is `true` and that object's `isIn(`*object*`)` method is `true`. Note that if `isIn` is `true`, it doesn't necessarily mean the object is reachable, because `isIn` is `true` if the object is merely visible within the location. The `thrudesc` method displays a message for when the player looks through the object (objects such as windows would use this property). The `moveInto(`*object*`)` method moves the object to be inside the specified *object*. To make an object disappear, move it into `nil`.

item

`item: thing`. A basic item which can be picked up by the player. It has no weight (0) and minimal bulk (1). The `weight` property should be set to a non-zero value for heavy objects. The `bulk` property should be set to a value greater than 1 for bulky objects, and to zero for objects that are very small and take essentially no effort to hold—or, more precisely, don't detract at all from the player's ability to hold other objects (for example, a piece of paper).

lightsource

`lightsource: item`. A portable lamp, candle, match, or other source of light. The light source can be turned on and off with the `islit` property. If `islit` is `true`, the object provides light, otherwise it's just an ordinary object.

hiddenItem

`hiddenItem: object`. This is an object that is hidden with one of the `hider` classes. A `hiddenItem` object doesn't have any special properties in its own right, but all objects hidden with one of the `hider` classes must be of class `hiddenItem` so that `initSearch` can find them.

hider

`hider: item`. This is a basic class of object that can hide other objects in various ways. The `underHider`, `behindHider`, and `searchHider` classes are examples of `hider` subclasses. The class defines the method `searchObj(`*actor, list*`)`, which is given the list of hidden items contained in the object (for example, this would be the `underCont` property, in the case of an `underHider`), and "finds" the object or objects. Its action is dependent upon a couple of other properties of the `hider` object. The `serialSearch` property, if `true`, indicates that items in the list are to be found one at a time; if `nil` (the default), the entire list is found on the first try. The `autoTake` property, if `true`, indicates that the actor automatically takes the item or items found; if `nil`, the item or items are moved to the actor's location. The `searchObj` method returns the *list* with the found object or objects removed; the caller should assign this returned value back to the appropriate property (for example, `underHider` will assign the return value to `underCont`).

Note that because the `hider` is hiding something, this class overrides the normal `verDoSearch` method to display the message, "You'll have to be more specific about how you want to search that." The reason is that the normal `verDoSearch` message ("You find nothing of interest") leads players to believe that the object was exhaustively searched, and we want to avoid misleading the player. On the other hand, we don't want a general search to be exhaustive for most `hider` objects. So, we just display a message letting the player know that the search was not enough, but we don't give away what they have to do instead.

The objects hidden with one of the `hider` classes must be of class `hiddenItem`.

underHider

`underHider: hider`. This is an object that can have other objects placed under it. The objects placed under it can only be found by looking under the object; see the description of `hider` for more information. You should set the `underLoc` property of each hidden object to point to the `underHider`.

Note that an `underHider` doesn't allow the *player* to put anything under the object during the game. Instead, it's to make it easy for the game writer to set up hidden objects while implementing the game. All you need to do to place an object under another object is declare the top object as an `underHider`, then declare the hidden object normally, except use `underLoc` rather than `location` to specify the location of the hidden object. The `behindHider` and `searchHider` objects work similarly.

The objects hidden with `underHider` must be of class `hiddenItem`.

behindHider

`behindHider: hider`. This is just like an `underHider`, except that objects are hidden behind this object. Objects to be behind this object should have their `behindLoc` property set to point to this object.

The objects hidden with `behindHider` must be of class `hiddenItem`.

---

searchHider

`searchHider: hider`. This is just like an `underHider`, except that objects are hidden within this object in such a way that the object must be looked in or searched. Objects to be hidden in this object should have their `searchLoc` property set to point to this object. Note that this is different from a normal container, in that the objects hidden within this object will not show up until the object is explicitly looked in or searched.

The items hidden with `searchHider` must be of class `hiddenItem`.

---

fixeditem

`fixeditem: thing`. An object that cannot be taken or otherwise moved from its location. Note that a `fixeditem` is sometimes part of a movable object; this can be done to make one object part of another, ensuring that they cannot be separated. By default, the functions that list a room's contents do not automatically describe `fixeditem` objects (because the `isListed` property is set to `nil`). Instead, the game author will generally describe the `fixeditem` objects separately as part of the room's `ldesc`.

---

readable

`readable: item`. An item that can be read. The `readdesc` property is displayed when the item is read. By default, the `readdesc` is the same as the `ldesc`, but the `readdesc` can be overridden to give a different message.

---

fooditem

`fooditem: item`. An object that can be eaten. When eaten, the object is removed from the game, and `global.lastMealTime` is decremented by the `foodvalue` property. By default, the `foodvalue` property is `global.eatTime`, which is the time between meals. So, the default `fooditem` will last for one "nourishment interval."

---

dialItem

`dialItem: fixeditem`. This class is used for making "dials," which are controls in your game that can be turned to a range of numbers. You must define the property `maxsetting` as a number specifying the highest number to which the dial can be turned; the lowest number on the dial is always 1. The `setting` property is the dial's current setting, and can be changed by the player

by typing the command "turn dial to *number*." By default, the `ldesc` method displays the current setting.

---

switchItem

`switchItem: fixeditem`. This is a class for things that can be turned on and off by the player. The only special property is `isActive`, which is `nil` if the switch is turned off and `true` when turned on. The object accepts the commands "turn it on" and "turn it off," as well as synonymous constructions, and updates `isActive` accordingly.

---

room

`room: thing`. A location in the game. By default, the `islit` property is `true`, which means that the room is lit (no light source is needed while in the room). You should create a `darkroom` object rather than a `room` with `islit` set to `nil` if you want a dark room, because other methods are affected as well. The `isseen` property records whether the player has entered the room before; initially it's `nil`, and is set to `true` the first time the player enters. The `roomAction(`*actor, verb, directObject, preposition, indirectObject*`)` method is activated for each player command; by default, all it does is call the room's location's `roomAction` method if the room is inside another room. The `lookAround(`*verbosity*`)` method displays the room's description for a given verbosity level; `true` means a full description, `nil` means only the short description (just the room name plus a list of the objects present). `roomDrop(`*object*`)` is called when an object is dropped within the room; normally, it just moves the object to the room and displays "Dropped." The `firstseen` method is called when `isseen` is about to be set `true` for the first time (i.e., when the player first sees the room); by default, this routine does nothing, but it's a convenient place to put any special code you want to execute when a room is first entered. The `firstseen` method is called *after* the room's description is displayed.

---

darkroom

`darkroom: room`. A dark room. The player must have some object that can act as a light source in order to move about and perform most operations while in this room. Note that the room's lights can be turned on by setting the room's `lightsOn` property to `true`; do this instead of setting `islit`, because `islit` is a method which checks for the presence of a light source.

Actor

`Actor: fixeditem, movableActor`. A character in the game. The `maxweight` property specifies the maximum weight that the character can carry, and the `maxbulk` property specifies the maximum bulk the character can carry. The `actorAction(`*verb, directObject, preposition, indirectObject*`)`

method specifies what happens when the actor is given a command by the player; by default, the actor ignores the command and displays a message to this effect. The `isCarrying(`*object*`)` method returns `true` if the *object* is being carried by the actor. The `actorDesc` method displays a message when the actor is in the current room; this message is displayed along with a room's description when the room is entered or examined. The `verGrab(`*object*`)` method is called when someone tries to take an object the actor is carrying; by default, an actor won't let other characters take its possessions.

If you want the player to be able to follow the actor when it leaves the room, you should define a `follower` object to shadow the character, and set the actor's `myfollower` property to the `follower` object. The `follower` is then automatically moved around just behind the actor by the actor's `moveInto` method.

The `isHim` property should return `true` if the actor can be referred to by the player as "him," and likewise `isHer` should be set to `true` if the actor can be referred to as "her." Note that both or neither can be set; if neither is set, the actor can only be referred to as "it," and if both are set, any of "him," "her," or "it" will be accepted.

---

movableActor

`movableActor: qcontainer`. Just like an `Actor` object, except that the player can manipulate the actor like an ordinary item. Useful for certain types of actors, such as small animals.

---

follower

`follower: Actor`. This is a special object that can "shadow" the movements of a character as it moves from room to room. The purpose of a `follower` is to allow the player to follow an actor as it leaves a room by typing a "follow" command. Each actor that is to be followed must have its own `follower` object. The `follower` object should define all of the same vocabulary words (nouns and adjectives) as the actual actor to which it refers. The `follower` must also define the `myactor` property to be the `Actor` object that the `follower` follows. The `follower` will always stay one room behind the character it follows; no commands are effective with a `follower` except for "follow."

---

basicMe

`basicMe: Actor`. A default implementation of the `Me` object, which is the player character. `adv.t` defines `basicMe` instead of `Me` to allow your game to override parts of the default implementation while still using the rest, and without changing `adv.t` itself. To use `basicMe` unchanged as your player character, include this in your game: "`Me: basicMe;`".

The `basicMe` object defines all of the methods and properties required for an actor, with appropriate values for the player character. The nouns "me" and "myself" are defined ("I" is not defined, because it conflicts with the "inventory" command's minimal abbreviation of "i" in certain circumstances, and is generally not compatible with the syntax of most player commands anyway). The `sdesc` is

"you"; the `thedesc` and `adesc` are "yourself," which is appropriate for most contexts. The `maxbulk` and `maxweight` properties are set to 10 each; a more sophisticated `Me` might include the player's state of health in determining the `maxweight` and `maxbulk` properties.

---

decoration

`decoration: fixeditem`. An item that doesn't have any function in the game, apart from having been mentioned in the room description. These items are immovable and can't be manipulated in any way, but can be referred to and inspected. Liberal use of `decoration` items can improve a game's playability by helping the parser recognize all the words the game uses in its descriptions of rooms.

---

buttonitem

`buttonitem: fixeditem`. A button (the type you push). The individual button's action method `doPush(`*actor*`)`, which must be specified in the button, carries out the function of the button. Note that all buttons have the noun "button" defined.

---

clothingItem

`clothingItem: item`. Something that can be worn. By default, the only thing that happens when the item is worn is that its `isworn` property is set to `true`. If you want more to happen, override the `doWear(`*actor*`)` property. Note that, when a `clothingItem` is being worn, certain operations will cause it to be removed (for example, dropping it causes it to be removed). If you want something else to happen, override the `checkDrop` method; if you want to disallow such actions while the object is worn, use an `exit` statement in the `checkDrop` method.

---

obstacle

`obstacle: object`. An `obstacle` is used in place of a room for a direction property. The `destination` property specifies the room that is reached if the obstacle is successfully negotiated; when the obstacle is not successfully negotiated, `destination` should display an appropriate message and return `nil`.

---

doorway

`doorway: fixeditem, obstacle`. A `doorway` is an obstacle that impedes progress when it is closed. When the door is open (`isopen` is `true`), the user ends up in the room specified in the `doordest` property upon going through the door. Since a doorway is an obstacle, use the door object for a direction property of the room containing the door.

If `noAutoOpen` is not set to `true`, the door will automatically be opened when the player tries to walk through the door, unless the door is locked (`islocked` = `true`). If the door is locked, it can be unlocked simply by typing "unlock door", unless the `mykey` property is set, in which case the object specified in `mykey` must be used to unlock the door. Note that the door can only be relocked by the player under the circumstances that allow unlocking, plus the property `islockable` must be set to `true`. By default, the door is closed; set `isopen` to `true` if the door is to start out open (and be sure to open the other side as well).

`otherside` specifies the corresponding doorway object in the destination room (`doordest`), if any. If `otherside` is specified, its `isopen` and `islocked` properties will be kept in sync automatically.

---

lockableDoorway

`lockableDoorway: doorway`. This is just a normal doorway with the `islockable` and `islocked` properties set to `true`. Fill in the other properties (`otherside` and `doordest`) as usual. If the door has a key, set property `mykey` to the key object.

---

vehicle

`vehicle: item, nestedroom`. This is an object that an actor can board. An actor cannot go anywhere while on board a vehicle (except where the vehicle goes); the actor must get out first.

---

surface

`surface: item`. Objects can be placed on a surface. Apart from using the preposition "on" rather than "in" to refer to objects contained by the object, a `surface` is identical to a `container`. Note: an object cannot be both a `surface` and a `container`, because there is no distinction between the two internally.

---

container

`container: item`. This object can contain other objects. The `iscontainer` property is set to `true`. The default `ldesc` displays a list of the objects inside the container, if any. The `maxbulk` property specifies the maximum amount of bulk the container can contain.

