The TADS 2.2 Upgrade Manual
Version 2.2 Release Notes

Mike Roberts[1] — High Energy Software

Jean-Henri Duteau[2] — Myrias Computer Technologies

November 14, 1996

[1]Original Author
[2]TEX converter

Contents

Chapter 1

Version 2.2 Release Notes

Welcome to TADS 2.2!

This booklet is a supplement to the *TADS Author's Manual*, version 2.0, that describes all of the new features that have been added since version 2.0.

This booklet has two parts. Part I describes the additions and changes to TADS since version 2.0. If you're already familiar with version 2.1, you've already seen some of this material before; we've marked the sections that were also described in the version 2.1 release notes to make it easy to see at a glance what's new.

Part II of this booklet is a full new chapter describing the TADS player command parser. Because so many of the new features since version 2.0 affect the parser, it would have been difficult to assemble a clear picture of how the parser works by reading unconnected descriptions of the numerous new features. Part II therefore is a complete description of the entire TADS parser, including all of the modern features, along with everything that was present in version 2.0.

1.1 "Replace" and "Modify" (2.1)

Most game authors find that, when writing a substantial game, they can't avoid modifying objects in adv.t. While there's nothing intrinsically wrong with this, it creates a problem when a new version of TADS is released: you must either continue to use your old version of adv.t, or take the time to reconcile your changes with any changes we made in the new version. If you continue to use the old version, you won't be able to take advantage of any improvements or corrections we've made to adv.t.

In version 2.1, we added a mechanism that lets you avoid this problem: the **replace** and **modify** keywords. These new keywords let you make changes to objects that have been previously defined in the game program. In other words, you can #include the standard definitions in adv.t, and then make changes to objects that the compiler has already finished compiling. Using these new keywords, you can make three types of changes to previously-defined objects: you can replace a function entirely, you can replace an object entirely, or you can add to or change methods already defined in an object.

To replace a function that's already been defined, you simply use the keyword **replace** before the new function definition. Following the keyword **replace** is an otherwise normal function definition.

```
#include <adv.t>

replace scoreStatus: function(points, turns)
{
         setscore(cvtstr(points) + ' points/' +
                  cvtstr(turns) + ' moves');
}
```

You can do exactly the same thing with objects. For example, you can entirely replace the fastenVerb defined in adv.t:

```
#include <adv.t>

/* we don't want 'buckle' - replace the verb */
replace fastenVerb: deepverb
        verb = 'fasten'
        sdesc = "fasten"
        prepDefault = toPrep
        ioAction(toPrep) = 'FastenTo'
;
```

Replacing an object entirely deletes the previous definition, including all inheritance information and vocabulary. The only properties of a replaced object are those defined in the replacement; the original definition is entirely discarded.

You can also **modify** an object, retaining its original definition, including inheritance information, vocabulary, and properties. This allows you to add new properties and vocabulary. You can also override properties, simply by redefining them in the new definition.

The most common addition to an object from adv.t will probably be new verb associations and added vocabulary:

```
modify pushVerb
        verb = 'nudge'
        ioAction(withPrep) = 'PushWith'
;
```

Note several things about this example. First, no superclass information can be specified in a modify statement; this is because the superclass list for the modified object is the same as for the original object. Second, note that vocabulary has been added. The new vocabulary doesn't replace the original vocabulary, but simply adds to the original vocabulary. Further note that the verb association pseudo-properties, doAction and ioAction, are legal in a modify statement. Any new doAction or ioAction definitions are added to the original set of definitions.

In a method that you redefine with **modify**, you can use pass and inherited to refer to the *replaced* method in the original definition of the object. In essence, using **modify** renames the original object, and then creates a new object under the original name; the new object is created as a subclass of the original (now unnamed) object. (There is no way to refer to the original, unmodified object directly; you can only refer to it indirectly through the modified object.) Here's an example of using pass with **modify**:

```
class testClass: object
        sdesc = "testClass"
;
testObj: testClass
        sdesc =
        {
                "testObj...";
                pass sdesc;
        }
;
```

```
modify testObj
        sdesc =
        {
                "modified testObj...";
                pass sdesc;
        }
;
```

Evaluating testObj.sdesc will display this:

```
modified testObj...testObj...testClass
```

You can also replace a property entirely, erasing all traces of the original definition of the property. The original definition is entirely forgotten–using pass or inherited will refer to the method inherited by the original object. To do this, use the replace keyword with the property itself:

```
modify testObj
        replace sdesc =
        {
                "modified testObj...";
                pass sdesc;
        }
;
```

This would result in a different display for testObj.sdesc:

```
modified testObj...testClass
```

The replace keyword before the property definition tells the compiler to completely delete the previous definitions of the property. This allows you to completely replace the property, and not merely override it, meaning that pass and inherited will refer to the property actually inherited from the superclass, and not the original definition of the property.

1.2 Dynamic Object Creation

TADS now allows you to create and delete objects dynamically at run-time. This is done through two new operators: new and delete. To create a new object, use this syntax:

```
x := new bookItem;
```

This dynamically creates a new object whose superclass is bookItem. When this statement is executed, the runtime creates a new object, assigns its superclass to be bookItem, and executes the construct method in the new object; this method can perform any creation-time setup that you wish to do. The default construct method in the class thing in adv.t simply moves the new object into its location–this is necessary so that the contents list of the location is updated to include the new object.

An expression involving the operator new applied to a class can be used wherever an object expression can be used. When the expression is evaluated, a new object of the given class is created.

Note that you can create a new object that has no superclass by using the keyword object:

```
x := new object;
```

If you're familiar with C++ constructors, you should note an important difference between the construct method and C++ constructors. A C++ constructor in a derived class will automatically call the construct in the base class (except for a virtual base class). The TADS construct method does *not*automatically call superclass construct methods; instead, construct works exactly the same as any other TADS method. You can, of course, call the superclass construct method explicitly using the inherited mechanism, just as with any other method. The same is true of the destruct method (described below).

A new object inherits all of the vocabulary of its superclass.