---

openable

`openable: container`. A container that can be opened and closed. The `isopenable` property is set to `true`. The default `ldesc` displays the contents of the container if the container is open, otherwise a message saying that the object is closed.

qcontainer

`qcontainer: container`. A "quiet" container: its contents are not listed when it shows up in a room description or inventory list. The `isqcontainer` property is set to `true`.

lockable

`lockable: openable`. A container that can be locked and unlocked. The `islocked` property specifies whether the object can be opened or not. The object can be locked and unlocked without the need for any other object; if you want a key to be involved, use a `keyedLockable`.

keyedLockable

`keyedLockable: lockable`. This subclass of `lockable` allows you to create an object that can only be locked and unlocked with a corresponding key. Set the property `mykey` to the `keyItem` object that can lock and unlock the object.

keyItem

`keyItem: item`. This is an object that can be used as a key for a `keyedLockable` or `lockable-Doorway` object. It otherwise behaves as an ordinary item.

transparentItem

`transparentItem: item`. An object whose contents are visible, even when the object is closed. Whether the contents are reachable is decided in the normal fashion. This class is useful for items such as glass bottles, whose contents can be seen when the bottle is closed but cannot be reached.

basicNumObj

`basicNumObj: object`. This object provides a default implementation for `numObj`. To use this default unchanged in your game, include in your game this line: "`numObj: basicNumObj`".

basicStrObj

`basicStrObj: object`. This object provides a default implementation for `strObj`. To use this default unchanged in your game, include in your game this line: "`strObj: basicStrObj`".

deepverb

`deepverb: object`. A "verb object" that is referenced by the parser when the player uses an associated vocabulary word. A `deepverb` contains both the vocabulary of the verb and a description of available syntax. The `verb` property lists the verb vocabulary words; one word (such as `'take'`) or a pair (such as `'pick up'`) can be used. In the latter case, the second word must be a preposition, and may move to the end of the sentence in a player's command, as in "pick it up." The `action(`*actor*`)` method specifies what happens when the verb is used without any objects; its absence specifies that the verb cannot be used without an object. The `doAction` specifies the root of the message names (in single quotes) sent to the direct object when the verb is used with a direct object; its absence means that a single object is not allowed. Likewise, the `ioAction(`*preposition*`)` specifies the root of the message name sent to the direct and indirect objects when the verb is used with both a direct and indirect object; its absence means that this syntax is illegal. Several `ioAction` properties may be present: one for each preposition that can be used with an indirect object with the verb.

The `validDo(`*actor, object, seqno*`)` method returns `true` if the indicated object is valid as a direct object for this actor. The `validIo(`*actor, object, seqno*`)` method does likewise for indirect objects. The *seqno* parameter is a "sequence number," starting with 1 for the first object tried for a given verb, 2 for the second, and so forth; this parameter is normally ignored, but can be used for some special purposes. For example, `askVerb` does not distinguish between objects matching vocabulary words, and therefore accepts only the first from a set of ambiguous objects. These methods do not normally need to be changed; by default, they return `true` if the object is accessible to the actor.

The `doDefault(`*actor, prep, indirectObject*`)` and `ioDefault(`*actor, prep*`)` methods return a list of the default direct and indirect objects, respectively. These lists are used for determining which objects are meant by "all" and which should be used when the player command is missing an object. These normally return a list of all objects that are applicable to the current command.

Prep

`Prep: object`. A preposition. The `preposition` property specifies the vocabulary word.

Verbs in the Adventure Definitions File

In the list below, the alternative vocabulary words for the verb are listed with slashes (/) between them; the alternate syntaxes are listed with commas between. When the verb takes objects, the root of the message the parser generates in shown in parentheses following the syntax description; these are prefaced with `do`, `verDo`, `io`, and `verIo`, as appropriate. See the section on the parser in chatper 4 for details.

againVerb: again/g

askVerb: ask *direct-object* about *indirect-object* (AskAbout)

attachVerb: attach *direct-object* to *indirect-object* (AttachTo)

attackVerb: attack/kill/hit *direct-object* with *indirect-object* (AttackWith)

boardVerb: get in/get into/board *direct-object* (Board)

centerVerb: center *direct-object* (Center)

cleanVerb: clean *direct-object* (Clean), clean *direct-object* with *indirect-object* (CleanWith)

climbVerb: climb *direct-object* (Climb)

closeVerb: close *direct-object* (Close)

digVerb: dig/dig in *direct-object* with *indirect-object* (DigWith)

drinkVerb: drink *direct-object* (Drink)

dropVerb: drop/put down *direct-object* (Drop), drop/put down *direct-object* on *indirect-object* (PutOn)

dVerb: down/d

eatVerb: eat *direct-object* (Eat)

eVerb: east/e

fastenVerb: fasten *direct-object* (Fasten)

getOutVerb: get out/get out of/get off/get off of *direct-object* (Unboard)

giveVerb: give/offer *direct-object* to *indirect-object* (GiveTo)

helloVerb: hello/hi/greetings

inspectVerb: inspect/examine/look at/x *direct-object* (Inspect)

inVerb: in/go in/enter

iVerb: inventory/i

jumpVerb: jump, jump *direct-object* (Jump)

lieVerb: lie/lie on/lie in/lie down/lie down on/lie down in *direct-object* (Lieon)

lockVerb: lock *direct-object* (Lock), lock *direct-object* with *indirect-object* (LockWith)

lookBehindVerb: look behind *direct-object* (LookBehind)

lookInVerb: look in *direct-object* (Lookin)

lookThruVerb: look through/look thru *direct-object* (Lookthru)

lookUnderVerb: look under/look beneath *direct-object* (LookUnder)

lookVerb: l/look/look around

moveNVerb: move north/move n/push north/push n *direct-object* (MoveN) *Note: moveSVerb, moveEVerb, moveWVerb, moveNEVerb, moveNWVerb, moveSEVerb, and moveSWVerb are defined similarly.*

moveVerb: move *direct-object* (Move), move *direct-object* to *indirect-object* (MoveTo), move *direct-object* with *indirect-object* (MoveWith)

nVerb: north/n

neVerb: northeast/ne

nwVerb: northwest/nw

openVerb: open *direct-object* (Open)

outVerb: out/go out/exit

plugVerb: plug *direct-object* into *indirect-object* (PlugIn)

pokeVerb: poke *direct-object* (Poke)

pullVerb: pull *direct-object* (Pull)

pushVerb: push *direct-object* (Push)

putVerb: put/place *direct-object* in *indirect-object* (PutIn), put/place *direct-object* on *indirect-object* (PutOn)

quitVerb: quit

readVerb: read *direct-object* (Read)

removeVerb: take off *direct-object* (Unwear)

restartVerb: restart

restoreVerb: restore *direct-object* (Restore)

standOnVerb: stand on *indirect-object* (Standon)

standVerb: stand/stand up/get up

saveVerb: save *direct-object* (Save)

sayVerb: say *direct-object* (Say)

scoreVerb: score/status

scriptVerb: script *direct-object* (Script)

screwVerb: screw *direct-object* (Screw), screw *direct-object* with *indirect-object* (ScrewWith)

seVerb: southeast/se

showVerb: show *direct-object* to *indirect-object* (ShowTo)

sitVerb: sit on/sit in/sit/sit down/sit down in/sit down on *direct-object* (Siton)

sleepVerb: sleep

sVerb: south/s

swVerb: southwest/sw

takeVerb: take/pick up/get/remove *direct-object* (Take), take/pick up/get/remove *direct-object* out of *indirect-object* (TakeOut), take/pick up/get/remove *direct-object* off/off of *indirect-object* (TakeOff)

tellVerb: tell *direct-object* about *indirect-object* (TellAbout)

terseVerb: terse

throwVerb: throw/toss *direct-object* at *indirect-object* (ThrowAt)

touchVerb: touch *direct-object* (Touch)

turnOffVerb: turn off/deactivate/switch off *direct-object* (Turnoff)

turnOnVerb: activate/turn on/switch on *direct-object* (Turnon)

turnVerb: turn *direct-object* (Turn), turn *direct-object* with *indirect-object* (TurnWith), turn *direct-object* to *indirect-object* (TurnTo)

typeVerb: type *direct-object* on *indirect-object* (TypeOn)

undoVerb: undo

unfastenVerb: unfasten/unbuckle *direct-object* (Unfasten)

unlockVerb: unlock *direct-object* (Unlock), unlock *direct-object* with *indirect-object* (UnlockWith)

unplugVerb: unplug *direct-object* (Unplug), unplug *direct-object* from *indirect-object* (UnplugFrom)

unscrewVerb: unscrew *direct-object* (Unscrew), unscrew *direct-object* with *indirect-object* (UnscrewWith)

unscriptVerb: unscript

uVerb: up/u

verboseVerb: verbose

versionVerb: version

waitVerb: wait/z

wearVerb: wear *direct-object* (Wear)

wVerb: west/w

yellVerb: yell

Prepositions in the Adventure Definitions

Each preposition is shown with its object, and the list of vocabulary words associated with the object, with slashes between equivalent words. Note that the two-word prepositions, such as "down on," are actually written as one word, such as "downon," in the definition file, because the parser collapses two-word prepositions into a single word by removing the intervening space.

aboutPrep: about

aroundPrep: around

atPrep: at

behindPrep: behind

betweenPrep: between/in between

dirPrep: north/south/east/west/up/down/northe/ne/northw/nw/southe/se/southw/sw

inPrep: in/into/in to/down in

fromPrep: from

offPrep: off/off of

onPrep: on/onto/on to/down on/upon

outPrep: out/out of

overPrep: over

thruPrep: through/thru

toPrep: to

withPrep: with

I hate definitions.
— BENJAMIN DISRAELI, *Vivian Grey* (1826)

Appendix B

Better Adventures

We now put aside the technical aspects of writing TADS programs, and turn to some musings on the creative side of writing adventures. TADS makes it possible to write games that are technically polished with little programming; it is up to the author, however, to provide the creativity and storytelling skill needed to make the game fun to play. This section describes some of the stylistic points of writing text adventures.

Style in Adventure Writing

The problem most adventure authors encounter immediately is the technical difficulty of writing a parser and all the other nasty bits of code that underlie an adventure. Most authors don't get very far into writing that code; those that do more often than not get lost in the never-ending quest for new features in their parsers, and never get around to writing a fun game, and those that don't end up with simple parsers, which most players find an inconvenience that hurts playability, and imposes severe limits on the range of actions the game can carry out.

TADS frees authors of the need to become adept programmers before they can write a good adventure game. By providing a ready-made parser, a huge chunk of the technical side of writing a game is done before the game is even started. TADS not only gives you a fine parser, but it also provides facilities for saving and restoring games, creating transcripts (a log of all the text in a game), dealing with the operating system, managing memory, initializing data structures, and virtually everything else that's generic to all adventure games. TADS will satisfy both authors who are not interested in the technical details of the parser and other low-level software, and players and authors who want a sophisticated parser.

But even with all of this, writing a good text adventure is not easy; surmounting the technical difficulties is only half the battle. Now it's necessary to design a good plot, challenging puzzles, interesting characters, and a satisfying conclusion. In this section I will offer some ideas about what makes a good game.

Problems with Adventures

Adventure games all have a major problem: they pretend to be what they're not. The degree to which you address this problem in your games is, in large part, the measure of how good your games are.

Adventures are simulations. Unfortunately, most adventures claim to be simulations of the real world; usually, this claim is implicit in the fact that many objects in the game act, superficially, at least, like their physical counterparts. The reason this is unfortunate is that no adventure game written so far has adequately simulated the real world. Although game objects resemble physical objects in some ways, the game objects are extremely limited in most other ways.

Adventures also pretend to be more intelligent than they are. Text adventures give the player a prompt which says, effectively, "Type anything you want, and I'll do as you ask." This is totally unrealistic. The prompt should really say, "Type one of the few commands I understand, using exactly the correct syntax that I expect, and I may just possibly be able to apply the command as a special case I've been programmed to accept." The TADS parser is about as smart as adventure game parsers come, but it will quickly disappoint anyone other than experienced text adventure players (who already know how limited these games tend to be).

Unfortunately, with the present level of computer hardware and software, it is simply not feasible to write a perfect and complete simulation of the real world, nor to design a parser that can understand every valid sentence (or, for that matter, even any reasonable subset of valid sentences) in a human language. These limitations are not immediately escapable. However, authors can work within these limitations to improve on the traditional text adventure.

The key is to choose a small universe that you can model completely. Don't try to simulate the real world; it's too complex. Instead, design a small imaginary world, which has relatively few possible operations on objects, and few relationships between objects. Make these operations and relationships clear to the player (in documentation, for example). If you choose a small enough set, you may be able to make your simulation robust and complete; the operations and relationships should be general enough that they are applicable to most if not all objects in the game.

You may think that this would make a game uninteresting, but this need not be the case. In fact, it makes adventures much harder to design, but, when done correctly, much more intuitive and satisfying to the player.

One trick that can help you do this is to avoid the use of verbs outside of a very small set. A certain number of verbs will work on pretty much any object in any adventure game; for example: take, drop, open, close, examine, put an object in (or on) another object, give an object to a character. All games have these verbs. Now, imagine writing a game with only these verbs, plus one more special verb: "use." This last verb invokes the "characteristic behavior" of an object. For example, "use button" means to push the button; "use knob" means to turn the knob; "use sandwich" means to eat the sandwich.

It may appear that limiting your verbs in this manner makes puzzles too obvious, since the player doesn't have to think of the actual thing to do with an object, but can just "use" it and have the

game figure out what to do. In fact, it encourages you, the designer, to choose objects that have exactly one very obvious use; doing so means that the "use" verb detracts nothing from the puzzle, since it's obvious to everyone that you push a button and turn a knob, and that you'd do nothing else with them. Hence, this forces you to avoid obscure verbs that apply to just one object as you choose your puzzles. In particular, it prevents you from using obscure relationships between objects that occur only one place in your game; if you can't support the relationship in general, don't support it at all.

This method of puzzle design has another advantage: it usually results in your puzzles being more logical and subtle. The reason is that you must construct interesting situations totally within the context of the very simple verbs in your system and the obvious uses of objects.

Of course, you should use the verb "use" for all of these activities only in the design phase, to force you to avoid vocabulary puzzles and puzzles based on unusual relationships that don't apply to objects in general in your game. When you actually write the game, you should go ahead and use the verbs that describe the characteristic behaviors of your objects. Thus, when the player actually plays the game, he'll type "push button" and "turn knob."

Writing a Game

Every adventure has a plot. In the Original Adventure and many games like it, the plot consisted of nothing more than exploring and gathering treasure. Most recent games, on the other hand, are more like novels whose evolution the player chooses. Most people prefer games with more plot. However, players also like freedom of choice; so, your plot should not impose a particular path through the game, but should instead serve as a backdrop, a context for the events within the game. You must try to balance the story you are telling with a player's desire to choose his own path through the story.

The basis for your plot is as unlimited as your imagination. Adventures have been written in such diverse genres as science fiction, fantasy, mystery, international intrigue, horror, and romance. Adventures cover as wide a spectrum as literature.

The main plot elements of adventures are exploration and problem-solving. While providing an interesting area to explore can stimulate the imagination, most players prefer being able to interact with their environment to simply wandering around it. This is where problem-solving comes in.

In designing puzzles, you should be careful to motivate the solutions; never withhold clues that are needed to solve a puzzle. Disguise the clues, certainly, but be sure the clues are present.

My basic principle of designing puzzles is that the player should always know what he's trying to accomplish. Metaphorically, a player should always be able to find a locked door before he finds the key. This way, he knows his goals. The most frustrating feeling while playing an adventure is that you don't know what you're supposed to do next. If you find a locked door, you know that you have to find some way to open it; you know *what* to do, so the trick is to figure out how to do it. Hence, I like to design games so that a set of short-term goals is clearly visible from the outset, so the player knows where work is to begin.

A well-designed adventure is a self-consistent world, not necessarily the real world, in which a player can conduct experiments to learn how objects and characters behave; he solves puzzles by learning the behavior of an object or character, and turning that to his advantage. This is probably why science fiction and fantasy adventures are so popular; these genres offer limitless possibilities for creating objects that behave in novel ways.

Since the core of problem solving should be experimentation and observation, objects that behave in special ways shouldn't be labelled as such. Instead, they should have physical characteristics which invite experimentation, and should respond to experiments with useful feedback. It is especially bad to provide an object that behaves in a novel way, but force the player simply to think to try the one special thing that it does; something should invite him to find its special characteristics, and its behavior should be consistent.

For example, you might design a machine, described as "a small metal box with a button, a short plastic hose on one side, and a large metal pipe on the other side." When the button is pushed, "a loud hissing comes from the plastic hose for a moment, then a large drop of clear liquid drops out of the pipe, which hits the floor and quickly evaporates into a white cloud of vapor." If the player puts the plastic hose in a glass of water and the button is pushed, "the water is sucked into the plastic tube, and few moments later a block of ice is dropped out of the pipe." This allows the player to learn by experimentation what the machine does, which is more fun for the player than if you had labelled the machine "a freezer" or some such.

Another type of puzzle that makes for good games is the kind that uses ordinary objects in an unusual but logical manner. For example, you could use a ladder to cross a chasm, or an incandescent bulb for its heat rather than its light. It's important that the use be logical; you don't want the player to have to guess a completely unmotivated use of an object. You want the player to think, "I need a heat source, but all I've got is this light bulb," and then realize that the light bulb *is* a heat source. It's also important that the properties of an object be consistent; if the player tries to take the light bulb while it's lit, he should get burned.

Another common type of puzzle involves hidden objects. When objects are hidden, the player should be able to find them without resorting to looking behind and under everything in every room. For example, if you hide a crystal statuette under a seat cushion, someone sitting on the seat may hear a sound like breaking glass coming from under the cushion; looking under the cushion reveals a smashed statuette. (Providing a clue that doesn't require starting the game over would be nice, too, though.)

A feature that is difficult for a designer to implement, but makes the game more realistic, is an object that is used for several purposes. In many adventures, the player figures out what each object is for, and, once used, feels free to discard it. You can set up subtle logical puzzles simply by using an object for more than one of its properties, since players might assume that the obvious useful property of the object is the only one and not bother looking for other uses of the object.

Most players prefer games in which they can work on several puzzles at once, since each player will be able to solve different puzzles more easily. This is why I generally structure my games overall with an introduction, then the body of the game, and finally an end-game. The introduction and end-game tend to be fairly linear; that is, each event leads to the next event in a fixed order. In the

linear portions of the game, then, the player can only work on one puzzle at a time, and must solve it before he moves on to the next. The body of the game, though, consists of many parallel puzzles; though these might have some interdependencies, I generally try to make it possible to solve them in any order. I try to keep the linear portions of the game as short as possible, leaving the course of events in most of the game up to the player.

Designing games is a subtle and complex art. Just as most good authors are well-read, you will probably become a better adventure game author if you play a few good adventure games. And nothing is a substitute for experience; your games will improve with each new one you write. TADS gives you an excellent tool for creating games; the rest is up to you.

In this final chapter, we approach style in its broader meaning:
style in the sense of what is dinstinguished and distinguishing.
Here we leave solid ground. Who can confidently say what ignites
a certain combination of words, causing them to explode in the mind?

— E. B. White, *The Elements of Style* (Third Edition, 1979)

Appendix C

The Compiler

This chapter describes how to use the TADS Compiler, and explains the options and parameters that the compiler accepts.

The TADS Compiler

The TADS Compiler is the program that reads your adventure source file, checks it for correct syntax, and converts it to a binary representation that can be executed by the TADS run-time system. Generally, the binary version of your game is considerably smaller than the source; the symbolic TADS code is converted to a byte-code similar to machine language, the strings are compressed using a variable byte length encoding (which also makes it difficult for players to cheat by looking at your program's strings), and the objects are converted to compact data structures.

This section describes the options and parameters that you use to control the TADS Compiler. Note that the format of the commands shown in this section is general, and some operating systems may have slightly different conventions; consult your system-specific release notes for information on using the Compiler on your computer. Note also that you may have to perform some minor configuration on your computer (for example, setting the command path so that your command interpreter can find the Compiler program file) before you can run the Compiler.

Note that on Macintosh systems, the compiler does not have a command line at all; instead, it uses a dialog box and pull-down menus to specify compiler options. All of the options described in this appendix are accessible in the Macintosh version; it should be easy to identify the menu selections and dialog entries corresponding to the options described below.

Running the Compiler

You can run the TADS Compiler simply by typing its name, `tc`, followed by the name of the file you want to compile. For example:

```
tc sample
```

This invokes the TADS Compiler, and tells it to compile the file `sample.t` (using the appropriate local conventions to add the extension `.t` to the given filename), and to write the binary version of the game to the file `sample.gam`.

If any errors occur during compilation, the Compiler provides an explanatory message and tells you the file and line number where the error occurred. If errors (other than warnings) occur, the binary file is not created, since the game would not be playable. (On Macintosh systems, due to the inability of some text editors to go directly to a line by number, the compiler also displays the text of lines at which errors are detected.)

Compiler Options

The TADS Compiler accepts several options as part of its command to control its operation. All of the options consist of a dash, followed by one or more option letters, possibly followed by a parameter. The options are specified before the name of the source file:

`tc` *options file*

For example, to compile `sample.t` and generate source-level debugging information, you would type this command:

```
tc -ds sample
```

Note that with options that take a parameter, you can either run the option and its parameter together without a space, or you can put a space between them; it makes no difference. For example, the following two commands are equivalent:

```
tc -ic:\tads -od:sample sample
tc -i c:\tads -o d:sample sample
```

Note, however, that the memory options require the memory type letter *immediately* after the `-m` option; the size, however, can be separated by a space. Hence, the following two commands are valid:

```
tc -mh5000 sample
tc -mh 5000 sample
```

Following is the complete list of Compiler options. For a brief list of the options, just type `tc` without any parameters; the Compiler will display a list of the options it accepts and what they do.

`-ds` Generate source-level debugging information. This option causes the compiler to include information in the binary game file that the TADS Debugger uses to trace execution of the program and correlate the generated code to the source files. You must use this

option if you wish to use the Debugger with your game. Using this option increases the size of the game file, so you will normally compile *without* the `-ds` option once you have finished debugging your game.

`-i`*path* Adds *path* to the include path. When you use the `#include` directive to include a file, and enclose the file in angle brackets (`<` and `>`), the list of directories specified with `-i` options is searched, in the order given, for the file. Note that previous versions of TADS required you to end the path specification with your operating system's path separator character; this is no longer required, but a trailing path separator character is harmless if present.

`-l`*file* Load a pre-compiled file prior to compiling the source program. The file must have been previously created with the `-w` option. This option allows you to speed up the compile process by compiling the include files that you commonly use separately from your program, then loading their binary forms when you compile your program. Loading a binary file is much faster than compiling the source file it came from.

Note that only one file can be loaded with the `-l` option. However, you can easily pre-compile several include files simply by making a short file that contains only `#include` directives listing the files you want to gather together into a single pre-compiled binary.

Note also that files can be included only once; if you attempt to include a file twice, the compiler will ignore the second directive. This makes it convenient to use pre-compiled files, because you do not need to change your source files—simply leave the normal `#include` directives in your program. If you use the `-l` option to load pre-compiled versions of these source files, the `#include` directives in your program will be ignored; if you do not use `-l`, the files will be included as normal. This way, you do not need to change your source file when you wish to use the `-l` option.

`-m`*item size* Set memory size for a given *item* to *size*. The compiler will attempt to pre-allocate a certain amount of space for several types of objects when it initializes itself; if you do not have enough memory in your computer, you must lower one of these sizes. Alternatively, if your program exceeds one of the pre-defined sizes, you must increase the size. The *item* specifiers are:

`g goto` label table size

`h` heap size

`l` local symbol table size

`p` parse node pool size

`s` stack size

To determine the default sizes on your computer, run the Compiler with the option "`-m?`"; this will display the memory options and their default values.

With no second option letter, the `-m` option specifies the size of the virtual object cache; this is the memory area that the compiler uses to contain the compiled representation of game objects and functions. Since the compiler has a virtual memory system, the cache size can be smaller than the actual memory needs of your game. Normally, you will not need to set the cache size, since the compiler automatically allocates additional memory for the cache as needed. However, since the compiler also allocates non-cache memory during compilation, the cache can grow so large that, in low-memory situations, your computer runs out of space for non-cache memory. If this happens, the compiler will exit and issue an error message telling you how big the cache was when the compiler ran out of non-cache memory; you should re-run the compiler specifying a smaller size with the `-m` parameter. For example, if the compiler runs out of memory and tells you that the cache is 256,000 bytes, you could try re-running with `-m 200000` to specify that the cache should grow no larger than 200,000 bytes.

`-o`*file* Set output filename to *file*. If the filename does not have an extension, the default extension `.gam` will be appended to the name (following the conventions of your operating system).

`-p` Pause before terminating the compiler. This option is useful for operating systems that operate from a "desktop"; on these systems, the information displayed by the compiler is usually cleared off the screen as soon as the program terminates, which doesn't leave enough time to see the messages the compiler produces. When the Compiler pauses, it prompts you to strike a key; when you do so, it continues. (Note that this option is not necessary on Macintosh systems, because the compiler does not exit after compilation until you explicitly select the "Quit" item from the "File" menu, or hit the "Quit" button.)

`-s` Enables statistics. After the game has been compiled, a set of statistics on memory usage will be displayed.

`-tf` *file* Use *file* to hold swapping information. By default, the file `TADSSWAP.DAT` in the current directory is used. If you are using a RAM disk, you may wish to place the swap file on the RAM disk to improve performance.

`-ts` *size* Limit the swap file to *size* bytes. You may wish to use this option if you have limited space on the disk containing the swap file.

`-t-` Turn off swapping. This forces the compiler to keep the entire game in memory. Note that, even when swapping is enabled, the compiler won't swap until necessary—that is, until the cache is full and no more space can be allocated for the cache.

`-w`*file* Writes the pre-compiled version of the source to *file*. No default extension is applied. When you use this option, any `-o` option that you have specified is ignored, since a game file is not produced. Files written with `-w` are intended to be loaded with the `-l` option; see the description of `-l` for more information on how to use pre-compiled files.