To destroy an object create with new, use this syntax:

```
delete x;
```

This first calls the destruct method of the object to notify it that it is about to be deleted, then destroys the object. Further references to the object are illegal, since its memory has been released (and thus may be given to an other object). The default destruct method in the class thing in adv.t moves the object into nil, which removes it from its container's contents list–this is necessary so that the reference to the object in that list is removed.

Only objects created with new can be destroyed with delete. Objects that are defined statically in your game's source file cannot be deleted at run-time.

Object creation and deletion works correctly with the UNDO mechanism. If the player uses UNDO after a move that created an object, the object will be destroyed; likewise, if a player uses UNDO after a turn that deletes an object, the object will be re-created with the same property values it had prior to deletion.

Similarly, dynamically-created objects are preserved across SAVE and RESTORE operations.

Note that TADS does not perform any garbage collection on dynamically-created objects. The system is not capable of determining whether an object is accessible or not. Hence, if you lose track of any objects you create with new, they will remain part of the game forever–they will even be saved along with saved games and restored when the games are restored. You must be careful to keep track of all objects you create to avoid filling all available memory (and the swap file) with unreachable objects. Even though TADS will eventually swap inaccessible objects out of memory, the objects will consume object identifiers, which are a limited resource: only 65535 objects can be created within a game, includi ng both static and dynamically-created objects.

1.3 Dynamic Vocabulary

New built-in functions let you dynamically add to and delete from the vocabulary words of an object. You can also get the vocabulary words of an object at run-time.

To add to an object's vocabulary, use the new addword() built-in function. This function takes three arguments: an object, a vocabulary property pointer, and a word to add. For example, to add "red" as an adjective to the object myBook, you would do this:

```
addword(myBook, &adjective, 'red');
```

To delete the same word, you would write a similar call to the new built-in function delword():

```
delword(myBook, &adjective, 'red');
```

You can add to and delete from the words of any object, including both static objects (explicitly defined in your source code) and dynamically-created objects (created with the operator new).

Changes made by addword() and delword() are tracked correctly by the UNDO mechanism, and are saved and restored along with saved games.

To get the words belonging to an object, use the new getwords() built-in function. This function takes two arguments: an object, and a property pointer; it returns a list of (single-quoted) strings, which are the vocabulary words for the object. For example, assume we define myBook as follows:

```
myBook:  item
        sdesc = "small red book"
        adjective = 'small' 'red' 'tiny'
        noun = 'book'
        location = room2
;
```

Also assume we haven't made any calls to addword() or delword() for myBook. In this case,

```
getwords(myBook, &adjective)
```

would return this list:

```
['small' 'red' 'tiny']
```

Note that the order of the words in the list is not predictable, so you shouldn't expect the words to be in the same order as they were when you defined them in the source file, or in the same order as they were added with addword().

1.4 Getting Verb Information

Another new built-in function lets you retrieve information on a verb. This new function, verbinfo(), takes as parameters a deepverb object and an optional preposition object, and returns a list that gives the parser information for this verb.

When you call verbinfo() with a single argument (a deepverb object), the return value is a list with two elements, giving information on the doAction definition for the deepverb object:

[1] direct object verification property pointer (verDo*Verb*)

[2] direct object action property pointer (do*Verb*)

When you call verbinfo() with two arguments (the first is a deepverb object, and the second is a preposition object), the return value is a list with four elements, giving information on the ioAction definition for the deepverb object that matches the given preposition object:

[1] direct object verification property pointer (verDo*Verb*)

[2] indirect object verification property pointer (verIo*Verb*)

[3] indirect object action property pointer (io*Verb*)

[4] true if this ioAction has [disambigDobjFirst] flag, nil otherwise

In either case, if the deepverb doesn't have a corresponding doAction or ioAction method, the function returns nil.

As an example, suppose we define a verb that looks like this:

```
pryVerb:  deepverb
        verb = 'pry'
        sdesc = "pry"
        doAction = 'Pry'
        ioAction(withPrep) = 'PryWith'
;
```

With this definition, verbinfo(pryVerb) would return this list, which provides information on the doAction definintion:

```
[&verDoPry, &doPry]
```

verbinfo(pryVerb, withPrep) would return this list:

```
[&verDoPry, &verIoPry, &ioPry, nil]
```

If the ioAction definition had included the [disambigDobjFirst] flag, the fourth element of the list would have been true.

Calling verbinfo() for pryVerb and any preposition other than withPrep will return nil, because the verb doesn't define an ioAction for any other preposition. Similarly, if the verb didn't have a doAction method, verbinfo(pryVerb) would have returned nil.

1.5 Indistinguishable Objects

The ability to create new objects at run-time leads to some interesting problems involving indistinguishable objects. Although you could use addword() to make your newly-created objects distinguishable from one another, this will not always be desirable; for example, if you create new gold pieces that serve as currency, you will probably not want them to be uniquely name d.

To support indistinguishable objects, especially those created dynamically at run-time, the system now has a property that you can set to indicate to the parser that an object does not need to be distinguished from others of the same class. The new propert y is isEquivalent. When isEquivalent returns true for an object, all other objects with the same *immediate* superclass are considered interchangeable by the parser. When a player uses one of these objects in a command, the parser will simply pick one arbitrarily and use it, without asking the player which one.

If a player uses a noun that is ambiguous with multiple equivalent items and one or more other items, the parser will need to disambiguate the objects as usual. In such cases, the parser's question will list the distinguishable items only once. For example, assume we have five gold coins that are all equivalent (in other words, they all have isEquivalent set to true, and they all are immediate subclasses of the same class). Assume further that a silver coin and a bronze coin are also present in the room.

```
Treasure Room
```

```
   You see a bronze coin, five gold coins,
   and a silver coin here.

>get coin

Which coin do you mean, the bronze coin, a gold coin,
  or the silver coin?
```

Note that the objects which appear only once are listed with "the" (using the property thedesc), while the indistinguishable objects are listed only once, with "a" (using adesc).

The functions listcont and itemcnt in adv.t have been modified so that they list equivalent objects intelligently; the functions isIndistinguishable and sayPrefixCount have been added to adv.t to help with counting and listing indistinguishable objects correctly. The contents of the Treasure Room are listed in the example above in th e new format. Refer to these functions in adv.t for examples of recognizing objects as equivalent, counting equivalent objects, and treating a set of equivalent objects together as a set.