`-Za` Suppress generation of argument-checking code. Whenever a function is called, TADS version 2 will check to make sure that the function received the same number of arguments that it was expecting. Since TADS version 1 did not do this, some games have a mismatch between actual and formal parameters for some functions and methods; with TADS 2, this will cause a run-time error. You can use the `-Za` option to suppress these run-time errors, which in many cases are harmless.

`-1` Enable all version 1 compatibility options.

`-1a` Same as `-Za`: disable run-time argument checking.

`-1e` Ignore semicolons between a closing brace and an `else` keyword. TADS version 1 allowed this construction, even though it is not actually allowed by the TADS language's grammar. Version 2, by default, does not allow semicolons between a closing brace and an

`else` keyword; using the `-1e` option will allow your old code to compile if it contains this construction.

`-1k` — Disable the new TADS version 2 keywords: `for`, `do`, `switch`, `case`, `default`, and `goto`. When you specify `-1k`, all of these words can be used as identifiers. Some older TADS programs use one or more of these words as identifiers (in particular, `do` for a direct object argument), which will cause errors when compiled with TADS version 2. Specifying the `-1k` option will remove these words from the keyword table, making them available for use as identifiers. Of course, specifying `-1k` will make it impossible to use any of these new language constructs in your game, so it should only be specified with old code that you haven't updated for version 2 yet.

`-1d`*keyword* — Changes the `do` keyword (used in the `do-while` construct) to a new keyword of your choosing. If you have a game written for TADS version 1 that used "`do`" as an identifier (such as the name of a local variable or a parameter to a method or function), but you still want to use the new language features in TADS version 2 (such as `switch` and `for`), you can use the `-1d` option to change the `do` keyword, so that it doesn't interfere with your game. For example, if you specify "`-1d DO`", you will be able to use "`do`" as a variable name, and you can write `do-while` loops using the keyword `DO` instead of `do`. If you use this option instead of `-1k`, you can still use `switch`, `for`, and the other new version 2 language constructs.

Configuration Files

On some operating systems, the TADS compiler allows you to place a set of command line options into a "configuration file" that is read each time the compiler is run. This file is named `CONFIG.TC`. TADS looks for this file first in the current directory, and if it fails to find it, in the same directory as your TADS Compiler executable. This allows you to have multiple configurations: one default configuration, stored in the `CONFIG.TC` file in your TADS compiler directory; and then special per-game configurations, stored in the `CONFIG.TC` files in your game source directories. If the file does not exist in either the current directory or in the TADS compiler directory, no configuration file is used.

The configuration file simply contains compiler options. The file is a standard text file; each line within the file can have as many options as you want, and the file can have as many lines as you want. Any blank lines within the file are ignored.

If you want to create a configuration file that specifies that the virtual object cache is to be limited to 128,000 bytes, that the swap file is to be named `SWAP.DAT` on the `D:` disk, and that the directory `c:\tads\include` is to be searched for header files, you could make a configuration file like this:

```
-m 128000 -tf d:\swap.dat
-I c:\tads\include
```

Note that options specified on the command line always override options in the configuration file. The configuration file is simply a convenient way to store default options that you often use. Suppose you are using the above configuration file, and you type this command:

```
tc -m 256000 -I c:\myinc mygame.t
```

The `-m 256000` option overrides the configuration file's `-m 128000` option, so the configuration file's version is ignored. Include directory options are a little different, in that they're additive: the command line's `-I c:\myinc` is *added* to the include path. However, since the command line takes precedence over the configuration file, the command line include path is added *before* the path in the configuration file.

You are not required to use a configuration file, and no error is generated if a configuration file is not found. The configuration file is simply a convenient way to store options that you frequently use, so you don't have to type them every time you run the compiler.

We must learn to explore all the options and possibilities that confront us in a complex and rapidly changing world.

— JAMES WILLIAM FULLBRIGHT, *Speech in the Senate* (1964)

Appendix D

The Run-Time System

This section describes the TADS Run-Time system that players use to run your adventure. The run-time system is very simple, so players can learn to run your games very easily.

The TADS Run-Time System

After you have compiled your game with the TADS Compiler, and you have tested the resulting program thoroughly, you are ready to give it to players. They are, after all, the reason you wrote the game.

Fortunately, to play the game, a player doesn't need to learn nearly as much as you did to write it. A simple run-time module is used to execute your game, called `tr` for TADS Run-time.

This program, when run without parameters, looks for a file called `tads.gam` in the current directory; if it finds it, the game is loaded and executed. Hence, players need only install two files on their computer in order to play your game: the `tr` program file, and the `tads.gam` data file.

On systems with a command line, you can specify the name of a game to run. For example, to execute `sample.gam`, you would type:

```
tr sample
```

However, in finished games, many authors may wish to create "self-loading" game files for distribution. A self-loading file contains the game data file (in this case, `sample.gam`) in the same file with the run-time executable, so only one file is needed to play the game. The process of building a self-loading game file is described later in this chapter.

Portability

By using TADS, your game is immediately portable to every system on which the TADS Compiler has been implemented. You do not need to change your source file at all. In addition, since the

TADS version 2 binary game file is fully portable, you can simply transfer your binary files to other computer systems and run them without recompilation.

Run-Time System Features

The TADS run-time system offers players many sophisticated features, many of which are not at all typical of text adventures. By using TADS, your game automatically supports all of these features without any work on your part. In addition, your game is immediately portable to every system on which the TADS Compiler has been implemented.

- Sophisticated command-line parsing, allowing multiple commands on a line, multiple direct objects, use of "all" and "except," and full support for adjectives, prepositions, nouns, and verbs. The parser can also automatically prompt for more information when needed, saving the player the trouble of retyping an entire command when something was left out.
- Operating-system specific file dialogs, simplifying the task of selecting files for saving and restoring games, and making the game consistent with the operating system's look and feel.
- On many systems, command line editing and recall is supported. Command line editing allows a player to edit the current command with arrow keys and other editing keys such as "delete," "home," and "end," making it easy to correct errors. Command line recall saves the most recent commands, and allows the player to recall them using editing keys (generally, several thousand characters of past commands can be saved, so practically an entire game can be recalled). Once recalled, a command can be edited and entered again. This is often a time-saving feature, and one that will be appreciated by regular computer users who are accustomed to such features in their operating system command shell.
- On many systems, TADS saves text that has scrolled off the screen, and allows the player to recall the text by using "review mode." After pressing a special key, the player can use arrow and paging keys to look through the most recent screens of text. Generally, dozens of pages of text are saved, allowing the player to review well back into the game.

Run-Time Options

The run-time system has several options, although they will generally not be needed by people who play your game.

First, the run-time has a set of memory size options. The `-m` *size* option specifies a maximum size (in bytes) for the virtual object cache at run-time; this works exactly the same as the compiler's `-m` *size* switch. This will generally not be necessary at run-time, because the run-time system does not

usually need to allocate memory outside of the cache while the game is being played. However, if you run into a low-memory problem during game play, you can use `-m` to limit the cache size.

Two other memory options specify the size of run-time memory areas. The `-mh` *size* option specifies the size (in bytes) of the run-time heap, and the `-ms` *size* option specifies the size (in stack elements) of the run-time stack. Under normal circumstances, it won't be necessary to specify either of these parameters; however, if your game encounters run-time errors related to these two sizes, you can use these options to increase the limits.

The `-u` *size* option specifies the size (in bytes) of the undo log, which is a memory area that records the information necessary to undo player actions. The default size for this memory area is large enough to hold information to undo about a hundred turns in a typical game.

The `-tf` *file* option specifies the location of the swap file, in the same manner as the compiler's `-tf` option, and the `-ts` *size* option limits the size of the swap file, in the same way as the compiler's `-ts` option.

The remaining options are related to game testing, and are described in the next section.

Testing Your Game

The TADS run-time provides a facility that allows you to test your game to make sure it runs the same way after a change. First, you write a "script" file, which is simply a list of commands for your game—exactly the same as commands that a user would type while playing the game. Next, you use a special option that tells the run-time system to read commands from this script file rather than from the keyboard. Then, you use another special option that captures all of the output of your game into a file rather than sending it to the screen.

The run-time has a special option that makes it easy to create a script file. Run your game with the `-o` *file* option: this will run the game as normal, with input coming from the keyboard and output going to the screen, but will *also* write every command you type to the file you specified. Play through your game as normal, typing the set of commands you want to be able to test later. When you're done, quit the game. The file you specified with the `-o` option now contains an input script consisting of all of the commands you typed while playing.

Next, you can generate a "reference log." This is a file that contains all of the output from your game using the input script. Use the `-i` *file* option to tell the run-time system to read commands from your script file, specifying the name of the script file you generated with the `-o` option above. Also specify the `-l` *logfile* option; this will redirect all output from your game to the log file. Save this log file; this is your reference log, which is the output of the game that you consider to be correct. For example:

```
tr -i walkthru.in -l walkthru.log mygame
```

In the future, when you make changes to your game, you can check that you haven't done anything to change the game in unexpected ways. Run your game just as above, but use a different file for the new log file:

```
tr -i walkthru.in -l new.log mygame
```

Now, using a "diff" utililty (a program that compares two text files and displays differences), you can easily find changes you have made to your game's behavior:

```
diff walkthru.log new.log
```

(The "diff" utility is *not* something that is provided with TADS, but good public-domain file difference utilities are available for most systems. In addition, many commercial programming language systems provide some sort of diff utility, and some of the better programmer's text editors have this capability built-in.) Inspect the differences between the new and old logs, and make sure that all of the changes are desirable. Once you're satisfied that the new log is correct, you can simply delete the original reference log, and rename the new log as the new reference log.

Incidentally, we use this facility to test TADS itself. Using a number of games, such as *Deep Space Drifter*, we run a series of input scripts through each new version of TADS, verifying that we haven't done anything to TADS that changes its behavior with existing games.

Making a Self-Loading Game

When your game is completely finished, and you want to distribute it to friends (or even make it widely available through computer networks), you may wish to package your game as a "self-loading" executable. This means that you combine your `.GAM` file (the binary compiled version of your game program) and the TADS Run-Time module into a single executable file.

Making your game into a self-loading executable has some advantages and some disadvantages. The disadvantages are that your game will be larger and less portable if you package it as an executable. If you distribute only the `.GAM` file, you save about 160,000 bytes, which could be important if you'll be uploading your game to BBS's or commercial on-line systems; you can simply tell users that the TADS run-time module is necessary to run your game, and provide directions on where they can find TADS. If players of your game play other TADS games as well, they'll only need to download the TADS run-time once, rather than along with every game. In addition, since the `.GAM` file is portable between operating systems, you only need to make one version of your `.GAM` file available, and anyone with TADS on any system can play your game.

On the other hand, combining the TADS run-time and your `.GAM` file into a single executable makes your game much simpler to install and play, and gives it a more professional appearance. A self-loading executable game is much easier for players, since only a single file is needed, and players won't need to look for TADS (or even know anything about it) before playing your game. In addition, the command to play your game is simply the name of your game.

If you do choose to create a self-loading game, the actual process varies by operating system. You need the TADS run-time itself, plus a special program called `MAKETRX` that combines the run-time and your binary game program. On most command-line systems, such as MS-DOS, you build your game with a command like this:

```
maketrx tr.exe mygame.gam mygame.exe
```

In this command, you tell MAKETRX three things. First, tr.exe specifies the name and location of the TADS run-time executable. Second, mygame.gam is your compiled game. Both of these files must already exist; if they're on a different disk, or in a different directory, simply specify their locations with normal file syntax. Finally, mygame.exe is the name of the self-loading version of your game that you wish to create. The MAKETRX program combines tr.exe and mygame.gam, creating a self-loading executable game in mygame.exe. To run your game, you simply type "mygame" at the DOS prompt.

On Macintosh systems, the process is very similar, except that you use the standard file selectors to select the three files. In addition, you must specify an "owner ID," which is a special code that the operating system uses to associate your executable game with saved games that it creates. You can choose just about any four-letter sequence for the owner ID; it's supposed to be unique among all applications (that is, you should choose a four-letter code that no other application uses as its owner ID), although there is no way to be sure that some other application somewhere isn't using the same ID. Once you create your self-loading game, simply double-click on its icon (from the Finder) to play your game.

Mind in its purest play is like some bat
That beats about in caverns all alone.
Contriving by a kind of senseless wit
Not to conclude against a wall of stone.

— RICHARD PURDY WILBUR, *Mind* (1956)

Appendix E

Errors

This chapter describes how the compiler and run-time system handle errors. A list of errors is provided.

Errors and Error Recovery

When the TADS Compiler detects a syntax or other error in your source file, it attempts to skip ahead to the end of the offending program section. Unfortunately, because the program has a problem, TADS sometimes becomes confused and issues several spurious error messages that are all really due to the original error.

When compiling code, such as inside a method, TADS will try to skip the statement that caused the error; it simply looks for the semicolon that ends the statement. This strategy is most likely to backfire when the error was due to a missing semicolon in the first place.

When compiling an object definition outside of code, TADS will skip the entire object that caused the error. Again, it skips ahead to the next semicolon. If program code intervenes, this often results in more errors; these errors are generally of the form "Expected object or function definition." You can normally ignore such errors when several of them occur consecutively, and look only at the first error which started the trouble, since the subsequent errors may be part of the compiler's attempts to recover from the original error. Once an object or function definition does show up, TADS will normally stop complaining.

Classes of Errors

TADS has four classes of errors. TADS recovers from errors based on their class.

A **fatal error** is one from which the compiler is totally unable to recover; these are generally due to missing files or lack of memory. TADS cannot continue operating after a fatal error, so it terminates.

A **compile-time error** results when the compiler detects a problem with your source code, such as a syntax error, or an undefined symbol. TADS attempts to recover from these errors by skipping the section of code that had the error. When a compile-time error is found in your source code, even though the compiler will continue to process your game, you won't be able to run the game until you have fixed the error.

A **warning** is generated when the compiler detects a questionable construct and wishes to bring it to your attention. Warnings do not otherwise affect compilation; they are produced purely for your information. You should carefully check each warning message to ensure that it's not justified before choosing to ignore it.

A **run-time error** is produced when your program, while it's running, does something illegal, such as adding two objects together; these errors cannot be detected during compilation, because they depend on the state of your program as it runs. If you have compiled your program with source-level debugging information, TADS will automatically produce a symbolic trace showing the function or method that was executing when the error occurred, as well as the function or method that called it, and so forth. When debugging information is not present, TADS will only be able to show the error message, and won't be able to provide any information on where it occurred. After a run-time error has occurred, TADS will attempt to continue executing your game by dropping what it was doing when the error occurred and returning to the player command line.

Error Listing

This section lists the errors that TADS generates. In addition, we explain the conditions that give rise to the errors, and what steps to take to fix them.

To make it easier for you to find information on an error in this book, all TADS errors and warning messages are numbered. When TADS displays an error, it will preface the message with a code such as "TADS-382"; you can use the number in this error code to find the description of the error in the numerically sorted listing in this section.

TADS-1: out of memory

The program has run out of memory. Even though TADS uses a virtual memory subsystem that allows your game to exceed available memory by swapping portions of memory to a disk file, TADS allocates a certain amount of memory that cannot be swapped to disk. As TADS executes, it extends the size of the virtual object cache, and also allocates fixed (unswapped) memory. It is possible for the cache to grow so large that additional fixed memory cannot be allocated; when this happens, TADS runs out of memory and can't continue. When this error occurs, the TADS compiler will display a message indicating the size of the virtual object cache; you should re-run the compiler with a smaller setting for the `-m` (cache size limit) option. See appendix C of this manual for details.

TADS-2: error seeking in file
: An operating system error occurred seeking in a file. This usually indicates an internal problem with TADS.

TADS-3: error reading from file
: An operating system error occurred reading from a file. This usually indicates an internal problem with TADS.

TADS-4: no more page slots
: The virtual memory subsystem does not have any more space for allocating internal tables to track virtual memory objects. The internal limits are high enough that this error should never occur.

TADS-5: attempting to reallocate a locked object
: This error indicates an internal problem with TADS.

TADS-6: swapfile limit reached - out of virtual memory
: The limit you specified with the `-ts` option for the size of the swap file has been reached. You should raise or remove this limit.

TADS-7: error writing file
: An operating system error has occurred writing a file. This usually indicates that the disk is full. You should try to make more space available by deleting files if possible, or by using another disk with more space available.

TADS-8: exceeded swap page table limit
: The compiler has reached its internal limit for memory to track objects swapped out to disk. This limit is high enough that this error should never occur.

TADS-9: requested client object number already in use
: This error indicates an internal problem with TADS.

TADS-10: client mapping table is full
: The compiler has reached an internal limit. This limit is high enough that this error should never occur.

TADS-11: no memory, even after swapping/garbage collection
: The virtual object cache is full or too fragmented to allocate needed memory. The most likely problem is that the cache is too small; increasing the size of the cache (specified with the `-m` compiler and run-time options) should alleviate this problem.

TADS-12: no memory to resize (expand) an object
: The virtual object cache is full. See error 11 above.

TADS-13: unable to open swap file
: The swap file cannot be created. This could mean that the filename is invalid, or the indicated device (volume) or directory does not exist, or that the disk is full or write-protected.

TADS-14: can't get a new object header
The virtual memory subsystem has run out of memory. See error 11 above.

TADS-15: mcm cannot find object to load (internal error)
This error indicates an internal problem with TADS.

TADS-16: attempting to free a locked object (internal error)
This error indicates an internal problem with TADS.

TADS-100: invalid token
Your source file contains a character that TADS does not recognize.

TADS-101: end of file while scanning string
Your source file ends within a quoted string. This usually indicates that you left out the closing quote of a string somewhere in the file.

TADS-102: symbol too long - truncated to "*xxx*"
The indicated symbol is too long. Processing can continue, but TADS will only pay attention to the truncated version.

TADS-103: no space in local symbol table
The local symbol table is full. You can increase the amount of space available for local symbols with the `-ml` compiler option; see appendix C.

TADS-104: invalid preprocessor (#) directive
You have specified an invalid preprocessor directive. When a pound sign (`#`) occurs in the first column of your source file, the compiler expects a valid preprocessing directive (such as `include`) to follow.

TADS-105: no filename in #include directive
You have specified a `#include` directive with no filename. The filename must occur on the same line as the `#include` directive, and must be enclosed in double quotes or angle brackets.

TADS-106: invalid #include syntax
The `#include` directive must be followed on the same line by a filename enclosed in double quotes or angle brackets.

TADS-107: can't find included file "*xxx*"
The compiler could not open the specified include file. Check to make sure that the file is in one of the directories specified with the `-i` compiler options (see appendix C).

TADS-108: no matching delimiter in #include filename
The filename of a `#include` directive was not properly terminated with a double quote or right angle bracket.

TADS-109: out of space for symbol table expansion
The symbol table is too large. Since the symbol table is stored in the virtual object cache, this generally indicates that the compiler is out of cache memory.

TADS-110: input line is too long
Your input file contains a line that is too long. The compiler can handle extremely long lines (up to about 16,000 characters). Break the long line into two or more lines.

TADS-111: warning: file "*xxx*" already included; #include ignored
The specified file has already been included and will not be included again. This warning is most often issued when using pre-compiled header files (see appendix C), and is almost always harmless.

TADS-200: operation is too big for undo log
A single run-time operation has exceeded the size of the undo log. The undo log will be discarded, since it can no longer be used to roll back game state.

TADS-201: no more undo information
No more undo information is available in the undo log. This means that you have applied all undo information, and cannot undo any more turns.

TADS-202: incomplete undo (no previous savepoint)
While attempting to roll back to a savepoint, the undo log ran out of information. This means that a turn may be partially undone.

TADS-300: expected *token*
The compiler requires the specified token. Insert the required token.

TADS-301: expected a symbol
The compiler requires a symbol in this context, and found something else.

TADS-302: expected a property name
The compiler requires a property name in this context, and found something else.

TADS-303: expected an operand
The compiler requires an operand in an expression, and found something else.

TADS-304: expected a comma or closing paren (in arg list)
Your program has a syntax error in an argument list; variables must be separated by commas, and the argument list must be terminated with a right parenthesis.

TADS-305: no space for new parse node
The compiler is out of space parsing an expression. This can sometimes happen with very long strings, especially those that contain embedded expressions. Use the compiler's `-mp` switch to increase the amount of space available for expression parsing.

TADS-306: expected object name
The compiler requires an object name in this context.

TADS-307: redefining symbol as external function
You have declared a symbol as an external function when it already has been declared as another type.

TADS-308: redefining symbol as function
You have declared a symbol as a function when it already has been declared as another type.

TADS-309: can't use "class" with function/external function
You have specified the `class` keyword in a function or external function declaration, which is not allowed (because it makes no sense).

TADS-310: unary operator required
Your expression requires a unary operator (that is, an operator which takes a single operand, such as `-` or `not`).

TADS-311: binary operator required
Your expression requires a binary operator (that is, an operator which takes two operands, such as `*` or `and`).

TADS-312: invalid binary operator
You have specified an invalid binary operator.

TADS-313: invalid assignment
You have used an invalid expression on the left hand side of an assignment operator (`:=`). You can only assign values to local variables, properties, and list elements. It is invalid, for example, to assign a value to an expression such as `(1+1)`.

TADS-314: variable name required
The compiler requires a variable name in this context.

TADS-315: comma or semicolon required in local list
Variables declared in a `local` statement must be separated by commas, and the list must be terminated by a semicolon.

TADS-316: right brace required (eof before end of group)
Your source file is missing the right brace to close a statement block.

TADS-317: 'break' without 'while'
The `break` statement can only occur within a `while`, `for`, `do`, or `case` statement. Your source file contains a `break` statement that is not associated with one of these statements.

TADS-318: 'continue' without 'while'
The `continue` statement can only occur within a `while`, `for`, or `do` statement.

TADS-319: 'else' without 'if'

Your source file contains an `else` that is not associated with an `if` statement. Note that TADS version 1 ignored a semicolon between a closing brace and an `else` keyword; in version 2, the semicolon is considered a null statement which terminates the `if` statement. If your program compiled correctly with TADS version 1 but encounteres error 319 with TADS version 2, you should specify the `-1e` compiler option, which will cause the compiler to adopt the behavior of version 1 with respect to this construct. See appendix C for details.

TADS-320: warning: possible use of '=' where ':=' intended

The compiler has detected a statement which it suspects is an assignment that has been accidentally miscoded with "`=`" (which is the relational equality comparison operator) instead of "`:=`" (the assignment operator). The compiler is almost always right in these situations; you should check the statement to ensure that you coded what you really meant.

TADS-321: unexpected end of file

The compiler has encountered the end of your source file when it thought it had more work to do. This could mean that you have left off the semicolon terminating an object definition, or a right brace terminating a function.

TADS-322: general syntax error

The compiler has detected an ill-formed statement.

TADS-323: invalid operand type

One of the operands in an expression is not suitable for the operator. This usually means that a constant in the expression is of the wrong type.

TADS-324: can't expand local symbol table

The compiler cannot allocate space for a new local symbol table. You should specify a larger value with the compiler's `-ml` option (see appendix C).

TADS-325: can't expand argument symbol table

The compiler cannot allocate space for a new argument symbol table. You should specify a larger value with the compiler's `-ml` option (see appendix C).

TADS-326: redefining a function which is already defined

You are attempting to define a function twice.

TADS-327: 'case' or 'default' not in switch block

You have coded a `case` or `default` label outside of a `switch` statement.

TADS-328: constant required in switch case value

The values in `case` labels must be constants. You have specified a non-constant expression as a `case` label.

TADS-329: label required for 'goto'

You have omitted the label of a `goto` statement, or the symbol specified is something other than a label.

TADS-330: 'goto' label never defined
A label used in a `goto` statement is not defined within the function or method in which the `goto` statement occurred. The `goto` statement can only be used with labels within the same function or method as the `goto` statement itself.

TADS-331: too many superclasses for object
You have specified too many superclasses in an object definition.

TADS-332: redefining symbol as object
You are attempting to define a symbol as an object, but you have already defined the symbol as something else.

TADS-333: property being redefined in object
You are attempting to define a symbol as an object, but the symbol has already been used as a property.

TADS-334: invalid property value
The value specified for a property's initial value in an object definition is not valid.

TADS-335: invalid vocabulary property value
The value specified for a vocabulary property is invalid. You can only use single-quoted strings for vocabulary properties.

TADS-336: invalid template property value (need string)
The value specified for a template is invalid. You can only use a single-quoted string for a template property.

TADS-337: template base property name too long
The base name of a template property value is too long. Since these values must be appended to the strings `do`, `verDo`, `io`, and `verIo`, the resulting symbols will be up to five characters longer than the property value itself.

TADS-338: too many templates (internal compiler limit)
You have defined an object with too many templates.

TADS-339: invalid value for compound word (string required)
You have specified something other than a single-quoted string in a `compoundWord` definition.

TADS-340: invalid value for format string (string required)
You have specified something other than a single-quoted string in a `formatString` definition.

TADS-341: invalid value for synonym (string required)
You have specified something other than a single-quoted string in a synonym definition.

TADS-342: undefined symbol
You have referenced a symbol which has not been defined.

TADS-343: invalid value for specialWords list (string required)
You have specified something other than a single-quoted string in a `specialWords` list.

TADS-400: object cannot grow any bigger - code too big
The object is too large. This is unlikely to occur, since the limit on the size of an object is very high; however, if it does, you can break the object up into two or more objects, with the first object inheriting from the second.

TADS-401: no more temporary labels/fixups (internal compiler limit)
The compiler has run out of space during code generation. You should reduce the complexity of your code (for example, reduce the nesting of loops and conditional statements).

TADS-402: (internal error) label never set
This error indicates an internal problem with TADS.