Note that listcont uses the new property pluraldesc to display the name of an object when more than one equivalent object is present. In the example above, pluraldesc was used to list the gold coins. This property has been added to the class thing in adv.t, but you may need to override it for some objects–the default implementation simply displays the sdesc plus the letter "s".

1.6 Capturing Displayed Text

TADS now has a feature that lets you capture the text displays that are generated by double-quoted strings and by the say() built-in function. This feature allows you to examine the values of methods such as ldesc and sdesc; since these methods display text, there is no direct way of manipulating their text as strings. The new output capture feature makes it possible for you to examine and manipulate any text t hat ordinarily would simply be displayed.

To use this new feature, you first tell TADS that you wish to begin capturing output by calling a new built-in function:

```
stat := outcapture(true);
```

This begins capturing output. The return value is a status code that you use in a subsequent call to turn off output capturing; this status code allows output capturing calls to be "nested". since the status code allows the capturing status that was in eff ect on a call to begin capturing to be restored on the subsequent call to end capturing. This status code is for use by TADS only–the only thing you do with it is use it in the subsequent call to end capturing.

While output is being captured, any text that would normally be displayed is instead saved by the system. The user does not see any text displayed while capturing is in effect. After you begin capturing, simply call any methods whose displays you want to examine; for example, you could call redbook.sdesc if you want to examine the short description of the object redbook.

After you've finished calling the methods whose displays you want to examine, end capturing by calling outcapture() again, passing as the argument the status code returned by the first call:

```
str := outcapture(stat);
```

This second call tells TADS to stop capturing output. The return value of this function is a (single-quoted) string containing all of the text that was displayed since the corresponding call to outcapture(true).

When text is being captured, TADS expands all format strings (strings such as "%you%"), and processes "`\^`" and "`\v`" sequences (which convert the next character to upper- and lower-case, respectively). However, all other escape sequences (such as "`\n`" and "`\t`") are left in the string intact.

You can display the string returned from the call to outcapture(stat) using the say() built-in function. This should result in the same text display that would have occurred if you hadn't turned on capturing.

You can nest calls to outcapture(); as explained above, the status code returned from the first call is used to restore the capturing status of the previous call. The string returned by a call to turn off capturing includes only te xt that was generated after the corresponding call to turn on capturing. Because calls to outcapture() can be nested, you don't have to worry when using text capturing about whether any methods you're calling also use the function.

Note that the system automatically turns off output capturing whenever it prompts the user for a command, for a missing direct or indirect object, or for disambiguation information. When the system turns off capturing, it clears the captured text, so any s ubsequent calls to turn off capturing return empty strings. You therefore can only capture text within a single command line.

1.7 Parser Changes

Many of the changes since version 2.0 have been made to the TADS player command parser. Since Part II of these release notes provides full documentation of the modern features of the parser, including new features as well as features that were present in v ersion 2.0, we won't go into too much detail in this section. Instead, we'll summarize the changes that have been made, and refer you to the appropriate section in Part II.

If you're not already familiar with the TADS parser, you shouldn't need to read this section at all, because everything described below is documented in Part II. This section is provided for users of past versions, so that you can see at a glance what's new.

1.7.1 validDoList and validIoList (2.1)

In version 2.1, we added a new mechanism for determining whether an object is accessible for a command. The new mechanism supplements the validDo and validIo methods, which test a single object to determine if it's accessible for a verb. The added methods are validDoList and validIoList; these methods return a list of valid direct objects and indirect objects (respectively) for a particular command. As with validDo and validIo, these new methods are defined in the deepverb object for a verb.

The new validDoList and validIoList methods are fully described the section on object resolution (page 94).

Note that this new mechanism, if you use it, requires that you use the new floatingItem class for any object that uses a method for its location property. The reason is that objects with non-object location values are *not* part of any contents list, so they will not be included in any validDoList or validIoList return value. You should make sure that any object in your game that has a method for its location property includes floatingItem in its superclass list.

1.7.2 floatingItem (2.1)

The new class floatingItem should be used for any object that uses a method for its location property, because of the new validDoList and validIoList validation mechanism, as described above. If you define an object with a method for its location property, the object *must* have floatingItem among its superclasses in order for it to be accepted as a valid object for the verbs defined in adv.t.

1.7.3 dobjGen and iobjGen (2.1)

In version 2.1, we enhanced the parsing process so that the TADS parser calls a general-purpose command handler in each object in a command prior to the normal verification and action method calls. These new general-purpose methods are intended to be "catch-all" handlers that let you treat all verbs, or any set of verbs, alike–since the same methods will be called in the objects regardless of the verb, you can give the same behavior to multiple verbs without having to explicitly code each verb handler indiv idually. These new methods, dobjGen and iobjGen, are described fully in part II in the section "General Verb Processing" (page 102).

1.7.4 Command Prompt Customization (2.1)

Whenever the parser displays a command prompt, it now calls the command-Prompt function. This is a function that your game program can define. See the section on reading a command (page 72) for details.

1.7.5 New Message Customization Function (2.1)

Several new functions have been added that let you customize messages that the parser generates. These functions are used to generate complex messages that are normally pieced together from multiple parseError messages. Since the parser needs to build these messages from a series of pieces, parseError lets you change the individual pieces, but not the overall method of construction–for example, it doesn't let you change the order of the phrases generated. These new functions address this problem.

The new functions are parseError2, which lets you control the message displayed when an object doesn't understand a verb (because it doesn't define or inherit a verification method for the verb); parseDefault, which lets you change the message generated when the parser is assuming a default object; parseDisambig, which lets you control the message asking the player which of several possible objects should be used for a command; and parseAskobj, which lets you change the message when the parser asks the player to supply a missing direct or indirect object. Note that parseAskobj should no longer be used, because the more useful parseAskobjActor function has been added; games written with parseAskobj will still work correctly, but new games should use the modern function. These functions are all described in detail in Part II.

1.7.6 parseAskobjActor Message Function

We added another new message customization function in version 2.2. This new function, parseAskobjActor, is similar to parseAskobj, but receives the actor as a parameter. This allows you to generate a better message when the actor isn't Me. See the section on verb templates (page 85) for details.