TADS-403: invalid datatype for list element
A list element is of an invalid type.

TADS-404: too many debugger source line records (internal limit)
Your source program has generated too many debugger source line records. This is an internal TADS limit; the limit is high enough that this error should never occur.

TADS-450: vocabulary being redefined for object
You have defined a vocabulary word more than once for a particular object.

TADS-500: location of object "*xxx*" is not an object
This is a warning. In most cases, you want an object's `location` property to refer directly to another object, since this allows the compiler to generate a matching `contents` property automatically. Under certain conditions, you may intentionally use a method for a `location` value. When you have intentionally coded something other than an object for a `location` value, you can add the property `locationOK = true` to the object, which will suppress this warning message.

TADS-501: contents of object "*xxx*" is not list
You should almost never code a `contents` property for an object, since the compiler automatically sets up these properties.

TADS-502: overflow trying to build contents list
A `contents` property list has grown too long. This is an internal limit; move some of the contents of the object somewhere else.

TADS-503: required object "*xxx*" not found
The specified object has not been defined in your game, but the TADS compiler requires you to define it.

TADS-504: warning - object "*xxx*" not found
The specified object has not been defined by your game. This is not an error, because the object is not required by TADS; however, TADS generates this warning in case you thought you had declared this object.

TADS-505: too many built-in functions (internal error)
This indicates an internal problem with TADS.

TADS-600: unable to open game for writing
The game binary file could not be created. This could mean that the name is invalid, or the device (volume) or directory is invalid or does not exist, or that the disk is full or write-protected.

TADS-601: error writing to game file
An operating system error occurred writing to the game file. This usually means that the disk is full.

TADS-602: too many sc's for writing in fiowrt
This indicates an internal problem with TADS.

TADS-603: undefined function "*xxx*"
The specified function was called or declared, but never defined.

TADS-604: undefined object "*xxx*"
The specified object was referenced, but never defined.

TADS-605: undefined symbols found
One or more undefined symbols was found in your program.

TADS-606: unable to open game for reading
The game binary file couldn't be opened. This could mean that the file does not exist, or that the filename is invalid, or that the device (volume) or directory is invalid or does not exist.

TADS-607: error reading game file
An operating system error has occurred reading the game file. This could mean that the file has been corrupted.

TADS-608: file has invalid header - not a TADS game file
The specified binary game file is not a TADS game file. This could mean that you have specified the incorrect file, or that the file has become corrupted.

TADS-609: unknown resource type in .gam file
The game file contains invalid data. This could mean that the file has become corrupted.

TADS-610: unknown object type in OBJ resource
The game file contains invalid data. This could mean that the file has become corrupted.

TADS-611: file saved by different (incompatible) version
The game file or save file was created by an incompatible version of TADS or of the game. You should either use the version of the TADS run-time that matches the version used to compile the game, or you should recompile the game with a compiler that matches your run-time.

TADS-612: error loading object on demand
An object could not be loaded. This could mean that the game file is corrupted, or that the disk file has become unavailable (for example, you have removed the floppy on which the game file was stored from the disk drive).

TADS-613: object too big for load region (internal error)
This indicates an internal problem with TADS.

TADS-614: did not expect external function
This indicates an internal problem with TADS.

TADS-990: user requested cancel of current operation
You have cancelled compilation of your game, such as by hitting the "cancel" button on the Macintosh compiler.

TADS-1001: stack overflow
Run-time stack space has been exceeded. This usually means that your program has gone into infinite recursion—that is, a function is calling itself repeatedly. If the program appears to be behaving correctly, you can increase the size of the run-time stack with the `-ms` run-time parameter (see appendix D).

TADS-1002: heap overflow
Run-time heap space has been exceeded. You can increase the size of the run-time heap with the `-mh` option (see appendix D).

TADS-1003: numeric value required
A run-time operation that requires a numeric value was invoked with some other type of value.

TADS-1004: stack underflow
Too many elements have been removed from the run-time stack. This usually indicates that you have called a function with an invalid number of arguments.

TADS-1005: logical value required
A run-time operation that requires a logical value `true` or `nil` was invoked with some other type of value.

TADS-1006: invalid datatypes for magnitude comparison
A magnitude comparision operator (`>`, `<`, `>=`, or `<=`) was called with something other than a number or string.

TADS-1007: string value required
A run-time operation that requires a string value was invoked with some other type of value.

TADS-1008: invalid datatypes for binary '+' operator
The binary "+" operator works only on strings, lists, and numbers.

TADS-1009: invalid datatypes for binary '-' operator
The binary "-" operator works only on lists and numbers.

TADS-1010: object value required
A run-time operation that requires an object value was invoked with some other type of value.

TADS-1011: function pointer required
A run-time operation that requires a function pointer value was invoked with some other type of value.

TADS-1012: property pointer value required
A run-time operation that requires a property pointer value was invoked with some other type of value.

TADS-1013: 'exit' statement executed
If this error occurs, it indicates an internal problem with TADS.

TADS-1014: 'abort' statement executed
If this error occurs, it indicates an internal problem with TADS.

TADS-1015: 'askdo' statement executed
If this error occurs, it indicates an internal problem with TADS.

TADS-1016: 'askio' executed; preposition is object #*n*
If this error occurs, it indicates an internal problem with TADS.

TADS-1017: 'quit' executed
If this error occurs, it indicates an internal problem with TADS.

TADS-1018: 'reset' executed
If this error occurs, it indicates an internal problem with TADS.

TADS-1020: list value required
A run-time operation that requires a list value was invoked with some other type of value.

TADS-1021: index value too low (must be >= 1)
The value of a list index operation was below 1.

TADS-1022: index value too high (must be <= length(list))
The value of a list index operation was higher than the number of elements in the list.

TADS-1023: invalid type for built-in function
Your program called a built-in function with the incorrect type of arguments.

TADS-1024: invalid value for built-in function "*xxx*"
Your program called a built-in function with the incorrect type of arguments.

TADS-1025: wrong number of arguments to built-in
Your program called a built-in function with the incorrect number of arguments.

TADS-1026: wrong number of arguments to user function
Your program called one of its own functions or methods with the incorrect number of arguments.

TADS-1027: string/list not allowed for fuse/daemon arg
The parameter to a fuse or daemon (specified in a `setdaemon()` or `setfuse()` call) cannot be a string or a list.

TADS-1028: internal error in setfuse/setdaemon/notify
This indicates an internal problem with TADS.

TADS-1029: too many fuses
You have attempted to register too many fuses simultaneously.

TADS-1030: too many daemons
You have attempted to register too many daemons simultaneously.

TADS-1031: too many notifiers
You have attempted to register too many notifiers simultaneously.

TADS-1032: fuse not found in remfuse
You have tried to remove a fuse that is not active.

TADS-1033: daemon not found in remdaemon
You have tried to remove a daemon that is not active.

TADS-1034: notifier not found in unnotify
You have tried to remove a notifier that is not active.

TADS-1035: internal error in remfuse/remdaemon/unnotify
This indicates an internal problem with TADS.

TADS-1036: load-on-demand loop: property not being set (internal)
This indicates an internal problem with TADS.

TADS-1037: undefined object in vocabulary tree
This indicates that a superclass object has not been defined.

TADS-1038: c-string conversion overflows buffer (internal limit)
This indicates an internal problem with TADS.

TADS-1039: invalid opcode (internal error)
This indicates an internal problem with TADS, or a corrupted game file.

TADS-1040: property evaluated for non-existent object
You have attempted to take a property of an object which has not been defined. This probably means that you are attempting to execute a game with compiler errors.

TADS-1041: unable to load external function "*xxx*"
The specified external function (user exit) is not available, because it has not been added to the game with the proper resource compiler (or other appropriate operating system-dependent mechanism). See appendix F for information on writing external functions.

TADS-1042: error executing external function "*xxx*"
The specified external function cannot be executed. See appendix F for information on writing external functions.

TADS-1043: circular synonym
You have declared a synonym which refers to another synonym which refers to the original synonym.

TADS-1500: invalid command-line usage
You have specified invalid command line arguments. This error will always be accompanied with an informational message describing the proper command line arguments.

TADS-1501: error opening input file
The debugger could not open the specified input file.

TADS-1502: game not compiled for debugging - use -ds option
You must use the `-ds` option when compiling your game if you wish to use it with the debugger.

TADS-2000: error setting breakpoint: unknown symbol
You have attempted to set a breakpoint on a non-existent symbol.

TADS-2001: I'm afraid I can't do that, Dave
The pod bay doors could not be opened. Check the AE-35 unit for a possible malfunction.

TADS-2002: error setting breakpoint: symbol is not a property
The specified symbol is not a property.

TADS-2003: error setting breakpoint: symbol is not a function
The specified symbol is not a function.

TADS-2004: error setting breakpoint: property not defined in object
The specified property is not defined in the specified object.

TADS-2005: error setting breakpoint: property is not code
The specified property is not code, but some other type of value. Breakpoints can only be set in executable code.

TADS-2006: error: breakpoint is already set at this location
You have attempted to set more than one breakpoint at the same location.

TADS-2007: error setting breakpoint: breakpoint not at line
You have attempted to set a breakpoint somewhere other than at a code location associated with a source line.

TADS-2008: too many breakpoints
The debugger cannot set any more breakpoints. You must remove one or more breakpoints before additional breakpoints can be set.

TADS-2009: breakpoint was not set
The breakpoint could not be set as requested.

TADS-2010: too many symbols in eval expression (internal limit)
You have specified an excessively complex expression. Simplify the expression.

TADS-2011: unable to find source file "*xxx*"
The specified source file cannot be opened.

TADS-2012: assignment to local is illegal in watch expression
You cannot make an assignment to a local variable within a watch expression.

TADS-2013: inactive frame (expression value not available)
The specified frame can not be accessed because it is no longer active. The value of the expression can not be obtained.

TADS-2014: too many watch expressions
You must remove one or more watch expressions before setting additional watch expressions.

TADS-2015: watch expression not set
The requested watch expression could not be set.

TADS-2016: extraneous text after end of command
Your command has extra text past the end that the debugger did not understand.

TADS-2017: error setting breakpoint: symbol is not an object
The specified symbol is not an object.

A book may be very amusing with numerous errors,
or it may be very dull without a single absurdity.
— OLIVER GOLDSMITH, *The Vicar of Wakefield* (1766)

Appendix F

External Functions

TADS has a facility that allows you to call a function written in another language, such as C or Assembler, from within your TADS game program. We refer to these routines written in other languages as "external functions" or "user exits." This appendix describes how to write and invoke external functions.

You may wish to be able to invoke a function written in another language, such as C or Assembler, from within your game program. For example, you may want to write a function that reads some information from a disk file, or you may want to program the sound hardware on your computer. TADS allows you to do this through a mechanism known as "user exits" or "external functions."

Before going further, we should warn you that writing a user exit is somewhat complicated. We've tried to make it as easy as possible to write an external function, but you will have to learn a number of subtle details in order to use this mechanism. Furthermore, we don't have the space in this appendix (or the pedagogical facility) to provide any information about programming in C or Assembler or any other language, or about using your compiler or linker or other programming tools. So, we'll assume that you already know how to program in C, and that you're already familiar with your programming tools.

Declaring and Invoking a User Exit

As far as your TADS program is concerned, an external function is nearly the same as a built-in function. The only difference is that you must tell the TADS compiler that the function is a user exit; you do this by using the special `external function` declaration. This declaration is similar to a forward declaration for a normal function; for example, if you wish to define a user exit named `myfunc`, you would include a declaration like this in your TADS program:

```
myfunc:  external function;
```

Once you've declared `myfunc` as an external function, you can call it just as you would call any built-in or user-defined function. For example, you could include code like this in your TADS program:

```
x := myfunc(10, 'hello');
```

The Sample User Exit

Invoking a user exit is easy; writing a user exit is a little more complicated. The rest of this appendix describes how to write a user exit. Note that the TADS software contains a sample user exit written in C, as well as a small TADS game program that invokes it; you will probably find it helpful to look at the example to get a better idea of how the external function mechanism works. The user exit itself is contained in the file `TESTUX.C`; the TADS game program that invokes the user exit is in the file `TESTUX.T`. You may also want to consult `TADSEXIT.H`, which is a file that you should include in your C user exits; this file defines the interface between TADS and your external function with easy-to-use macros.

Note that the process of compiling and linking an external function varies by operating system. You should refer to `TESTUX.C`, which contains information about compiling and linking a user exit for each operating system. Note that you can write user exits in any language that will support a C-style interface; you may wish to use assembly language in particular if your user exit must access your computer's hardware. However, we only provide the `TADSEXIT.H` definitions file for the C language, and the instructions in `TESTUX.C` are specific to C. If you want to use another language, you'll have to be familiar enough with your compiler and operating system to be able to apply the instructions we've provided to your language.

Writing the External Function

In general, you write your TADS user exit as a regular C function. You must not use any static or global variables, but you can use variables on the stack. You can't call any of the C run-time library functions (such as printf) that may use global variables. If your function or any function it calls uses global variables, the results will be unpredictable, and you may crash the machine.