1.7.7 "Any" and "Either"

The parser now accepts noun phrases that begin with "any" or "either" (or different words that you define with an added slot in the specialWords statement). The player can use "any" plus a singular noun phrase, or "any of" plus a plural noun phrase ("either" is equivalent); this instructs the parser to choose one object arbitrarily when the noun phrase refers t o more than one object. See the section on noun phrases (page 80) for details.

1.7.8 Verb Action Redirection

The compiler accepts new syntax that lets you redirect, with a single, concise declaration, the verification and action methods for a particular verb from one object to another object. For example, if you have a desk that contains a drawer, and you want the verbs "open" and "close" to be directed to the drawer when the player applies them to the desk, the new syntax lets you specify this indirection with these two lines in the definition of the desk object:

```
doOpen -> deskDrawer
```

```
doClose -> deskDrawer
```

This new syntax is described in Part II in the section on verb synonyms and redirection (page 108).

1.7.9 multisdesc

The parser calls a new property, multisdesc, when listing the current direct object while executing a command with multiple objects. For example:

```
>put all in box

red ball:  Done.
rubber raft:  The raft is too large.
```

The object name listed before the response for each object is now displayed with multisdesc (in previous versions, the parser used sdesc for this purpose; for compatibility with existing games, the parser will still call sdesc for any objects that don't define or inherit multisdesc). This is the only place where multisdesc is used, so if you want to customize this type of display, you can make changes to multisdesc without any effect on the general sdesc.

1.7.10 preparseCmd

The parser calls a new function, preparseCmd, before executing each individual command on a command line. This new command is similar to preparse, but is more powerful for certain situations, because it is called with a list of words (the result of scanning the command line and breaking it into individual words), and is called with only the words making up a single c ommand, rather than the entire command line. Of course, preparse still works as before.

1.7.11 parseError improvements

Several new parseError codes have been added, and the default message generation process has been improved for certain cases. The main changes are to the messages that the parser generates when no parseAskobjActor or parseAskobj functions are present. The full set of parseError codes is listed on page 114, and the parseAskobjActor function, and the new default messages generated when it's not present, are described in the section on verb templates in (page 85).

1.7.12 Multiple Actors are allowed on a Command Line

The parser will now allow you to specify the actor to which a command is to be directed for each individual command on a command line. Previously, the parser only allowed the actor to be specified at the very beginning of the whole line, so any one command line could include commands only to a single actor. This restriction has been removed; each individual command can specify its own actor. Any command that doesn't specify an actor is directed to the same actor as the previous command on the same line. The parser now accepts sentences like this:

```
>go north.  Joe, get the box.  give it to Biff.
Biff, open the box.
```

The command "go north" is directed to the player actor (Me). "Get the box" and "give it to Biff" are directed to Joe, and "open the box" is directed to Biff.

Note that you can still direct each command to only a single actor.

1.7.13 More Notifiers, Daemons, and Fuses

The number of notifiers (set with the notify() built-in function) that can be simultaneously active has been increased to 200 (the previous limit was 100). The limit on the number of simultaneously-active fuses has been lifted from 50 to 100, and the limit on the number of simultaneously-active daemons has likewise been lifted from 50 to 100.

1.7.14 More Ambiguous Words

The number of objects to which a single word can refer has been lifted from 100 to 200. If you've received an error message such as "The word 'door' refers to too many objects", you've encountered this limit. As the TADS parser scans a noun phrase, it mus t build a list of the objects to which a particular object refers; these lists are limited in size. For this reason, you should avoid defining a word in your basic classes (such as thing). Although this limit has not been removed entirely, we have raised it to make it less likely that you'll encounter it.

1.7.15 disambigDobjFirst

You can now specify that the processing for an ioAction definition should resolve (disambiguate) the direct object first. Normally, TADS will disambiguate the indirect object first, and then resolve the direct objects with a known indirect object. For some verbs, it's desirable to be able to resolve the direct object first, so that you can decide which indirect object to use based on the known direct object. The new ioAction flag, disambigDobjFirst, allows you to tell TADS to reverse its normal object resolution order. The section on verb templates (page 85) describes this new flag.

1.7.16 Actor Disambiguation

Two new methods have been added that give you more control over the process of disambiguating actors: validActor and preferredActor. In previous versions of TADS, the parser disambiguated an actor as though the player were trying to "take" the actor–in other words, the parser used the takeVerb disambiguation methods to determine if an object was valid as an actor. TADS now gives you more control over this process by calling these new methods, which are used only for actor disambiguation. The section on checking for an actor (page 76) discusses these new methods.

For compatibility with games written for versions prior to 2.2, if your objects don't define or inherit a validActor method, the parser will continue to use the old mechanism.

1.7.17 Improved Plural Handling

The parser's handling of plurals has been improved. When you define a word as a plural, and the same word is used by another object as a noun, the parser failed to recognize the object with the noun in past versions. The parser will now try the plural fir st; if no objects are found matching the plural usage, the parser will try the noun usage instead.

1.7.18 "Of" can be used as a Preposition

The word "of" can now be used as a preposition, even if it's defined as a special word in the "of" slot. The parser no longer considers "of" to be a reserved word; the parser only checks for the special meaning in noun phrases (such as "piece of paper") wh en the context allows it. You can now create commands such as "accuse *dobj* of *iobj*".

1.7.19 Adjective-Preposition Overlaps

When we introduced the new sentence format VERB PREP IOBJ DOBJ (introduced in version 2.1.1) created a problem when the same word was defined as both an adjective and a preposition. The problem was that the parser always attempted to interpret a phrase suc h as "push off button", where "off" is defined as a preposition but also as an adjective for the button, with "off" as a preposition.

The parser has been changed so that the adjective affiliation is stronger. If an word that can be either a preposition or an adjective is used as part of a noun phrase that refers to at least one object, the parser will interpret the word as an adjective. Only when the word doesn't form a noun phrase with subsequent words will it be interpreted as a preposition.