(Note that the prohibition of static and global variables is a simplification. In actuality, you can use globals on most systems under specific conditions. On MS-DOS, Atari ST, and Macintosh, you can use global variables as long as they are PC-relative—that is, they must be contained in the same program section as the object code of your user exit and must be addressed using the program counter. If you are familiar enough with your compiler to know how to generate PC-relative static variables, which most compilers on these systems can do when presented with the correct combination of secret options, you should be able to use static variables without any harm. Exactly how you go about doing this with your compiler is, obviously, beyond the scope of this appendix.)

The first thing in the C program that contains your user exit should be a `#include` directive like this:

```
#include <tadsexit.h>
```

This inserts the header file `tadsexit.h` (which is part of the TADS software) into the current compilation unit. This file defines the interface between TADS and your user exit program with a set of easy-to-use macros. Using these macros, you don't need to be aware of the strange details of the data structures and calling sequences that TADS uses to call your user exit and vice versa.

Whatever you call your user exit with the `external function` declaration in your TADS program, the user exit in your C program is called `main`. You declare it like this:

```
int main(ctx)
tadsuxdef far *ctx;
{
```

You may have two worries at this point. First, if the C program calls the function `main`, how does TADS know to call it when you call `myexit` from your game program? Second, if you want to call more than one user exit from your game program, how can they both be named `main`?

The answer to both questions is that TADS learns the name of your user exit not from the C compiler (which only knows the function as `main`), but from a "resource editor." On MS-DOS and Atari ST systems, this resource editor is called TADSRSC, and is provided with the TADS software; refer to the documentation in the file `TESTUX.C` for information on how to use TADSRSC. On Macintosh systems, you should use a standard resource editor, such as ResEdit, which should be packaged with your C programming tools.

In any case, each user exit is compiled separately, which means that you can have as many as you want—all named `main`—without your C compiler or TADS becoming confused about all the identical names. Using the resource editor, you add each separately compiled user exit to your game program, assigning the resource the name that you use in your `external function` declaration. The important thing is that the resource name defined with the resource editor matches the name declared in the `external function` statement.

The argument to the function `main` is not the argument that you provided in the invocation of the user exit in your TADS program. Instead, it is a "context," which you use for communications with TADS. You use this context argument to obtain the actual arguments to your user exit function.

All of your arguments are on a "stack," which means that you can get to them one at a time. First, you call the function `tads_tostyp(ctx)` to learn the *type* of the first argument. This returns one of these values (defined in `TADSEXIT.H`): `TADS_NUMBER`, indicating the argument is a long integer; `TADS_STRING`, indicating the argument is a (single-quoted) string; `TADS_NIL` or `TADS_TRUE`, indicating one of the truth values. You cannot pass other types of values to user exits, because the user exits wouldn't know what to do with other types.

Next, you retrieve the first argument from the stack by calling the function appropriate to its type: `tads_popnum(ctx)` for numbers, `tads_popstr(ctx)` for strings, or just `tads_pop(ctx)` for truth values. This last function has no return value, because all you need to know is the type (true or nil), but you still must call it to remove the argument from the stack; the other two functions return either a long integer or a string descriptor, respectively.

The return value of the `tads_popstr(ctx)` function is a "string descriptor," which is *not* a string pointer in the usual C sense. Instead, it is a special data structure that contains the length of the

string and the text of the string. TADS version 2 does not use C-style null-terminated strings, but rather keeps the length of each string along with its text value. You can't use the string descriptor directly, but two functions are provided that allow you to decompose it into values you can use. Call `tads_strlen(ctx, str)`, where *str* is the descriptor value returned by `tads_popstr()`, to get the length of the string; this function returns an integer giving the number of bytes in the string. Call `tads_strptr(ctx, str)` to get the text of the string; this function returns a character pointer giving the address of the first byte of the string's text. Note that the string's text is *not* null-terminated—you must use the value from `tads_strlen()` to determine the length of the text buffer. Be careful not to use any of the C functions that require a null-terminated string with the text buffer, because the buffer does not have a terminating null byte.

You should use the special type `tads_strdesc` to hold the return value of the `tads_strpop()` function. This type is defined in `TADSEXIT.H` along with the TADS interface functions.

Never write into a text buffer obtained with `tads_strptr()`. TADS assumes that you will leave these text buffers unchanged; modifying an argument string's text buffer will have unpredictable results.

After you have retrieved the first argument, you proceed with other arguments in the same manner. You *must* retrieve all arguments from the stack with one of the `tads_pop`*xxx*`(ctx)` functions, and you *must not* retrieve any more than you received. You can find out how many arguments your function actually received by calling the function `tads_argc(ctx)`; this function returns an integer telling you the actual number of arguments. (This function returns the actual number of arguments regardless of whether you have retrieved any arguments; it does *not* change as you pop arguments off the stack.)

If your user exit returns a value, you use one of the `tads_push`*xxx*`(ctx, value)` functions. To return a number, call `tads_pushnum(ctx, value)`, where *value* is a long integer. To return a truth value, call `tads_pushtrue(ctx)` or `tads_pushnil(ctx)`. To return a C-style string (that is, a null-terminated string that you built on the C stack, using a local character array variable), use `tads_pushcstr(ctx, pointer)`, where *pointer* is the address of the first byte of the string.

You can also ask TADS to allocate space for a string, and build your return value using this space. Call `tads_stralo(ctx, len)`, where *len* is an integer specifying the number of bytes in the string you wish to return. This function returns a pointer to the first byte of the allocated space. Write your string into this space. The string must *not* be null-terminated; instead, use *exactly* the number of bytes that you allocated. Once you have finished building your string, call `tads_pushastr(ctx, pointer)` to push the return value, where *pointer* is the value originally returned by `tads_stralo()`. TADS remembers the length of buffer you allocated, so there is no need to specify the length when returning the string.

Skilled in the works of both languages.
— HORACE, *Satires* (23 B. C.)

Appendix G

TADS Debugger

TADS features a source-level debugger that can be helpful in making your game work properly. The debugger makes it possible to watch your game program's execution line by line, and examine the state of variables and properties. This chapter describes the debugger, and how to use it to get your game working.

What is a Debugger?

The term "debugger" isn't exactly accurate for this type of program, because it doesn't actually describe what the program is so much as what you normally use it for. Unfortunately, a debugger doesn't actually find and remove problems from your program, as the name suggests; instead, it provides a set of tools that makes it easier for you, the programmer, to find and remove the bugs in your program. A better term would be "debugging tool"; you're the real debugger.

What the debugger actually does is to let you examine your game program in detail while it executes. Using the debugger, you can step through your game line by line, which allows you to see exactly what your program does as it runs. You can also set "breakpoints," which cause the game's execution to be suspended at certain lines of code that you specify; you can use breakpoints to determine when and why certain lines of code are executed (or not executed). You can also examine the state of your game, by displaying the value of a variable or property. You can even change the value of a variable or property.

If you're just learning TADS, you may find that the debugger can help you understand how the system works. Using TDB, you can step through your game line by line, so you can see how the parser calls routines in your program, how inheritance works, how the functions and objects in `adv.t` work, and other details that can be confusing at first.

Using the Debugger

The rest of this chapter describes how to use the debugger. Note that some of the information in this chapter depends on the type of computer you are using, because the user interface to the debugger is different on each computer. So, the mechanism for invoking a debugger feature is different on some computers than described here; for example, a breakpoint may be set by typing a command on some computers, by pressing a function key on other machines, and by selecting an item from a menu on other systems.

Getting Ready for Debugging

Before you start using the debugger, you must first compile your game program with the special option that enables the generation of source-level debugging information in the binary (`.GAM`) file that the compiler produces. On most systems, this means that you must include the `-ds` option on the compiler command line, along with any other options that you normally use.

Other than including the `-ds` option on the compiler command line, compiling with debugging information is exactly the same as compiling without the extra information. The compiler still produces a `.GAM` file as usual, and you can still use this `.GAM` file with the TADS run-time to play the game. However, the `.GAM` file will be somewhat larger when you use the `-ds` option, because the compiler includes extra information that the debugger uses to identify the source code and variables within your game. (The fact that the `.GAM` file is larger when you include debugging information is the only reason not to use the `-ds` option at all times. When you're ready to give your game to other people, you probably will want to recompile without the `-ds` option so that the `.GAM` file is smaller.)

Starting the Debugger

On most systems, you start the debugger by typing the `tdb` command. This command is very similar to the `tr` command that you use to run your game normally. In fact, most of the same options can be used with the debugger, in addition to a couple of special debugger options. With the exception of the `-i`, `-o`, and `-l` options, which are all used for the run-time's script processing feature, the debugger accepts all run-time options.

In addition, the debugger accepts the `-mp` *size* option, which has the same meaning as with the compiler: set the parse node pool to *size* bytes. The reason that this option is used by the debugger as well as the compiler is that the debugger uses the same subsystem as the compiler to parse expressions. This means that you can use the same types of expressions that you use in your source code when typing certain commands to the debugger.

The debugger accepts another special option: `-i` *path*. This option is very similar to the `-i` *path* option accepted by the compiler. With the debugger, this option indicates a directory that should

be searched for source code. The only difference between the `-i` option of the debugger and that of the compiler is that the debugger will search each directory specified with a `-i` option for the main source file as well as header (`#include`) files. Normally, you will use the same set of `-i` options with the debugger that you use with the compiler.

Here's an example debugger command line. This command starts the debugger and loads the binary game file `MYGAME.GAM`. The debugger will search the directory `c:\tads` for source files that it doesn't find in the current directory.

```
tdb -i c:\tads mygame
```

Note that the debugger looks for a configuration file named `CONFIG.TDB`, in the same way that the compiler uses `CONFIG.TC`. Refer to Appendix C, which describes the TADS Compiler, for information about using configuration files. The debugger uses the same method as the compiler to find and interpret the configuration file.

Once you're running the debugger, you can get help at any time by typing "`?`" or "`help`". On MS-DOS, you can also press the function key F1. The debugger will display a brief list of commands (and function keys, if appropriate for your operating system) in the source window.

Screen Layout

The screen layout used by the debugger varies by system, but each system shows roughly the same information. On some machines (such as MS-DOS PC's), the debugger actually uses two different "screens" to display information. These aren't actually two physical monitors, but two virtual monitors that you can switch between; the debugger will normally display the active virtual monitor, and you can switch to the other screen by pressing a certain function key or typing a command. On other systems, the debugger uses windows to display the different areas.

On two-screen systems, the first screen is the normal run-time system screen, showing the status line, the output of your game, and the player's commands. The second screen is the debugger screen, which shows source code, variable values, a debugger status line that indicates the current state of execution of your game (such as which source code line will be executed next), and the debugger command and display area. On windowing systems, these areas will be displayed in different windows all on one screen, and you can use your mouse as usual to bring different windows to the foreground.

Controlling Execution

The debugger gives you control over your game's execution. When the debugger first starts, it will load your game and wait for a command. Nothing in your game will have been executed at this point, so the debugger source code area will show your `preinit()` function; the first line of this function will be highlighted, which indicates that it is the next line to be executed.

At this point, your game is "suspended." This means that the debugger has control, not your game. Execution of your game can be suspended in a number of ways: after stepping by one line,

after hitting a breakpoint, immediately after entering the debugger, or after your game calls the `debugTrace()` built-in function. When your game is suspended, you can type debugger commands, and you can return control to your game at any time.

To start your game running when it's suspended, you can use the `g` (for "go") command. This command resumes execution of your game. After typing `g`, your game will execute until something causes it to be suspended, such as a breakpoint or a call to the `debugTrace()` function.

You can also step through a single line of your game's source code by using the `t` (for "trace") command. This command simply executes the current line and then suspends execution again; if the current line calls another method or function, the debugger will step into that function, and suspend your game before executing any lines in the new function.

You can also step a single line using the `p` (for "procedure trace") command. This command is similar to the `t` command, in that it executes a single line of code; however, unlike the `t` command, the `p` command steps over, rather than into, functions and methods that are called by the current line of code. That is, if a function or method is called, the debugger will execute the entire called function (and any functions and methods it calls) without stopping. After the `p` command, execution is suspended at the next line of source after the current line.

Breakpoints

You can arrange for execution to be suspended at a particular point in your program by setting a "breakpoint." To set a breakpoint, use the `bp` (for "break point") command: type `bp` followed by the name of a function, or *object.property*, where you want execution to be suspended.

You can set many breakpoints. Use the `bl` ("breakpoint list") command to list the breakpoints that are currently active. Each breakpoint in the list is given a number; you use this number to refer to the breakpoints you have previously set.

To remove a breakpoint, use the `bc` ("breakpoint clear") command. Type `bc` followed by the number (from the `bl` command's output) of the breakpoint you wish to remove. You can also temporarily disable a breakpoint without removing it entirely by using the `bd` ("breakpoint disable") command. When a breakpoint is disabled, the debugger remembers the breakpoint, but doesn't do anything about it; that is, when the breakpoint is hit, execution continues anyway as though no breakpoint were set there. You can re-enable a disabled breakpoint with the `be` ("breakpoint enable") command.