1.7.20 rejectMultiDobj

The parser now lets you prevent the player from using multiple words for a particular command; you can control whether multiple objects are acceptable separately for each verb. The parser calls the new method rejectMultiDobj whenever the player uses a list of objects or "all" with a command; if this method returns true, the parser ignores the command. The section on multiple objects (page 99) discusses this new method.

1.7.21 Improved cantReach Handling

The parser gives you greater control over the error message generated when a set of objects are visible but not accessible for a command. If you define a cantReach method in a deepverb object, the parser will use that method rather than calling the cantReach method for each inaccessible object. The section on object resolution (page 94) describes this new method.

1.7.22 Using the Same Verb for Two Objects

The parser now generates a warning if you use the same vocabulary word as a verb property for multiple objects. The new message is TADS-452, "warning: same verb defined for two objects". If you define the same vocabulary word as a verb multiple times, the parser will arbitrarily use one of the deepverb definitions, and ignore the others; if you weren't aware you were re-using a verb, it could be difficult to track down why your verb handlers were never being called. The new warning flags this situation for you.

1.7.23 Multiple Numbers and Strings

The parser now properly sets the strObj or numObj value property for each individual number of string in a command if more than one appears in a single command. For example:

```
>type 1, 2, 3 on keypad
```

The value property of the numObj object will be set to 1 when processing the first number, 2 for the second, and 3 for the third, as you'd expect. In past versions of TADS, the numObj and strObj value properties were set only for the first number or string in a command.

1.7.24 "One" and "Ones" are no longer Reserved

The special words "one" and "ones" (or the words you define in these specialWords slots) are no longer reserved, so you can use them as ordinary vocabulary words. The parser now treats the words as having their special meanings only in the contexts where they're needed (in particular, they'll have t he special meaning in responses to disambiguation questions, so that the player can respond to a disambiguation question with a phrase such as "the red one").

1.7.25 "abort" within a Daemon or Fuse

You can now use the abort statement within a daemon or fuse, and it will work as expected, by terminating the current routine and skipping any remaining fuses or daemons on this turn. In past versions, abort was not allowed in a fuse or daemon.

1.7.26 New sentence format: VERB PREP IOBJ DOB

The parser now accepts sentences in the format VERB PREP IOBJ DOBJ. This construction is intended mostly for use in games written in non-English languages that use this sentence format, since it's not a common format for English sentences. Refer to the se ction on verb templates (page 85) for details of how this sentence format is treated.

1.7.27 nilPrep

In past versions of TADS, when the player entered a sentence in the format VERB IOBJ DOBJ, the parser used "to" as the preposition; there was no way to change this. You can now specify the preposition to use for sentences in this format by specifying the p reposition object as the value of the property nilPrep in the deepverb object for the verb. This new property is discussed in the section on verb templates (page 85).

1.8 C-Style Operators

TADS users who are also C programmers often find the substantial similarity between TADS and C to be convenient, but also find the slight differences to be a source of confusion when switching between the two languages. TADS now offers the option to use C- style operators.

First, we've added the remaining C operators to the TADS language:

`a % b` Returns the remainder of dividing a by b

`a %= b` Assigns (`a % b`) to a

`a != b` Equivalent to (`a <> b`)

`!a` Equivalent to (`not a`)

`a & b` Bitwise AND

`a &= b` Sets a to (`a & b`)

`a | b` Bitwise OR

`a |= b` Sets a to (`a | b`)

`a && b` Equivalent to (`a and b`)

`a || b` Equivalent to (`a or b`)

`a ^ b` Bitwise XOR of a and b

`a ^= b` Sets a to (`a ^ b`)

`~a` Bitwise negation of a

`a << b` a shifted left by b bits

`a <<= b` Sets a to (`a << b`)
`a >> b` a shifted right by b bits
`a >>= b` Sets a to (`a >> b`)

Some of these operators, such as !, &&, and `||`, are merely synonyms for existing operators. The "bitwise" operators act on numeric values rather than logical values; they treat their operands as bit vectors, and apply the operation to each bit of the numbers. For example, 3 & 2 has the value 2, since the bit patterns are "011" and "010", respectively. The bit-shift operations are equivalent to multiplying or dividing by a power of 2: 1 ¡¡ 5 has the value 32, since it's equivalent to multiplying 1 by 2 raised to the 5th power.

In addition, TADS now has a mode which uses the C-style assignment operator. Normally, the TADS assignment operator is :=, and the equality operator is =. In C, these operators are = and == respectively. You can now tell TADS to use the C-style operators instead of the TADS version. By default, TADS still uses its own version of the operators. There are two ways to switch into C-sty le operator mode: by using a command-line option, or by using a new #pragma compiler directive in your source code.

To compile an entire game in C mode, use the -C+ command line option (Macintosh users will find a menu item for C-style operators under the "Options" menu; check this item to enable C operators, and uncheck it to use standard TADS o perators). Using the -C+ compiler option enables C operator mode for the entire game's source code. (The -C- option explicitly turns off C operator mode. This is the default mode.)

To specify that a particular file is to be compiled in C mode, you can use the new directive #pragma C+. The similar directive #pragma C- specifies that TADS operator mode is to be used. These directives can appear anywhere in a source file (outside of comments and strings); they must be alone on the line, and must not be preceded by any whitespace on the line.

A #pragma setting affects only the current source file, and any files it includes. The header files included with TADS (adv.t and std.t) both use TADS operators, so they explicitly specify #pragma C-; because these directives are limited to the header files, you can freely include adv.t from a file that uses C operator mode without having to worry about setting the mode to or from TADS mode. Simply #include ¡adv.t¿ exactly as you did before–even if your source file uses C mode, adv.t will compile correctly, because it sets the operator mode back to TADS for its own contents, and TADS automatically restores the enclosing fi le's mode at the end of adv.t.

Note that the C-style operator mode setting affects only the assignment and equality operators. You can use all of the new C operators (such as the new bitwise operators) in either mode–all of these symbols were invalid in previous versions of TADS, so th ere's no danger that they'll be misinterpreted for old games.

When the compiler is using C-style assignment operators, it issues a new warning, "possibly incorrect assignment", whenever it finds a statement in this format:

```
if (a = 1) ...
```

While this statement is legal, with C-style operators it has the effect of assigning the value 1 to a; since the value of an assignment is the value assigned, this if will always succeed. It's a common error for C programmers (even highly experienced ones) to write this type of statement when they really want to compare the values; we originally chose to use ":=" as the assignment operator in TADS to reduce the likelihood of this type of error. Now that TADS can be switched to C syntax for assignments and comparisons, we've added the "possibly incorrect assignment" warning to help catch these; the compiler will flag as signments made in if, while, and do statements, and in the condition clause of for statements. To suppress this warning, you can explicitly test the value of the assignment like this:

```
if ((a = 1) != 0) ...
```

There are a couple of minor complications with the new operators.

First, the ¿¿ operator can't be used in an expression embedded in a string with the ¡¡ ¿¿ construct, because it would be taken for the ¿¿ that terminates the embedded expression. Even adding parentheses won't help, because the compiler recognizes the ¡¡ ¿¿ construct before it looks at any expression in progress. So, this type of code won't work:

```
sdesc = "a / 128 = << (a >> 7) >> "
//Won't work!
```

Second, the & operator now has a binary interpretation in addition to its unary interpretation. For the most part, this won't create any confusion, but there's one situation in which it might: in lists. You might have lists in yo ur games that look like this:

```
mylist = [&prop1 &prop2 &prop3]
```

In past versions, since the & operator could only be a unary operator, this construct was unambiguous; however, now that & can be a binary operator, this could be interpreted either as three expressions involving unary & operators, or as a single expression involving one unary & operator and two binary & operators.

For compatibility with past versions, TADS will interpret the & operators as *unary* operators. When it finds this construct, though, it will warn you that it is ambiguous. (The new warning is TADS-357, "operator '&' interpreted as unary in list.) You can suppress this warning in one of two ways. First, you can render the list unambiguous. To do this, use a comma between each pair of list elements:

```
mylist = [&prop1, &prop2, &prop3]
```

Note that if you actually want the binary interpretation, you should simply enclose the expression in parentheses:

```
mylist = [(2 & 3 & 4)]
```

The other way you can suppress this message is with the new -v-abin compiler option, which tells the compiler not to generate the warning. The compiler still interprets the operator the same way when you specify -v-abin–it just doesn't tell you about it.

Note that TADS will treat *any*operator which has both a unary and binary interpretation as a unary operator when it finds it within a list, and will generate the TADS-357 warning. For the - operator, this is a change from past versions, which used the binary interpretation when in a list. We don't anticipate that this will be a compatibility problem, because the old binary interpretation was almost never desirable, and we think users avoided it; however, if you have an older game, you may wish to compile without the -v-abin option at least once, and check any lines where the TADS-357 warning is generated for - or + operators, to determine if your game's behavior will change with the new version. Any TADS-357 warnings generated for the & operator can be safely ignored for a game written with a previous version of TADS.

1.9 New Preprocessor Directives

TADS now lets you define preprocessor symbols. The #define directive creates a preprocessor symbol definition:

```
#define TEST  "This is a test!"
```

This defines the symbol TEST, and specifies that this symbol is to be replaced by the text “This is a test!” whenever it appears (outside of strings or comments) in your source code. Note that the definition of a symbol need not be a string–the entire text of the rest of the line is assigned to the symbol.

You can use a preprocessor symbol that you've defined simply by putting the symbol in your code:

```
sdesc =
{
        TEST;
}
```

When the compiler encounters the preprocessor symbol, it replaces the symbol with its definition, so the compiler treats this the same as:

```
sdesc =
{
        "This is a test!";
}
```

You can delete a preprocessor symbol that you've defined using the #undef directive:

```
\#undef TEST
```

The compiler automatically defines a number of preprocessor symbol:

__TADS_VERSION_MAJOR is defined as the major version number of the current compiler. (Note that this symbol starts with *two*underscore characters, as do most of the symbols that the compiler defines for you.) Currently, this is defined as 2.

__TADS_VERSION_MINOR is the minor version number, currently 2.

__TADS_SYSTEM_NAME is defined as a single-quoted string giving the name of the current system. For DOS systems, this is 'MSDOS'; for Macintosh systems, this is 'Macintosh'. TADS also defines the value of this symbol as a symbol itself–on MSDOS systems, the compiler defines the symbol MSDOS, and on Macintosh systems, the compiler defines Macintosh. (The value of the system name symbol is simply 1; it's purpose is to allow for conditional compilation, so the value isn't important.)

__DEBUG is defined if the game is being compiled for debugging (with the -ds command line option). By testing this symbol with #ifdef, you can easily include parts of your game (such as special "magic" verbs that you use for debugging) only in testing versions of your game, without having to worry about removing them manually when producing the final versio n of the game for players. The value of this symbol is 1 if it's defined; the purpose of this symbol is to allow for conditional compilation, so its value isn't important.

__DATE__ is defined as a single-quoted string giving the date when the current compilation began, in the format 'Jan 01 1994'.

__TIME__ is defined as a single-quoted string giving the time when the current compilation began, in the 24-hour format '01:23:45'. The time isn't updated during the course of a compilation–it's always the time when compilation began.

__FILE__ is defined as a single-quoted string giving the name of the current source file being compiled.

__LINE__ is defined as a number giving the line number currently being compiled.

You can define symbols from the compiler command line. The new D option lets you specify a symbol to define, and optionally a value. Specify the symbol, then an equals sign, then the value; if you omit the equals sign, the default value is 1. For example, to define the symbol DEMO from the command line, you could use this command:

```
tc -DDEMO mygame
```

You can also specifically delete preprocessor symbols that the compiler defines (other than __FILE__ or __LINE__). You can also undefine any symbols defined earlier on the command line with D options, which may be useful if you're using a configuration file that defines certain symbol. To undefine a symbol, use the U option:

```
tc -UDEMO mygame
```

If the symbol DEMO was defined earlier on the command line, the definition is deleted.

The Macintosh compiler has a new preprocessor options dialog that you can access from the "Options" menu. You can use this dialog to enter symbols to define; in the text box, simply list the symbols (one per line) that you wish to define. If you want to a ssign values to these symbols, use an equals sign, followed by the value. Another text box in the same dialog lets you list predefined symbols that you want to delete; simply list the symbols to be deleted, one per line.