You can also set a breakpoint at a particular line of source code by pointing at the line of code and using a function key or menu selection. On MS-DOS PC's, use the arrow keys to position the cursor at the line where you wish to set a breakpoint, and press the F2 function key; this will highlight the line with a special color to indicate that a breakpoint is set there.

You can also set breakpoints that only suspend execution when a specific condition is met. You do this by adding the keyword `when` followed by an expression to a normal `bp` command; the breakpoint only stops execution when the particular line of source is executed *and* the expression is true. For example, to set a breakpoint when the `sleepDaemon()` function is called, but only when the player is in the room `kitchen`, you would specify this:

```
bp sleepDaemon when Me.location = kitchen
```

You can also set a "global breakpoint"; this is a breakpoint that stops execution when an expression becomes true, regardless of which line in your game is being executed. To set a global breakpoint, simply omit the location; for example, to stop execution when the player moves into the kitchen:

```
bp when Me.location = kitchen
```

Note that the debugger automatically disables a global breakpoint when it is hit. This is because, once the condition becomes true, it will generally remain true for a while; if the breakpoint remained enabled, `g` would be the same as `t`, since the breakpoint would be hit on every line as long as the condition is true. You can re-enable the breakpoint with the `be` command.

Where Am I?

When your game is suspended, you can determine what line in your game is about to be executed, and how you got there, using the `k` ("stacK") command. This command displays the function or method that is currently executing, followed by the function or method that called the current function, followed by the function or method that called it, and so forth. The run-time system's player command parser is always the last thing in this chain (although the debugger doesn't display "player command parser" anywhere, since it's not part of your game program, but part of the run-time system).

Examining Variables

Any time your game is suspended, you can examine the value of any variable or property in your program. Use the `e` ("evaluate") command to examine a value: type `e` followed by an expression. The expression is exactly the same as you would use in a source program. For example, to check the player's current location:

```
e Me.location
```

You can use a local variable in an expression, but only if that local variable is currently active. Naturally, you can't use a local variable from a function that isn't currently executing.

You can look at the value of a local variable from the current function, or even from a caller of the current function (see the `k` command above). Use the `u` ("up") and `d` ("down") commands to select the active function. Whenever your program is suspended, the debugger sets the active function to the current function, so you can look at variables in the current function. To look at a variable in the caller of the current function, use the `u` command to set the active function to the caller of the current function. Use the `u` command to set the active function to the caller of that function. Use the `d` command to go back down the stack chain.

Note that the `k` command displays an asterisk next to the active function. You can use the `k` command if you forget how many times you've typed `u` and `d`. Note also that the `u` and `d` commands

don't have any effect on your program's execution; they simply tell the debugger which function's variables you want to use in expressions.

If you want to change the value of a variable, you can simply evaluate an expression that involves an assignment. For example, to change the value of the local variable `cnt` to 3, you would type this:

```
e cnt := 3
```

You should use assignments with extreme care, because you could accidentally introduce an inconsistentency that confuses your game.

If you evaluate an expression that displays double-quoted text strings, or calls the `say()` built-in function, the text will be displayed in the *game* window. This may cause confusing displays, because the text will simply be added to whatever was already in the game window. Note that, on MS-DOS, you'll have to use the F5 key (or the \ command) to switch to the game screen to see the results of any double-quoted strings displayed by the `e` command.

In addition, since the run-time system formats text displayed in the game window, you may sometimes find that double-quoted text isn't immediately displayed. This is because the run-time system saves up text until it has enough to fill an entire line, then displays the text all at once. If this happens, you can force the debugger to display the text by evaluating a double-quoted string that contains a newline. For example:

```
e "\n"
```

This will force the debugger to display any text that it has been saving.

Watch Expressions

You will often want to see the value of a variable, property, or other expression as it changes while you're stepping through code. You could repeatedly use the `e` command to evaluate the expression, but the debugger makes this more convenient by allowing you to enter "watch expressions." Once you create a watch expression, the debugger will display the current value of the expression in a special area on the screen. The current value will be updated on the screen whenever the expression's value changes. This means that you can monitor a set of expressions without having to type the `e` command repeatedly.

To set a watch expression, use the `ws` ("watch set") command. Type `ws` followed by an expression. The debugger will add the expression to the watch area.

To delete a watch expression, use the `wd` ("watch delete") command. Type `wd` followed by the watch expression number (which is displayed next to the watch expression in the watch window).

Note that watch expressions do *not* display any double-quoted string text, or any text displayed with the `say()` built-in function. If any double-quoted text is displayed in the course of evaluating a watch expression, the text is discarded. In general, it's not a good idea to evaluate expressions with side effects (such as assignments) in watch expressions, because the expressions are evaluated repeatedly, and at times that you can't directly control.

If you set a watch expression that involves a local variable, the expression will only be meaningful when the local variable is active—that is, when the block that contains the local variable is being executed. When execution moves outside of the block that contains the local variable, TDB will not be able to evaluate the expression, and will simply display an error message in place of a value for that watch expression. When execution returns to the block that contains the local variable, TDB will once again be able to display a value for the expression.

Source Window Commands

The debugger provides a couple of commands for displaying other source files in the source window, and for repositioning the source window display.

To list the source files that make up your game, use the `fl` ("file list") command. This command displays a list of the source files in your game, giving each file a number. You can use this number as a shortcut with the `fv` command.

To display a different file in the source window, use the `fv` ("file view") command. Type `fv` followed by either a filename or a file number (this number comes from the output of the `fl` command). The source window will show the newly selected file.

You can search for text within the current source file using the `/` command. Type `/` followed immediately (without any spaces) by the string you want to find. The debugger will position the cursor at the first occurrence of the text within the file, if it was found, or at the end of the file if not. To search again for the same text, type `/` without anything following it. Note that the cursor will be moved to the source window after searching for text.

Miscellaneous Commands

To quit the debugger, use the `quit` command.

You can get help with the debugger's commands and function keys by using the `?` command; the `help` command is equivalent. The debugger will display in the source window a file with a brief description of TDB's commands and function keys.

TDB on MS-DOS

On MS-DOS, the Debugger makes use of screen switching. This means that your monitor is completely filled with either your game's output, or with the debugger screen. While your game is active, it takes up the entire screen; it looks exactly like it does when you run with the normal TADS runtime. When the debugger has control (after a breakpoint has been encountered, for example), the debugger takes up the entire screen.

You can switch the display back to the game screen by pressing F5, or by using the "\" command. This will display the game screen; press any key to return to the debugger screen. Note that you can't type any commands into the game, since it's suspended.

The debugger's screen is separated into three windows. The top window is the source window; this window displays the current source file (although you can also display other source files from your game, or any other text file, in this window; see the **fv** command above). The bottom of the source window is a status line that shows you the function or method where execution is currently suspended, and the name of the file that contains that code.

The bottom window is the command window. You type commands to TDB and see the results in this window. When the debugger is ready for a command, it will show you this prompt:

```
tdb>
```

The cursor will be positioned after the prompt. You can now type a command to the debugger (terminated with the Return or Enter key), and the results will be displayed in the command window.

The middle window is the watch expression window. This window is only displayed when one or more watch expressions are active, so it is not shown when you first enter the debugger. The debugger automatically adjusts the size of this window to make room for the watch expressions you have entered. See the discussion of watch expressions earlier in this chapter for information on creating watch expressions.

Function Keys on MS-DOS

This section lists the special function keys used in the MS-DOS version of the TADS Debugger. Note that some of these keys can be used only in the source window; to move the cursor to the source window from the command window, press the Tab key. Note also that most of the functions performed by these keys can be obtained with a command typed in the command window; however, the function key equivalent is often more convenient.

F2: (Used only when the cursor is in the source window.) Sets or clears the breakpoint at the current line (the line at which the cursor is positioned). This is similar to the `bp` command, except that it allows you to set a break point at any executable line of code, rather than just at the start of a function or method.

Shift-F2: (Used only when the cursor is in the source window.) Sets a breakpoint at the current line, with a conditional expression. The debugger will prompt you for a conditional expression; type an expression and hit the Enter key. This is similar to the `bp` command using the `when` keyword, but allows you to set a conditional breakpoint at any executable line of code.

F5: Show the game screen. When the game is suspended, the game output is replaced on your monitor by the debugger screen; hitting F5 switches back to the game screen. Hit any key to return to the debugger screen. This is the same as the \ command.

F7: Same as `t`: trace a line of code, stepping into functions and method calls.

F8: Same as `p`: trace a line of code, stepping over function and method calls.

TAB: Switch windows. When the cursor is in the command area, it moves to the source window; when in the source window, it moves back into the command area.

Up and Down arrows: When the cursor is in the command area, these keys scroll the source display up and down one line. When in the source window, they move the cursor up and down one line.

PgUp and PgDn keys: Scrolls text in the source window up or down one page (i.e., the number of lines of text displayed in the window).

Ctrl-PgUp and ctrl-PgDn: Moves text in the source window to the top or bottom (respectively) of the source file.

Ctrl-Home: Shows current line of source in the source window. If you have switched to another file with the `fv` command, the source file containing the current line will replace the other file in the source window.

TDB on Macintosh

When you start the Macintosh version of TDB, a dialog will let you select the game you want to debug and any additional information you need to specify. The information that you specify with command line options on other operating systems is entered through this dialog on the Macintosh. Once you have entered all of the necessary information, press the "OK" button to start the debugging session.

The debugger has a window for your game's output, and a separate window for entering debugger commands and viewing debugger output. An additional window displays the current source file (although you can use this window to view another file using the `fv` command, described earlier in this chapter). In addition, a separate window displays watch expressions.

Menu Commands

You can access most debugger commands directly from the menu bar. A command selected from the menu bar uses a dialog to ask you for any additional information required for the command. Note that many of the menu items have associated Command keys, so you can select these items with a single keystroke. For example, you can press Command-G to resume execution (the `g` command), or Command-T to step one line (the `t` command).

Source Window

You can set breakpoints using the mouse in the source window. To set a breakpoint at a particular line, simply move the mouse over that line in the source window, and click the mouse. A small

diamond mark will appear in the left margin of that line, indicating that a breakpoint is set at that line. (Note that breakpoints that you set using the `bp` command will also show up in the source window with a diamond mark.) To remove the breakpoint, click on the line again, and the diamond mark will disappear.

If you hold down the Option key when you click the mouse on a line in the source window, the debugger will allow you to set a conditional breakpoint on that line. A dialog box will appear; type an expression in the dialog box and select the "OK" button to set a breakpoint with a condition.

Whenever the debugger gets control, it will update the source window to display the next line to be executed (which is referred to as the current line), with a small arrow to the left of the line. When your game is running, the debugger does *not* keep the current line display up-to-date; the current line arrow is only meaningful when the debugger has control.

Watch Window

The watch window is displayed whenever one or more watch expressions are active; it is removed from the screen when no watch expressions are set.

You can delete a watch expression directly from the watch window. Click the mouse on a watch expression; it will be highlighted to show that it has been selected. Now you can use either the delete key or the "Clear" item in the "Edit" menu to delete the watch expression.

Note that watch expressions are kept up-to-date while the game is running, even when the debugger doesn't have control.

Who has control?

On MS-DOS, it's always obvious whether the debugger or the game has control, because the entire screen displays one or the other. On the Macintosh, however, all of the windows are always on the screen, so it may sometimes not be entirely obvious whether your game is running or suspended. The way to tell is that you can only enter commands in either the game window or the debugger window. When your game is active, you can only enter commands into the game window; when your game is suspended (so the debugger has control), you can only enter commands in the debugger command window. You can also tell by looking at the debugger menus: all of the debugger menu items are disabled while your game is running, except for the "Stop" item under the "Execution" menu. While the debugger has control, everything except the "Stop" item is enabled.

Give us the tools, and we will finish the job.
— WINSTON CHURCHILL, *Radio broadcast* (1941)

Index

In addition to checking for entries in this Index, readers are encouraged to check the several reference sections of this manual for information on particular items.

- Descriptions of the built-in functions appear in an alphabetical listing in Chapter 5, "Language Reference."
- The classes defined in `adv.t` are described in a listing in Appendix A, "Adventure Definitions"; an alphabetical index to the listing appears at the beginning of that appendix. In addition, the appendix includes a list of the verbs and a list of the prepositions defined in `adv.t`.
- The built-in property names are described in an alphabetical listing in Chapter 4. This chapter also contains an alphabetical list of the objects and functions that game programs are required to define.
- Appendix E contains a listing of all of the system error messages, arranged by error number.

The author prefers not to generate indexes automatically; he likes to reread his books as he checks the cross-references, thereby having the opportunity to rethink everything and to catch miscellaneous errors before it is too late. As a result, his books tend to be delayed, but the indexes tend to be pretty good.

— DONALD E. KNUTH, *The TEXbook* (1984)