You can test for the existence of a preprocessor symbol with #ifdef ("if defined") and you can test to see if a preprocessor symbol is *undefined* with the #ifndef ("if *not* defined") directive. These directives let you conditionally include code in your program, depending on whether or not a symbol is defined. For example:

```
#ifdef TEST

sdesc = { TEST; }

#endif
```

The code between the #ifdef and the #endif is included only if the preprocessor symbol TEST is defined. There's also a #else directive, which lets you include a block of lines if the most recent condition failed.

Conditional compilation is particularly useful with the symbols you define from the command line with the -D option, since it allows you to write your game in such a way that certain features are enabled when you use a particular set of -D options. This allows you to use a single set of source files, but produce a variety of different .GAM files. For example, if you want to be able to generate a subset of your game as a demo, you could use conditional compilation to include or discard sections of the game depending on whether you're compiling the demo or the full game. Similarly, you could use conditional compilation to include certain features only when you're compiling a debugging version of your game; note that the compiler makes this even easier, because it defines the symbol __DEBUG when you're compiling with debugging symbols.

We've also added the #error directive. This directive lets you generate an error from within your game. Any text following the #error on the line is displayed to the user as the text of the error message. For example:

```
#ifndef TEST

#error TEST is not defined!

#endif
```

1.10 New Output Escape Sequences

The output formatter now recognizes two new escape sequences.

The first is "\-" (a backslash followed by a hyphen), and was added in version 2.1. This sequence tells the output formatter to pass the following two bytes as-is, without any interpretation. This is useful for multi-byte characte r sets (such as on a Japanese-localized Macintosh), where a single character may be represented by two bytes. For the most part, you can freely mix single- and double-byte characters within text strings without any special work. However, some double-byte characters contain the ASCII code for a backslash as one of their two bytes; in these cases, the output formatter incorrectly interprets the byte whose code is "\\" as introducing an escape sequence. By preceding such characters with "\-", you can prevent the parser from interpreting the byte with the backslash code as an escape sequence, so the character will be displayed correctly.

The second new escape sequence is "\v" (a backslash followed by the small letter "v"). This sequence is the opposite of "\^" (a backslash followed by a circumflex). Whereas "\^" converts the next character displayed to a capital letter, "\v" converts the next character to a small letter. Displaying "\v" is equivalent to calling the nocaps() built-in function.

1.11 File Operations

TADS now has support for reading and writing files from within your game program. This new feature is intended to let you save information independently of the game-saving mechanism, which allows you to transfer information between sessions of a game, or e ven between two different games.

Operations on files are all performed through a *file handle*. This is a special value, generated by TADS, that you use to refer to a file. TADS creates a file handle when you *open* a file; once you've opened a file and obtained its file handle, you can read and write the file. Once you're done with the file, you close the file, which deletes the file handle.

To open a file, use the fopen() function. This function takes two arguments: a (single-quoted) string giving the name of the file to open, using local file system conventions, and a "mode". The mode argument is one of these single -quoted string values:

'r' open file for reading; file must already exist

'r+' open file for reading and writing; the file is created if it doesn't already exist

'w' create a new file for writing; the file is deleted if it already exists

'w+' create a new file for reading and writing; the file is deleted if it already exists

For maximum portability, you should avoid using volume names, directories, folders, or other path information in filenames, because any such information might be specific to your computer.

The return value of fopen() is a file handle; you must save the file handle so you can use it later to perform operations on the file. If the operation fails, and the file cannot be opened as requested, fopen() returns nil. Opening a file could fail for a number of reasons; for example, if you attempt to open a non-existent file with mode 'r', fopen() will return nil, because this mode requires that the file already exists.

This example opens a new file called TEST.OUT for writing:

```
fnum := fopen('test.out', 'w');
```

To close an open file, use fclose():

```
fclose(fnum);
```

Once you've closed a file, the file handle is no longer valid; TADS may re-use the same file handle on a subsequent fopen() call. Note that the TADS runtime allows only a limited number of files (currently 10) to be open simultaneously, so you should close a file when you're done with it.

To write to a file, use fwrite(). This function takes a file handle, and a value to write; the value can be a string, a number, or true. The value can't be nil (this is because the fread() function returns nil to indicate failure; if you could write nil to a file, there would be no way to distinguish reading a valid nil from an error condition). fwrite() stores the value, along with information on its type.

The fwrite() function returns nil on success, true on failure. If the function returns true, it usually means that the disk is full.

```
if (fwrite(fnum, 'string value!')
        or fwrite(fnum, 123))
        "Error writing file!";
```

If the file is open for reading, you can read from the file with the fread() function. This function takes a file handle, and it returns a value it reads from the file. The value returned is of the same type as the value originally written at this position in the file with fwrite(). If this function returns nil, it indicates that an error occurred; this usually means that no more information is in the file (which is the case once you've read to the end of the file).

```
res := fread(fnum);

say(res);
```

You can get the current byte position in the file with the ftell() function:

```
"The current seek position is << ftell(fnum) >>. ";
```

The ftell() function returns a number giving the byte position that will be read or written by the next file operation.

You can set the file position with fseek() and fseekeof(). The fseek() function moves the file position to a particular byte position, relative to the beginning of the file. For example, this seeks to the very beginning of a file:

```
fseek(fnum, 0);
```

The fseekeof() function positions the file at its end (EOF stands for "end of file"):

```
fseekeof(fnum);
```

Note that you must be careful with fseek(). You should only seek to positions that you obtained with the ftell() function; other positions may be in the middle of a string or a number in the file, so seeking to an arbitrary location and writing could render the file unusable by partially overwriting existing data.

Note that the TADS file operations are not designed as general-purpose file system operations; in particular, these new functions don't have any provisions for creating or reading formatted files, or for exchanging information with programs other than TADS games. TADS reads and writes files in its own binary format, which is portable across platforms–you can take a binary file written on DOS, and read it on the Macintosh; however, the TADS format isn't designed to be interchanged with other programs, or eve n to produce simple text files that can be viewed or edited by a user.

1.12 New and Improved Built-In Functions

1.12.1 addword(*obj*, &*prop*, *word*)

Adds the *word* (a single-quoted string value) to the object as the given part of speech. The *prop*parameter can be noun, adjective, plural, verb, article, or preposition. You can add words to any object, including objects defined statically in your game, as well as objects created dynamically at run-time.

For examples of using this function, see the section on Dynamic Vocabulary (page 8).

1.12.2 clearscreen()

Clears the screen. Note that this function may have no effect on some platforms; for example, when the DOS runtime is operating in plain ASCII mode, clearscreen() will have no effect.

1.12.3 debugTrace(1, *flag*)

This new form of debugTrace lets you turn a new player command parser diagnostic mode on and off. If *flag* is true, this function activates the new diagnostic mode; if *flag* is nil, it turns the diagnostic mode off.

When the new diagnostic mode is active, the parser generates a series of progress messages as it analyzes the player's command. These messages provide information on how the parser interprets the words in the command. When first reading a sentence, the pa rser displays all possible parts of speech for each word. As the parser further analyzes the sentence, it displays information on each noun phrase: the words involved, the part of speech that the parser uses for each word in the noun phrase (when a word c an be used as multiple parts of speech, the parser chooses one part of speech as it reads the noun phrase), and the objects that match the words in the noun phrase.

The parser diagnostic mode may help you track down problems in which the parser refuses to recognize certain noun phrases that you would expect to be valid. Since the parser chooses among ambiguous interpretations of words, it's frequently helpful to under stand exactly how the parser is interpreting your commands; this new debugging mode should make it easier to gain this understanding.

This new mode is available in the standard runtime as well as the debugger, so debugTrace(1, *flag*)always succeeds. The function returns no value.

The original debugTrace() function still works as it always has.

1.12.4 delword(*obj, &prop, word*)

Deletes the *word* (a single-quoted string value) from the object's vocabulary for the given part of speech. The *prop*parameter can be noun, adjective, plural, verb, article, or preposition. You can delete words from any object, including objects defined statically in your game, as well as objects created dynamically at run-time. Furthermore, you can delete words that were added dynamically, as well as wo rds that were statically defined in your game.

For examples of using this function, see the section on Dynamic Vocabulary (page 8).

1.12.5 firstsc(*obj*)

Returns the first superclass of the given object. Returns nil if the object has no superclass (which will only be the case if the object was defined as being of type object).

This function is provided primarily to facilitate handling equivalent objects. In conjunction with the isEquivalent property, this function lets you determine if two objects are indistinguishable from one another. If two objects have the same immediate superclass, and they both have the isEquivalent property set to true, the two objects are equivalent.

1.12.6 fclose(*filehandle*)

Closes the file indicated by *filehandle*. Once the file is closed, no further operations on *filehandle* are valid.

1.12.7 fopen(*filename, mode*)

Opens the file whose name is given by the single-quoted string value *filename*; the file is opened in a manner according to the *mode* argument, which is a single-quoted string value:

'r' open file for reading; file must already exist

'r+' open file for reading and writing; the file is created if it doesn't already exist

'w' create a new file for writing; the file is deleted if it already exists

'w+' create a new file for reading and writing; the file is deleted if it already exists

The function returns a file handle that is used in subsequent file operations (fwrite(), fread(), fclose(), and the like) to refer to the open file.

If the operation fails, fopen() returns nil. This function can fail for a number of reasons; for example, if you attempt to open a file that doesn't exist with mode 'r', the operation will fail because this mode can only be used to open an existing file.

1.12.8 fread(*filehandle*)

Reads the next data item from the file and returns its value. The value will be of the same type as that originally written at the current position in the file with fwrite(). If an error occurs, this function returns nil; this usually indicates that you are attempting to read past the end of the file.

1.12.9 fseek(*filehandle, byteposition*)

Seeks to a byte position in the file. The *byteposition* value should be a value previously returned by ftell(), since other values may not correspond to a value in the file.

1.12.10 fseekeof(*filehandle*)

Positions the file at its end.

1.12.11 ftell(*filehandle*)

Returns the current seek position in the file, given as a number of bytes from the beginning of the file.

1.12.12 fwrite(*filehandle, value*)

Writes the given value to the file. The *value*argument must be a number, a single-quoted string, or true. Returns nil on success, true on failure; a true return value usually indicates that the disk is full.

1.12.13 getwords(*obj, &prop*)

This function returns a list of single-quoted strings giving the vocabulary words defined for the given part of speech for the specified object. The *prop*parameter can be noun, adjective, plural, verb, article, or preposition.

For examples of using this function, see the section on Dynamic Vocabulary (page 8).

1.12.14 getfuse(*funcptr, parm*) (2.1)

The new function getfuse(*funcptr, parm*) lets you determine if the indicated fuse is still active, and if so, how many turns are left until it is activated. If the fuse is not active (it has already fired, or it has been removed with a call to remfuse, or it was simply never set), the function returns nil. Otherwise, it returns the number of turns before the fuse is activated.

1.12.15 getfuse(*obj, &msg*) (2.1)

This form of getfuse() lets you check on a fuse set with the notify() function. If the fuse is not active, the function returns nil, otherwise it returns the number of turns before the fuse is activated.

1.12.16 gettime() (2.1)

Returns the current system clock time. The time is returned as a list of numeric values for easy processing. The list elements are:

[1] year calendar year (e.g., 1992)
[2] month month number (1 for January, 2 for February, etc.)
[3] day day of the month (1 for the first day)
[4] weekday day of the week (1 for Sunday, 2 for Monday, etc.)
[5] year day day of the year (1 for January 1)
[6] hour hour of the day on a 24-hour clock (0 is midnight)
[7] minute minute within the hour (0 to 59)
[8] second second within the minute (0 to 59)
[9] elapsed seconds since January 1, 1970, 00:00:00 GMT

1.12.17 incturn(*num*)

This function has been extended to allow you to run a series of turns all at once. You can now specify a numeric argument to incturn(); the argument gives the number of turns that should pass. An argument of 1 is equivalent to calling incturn() with no arguments.

When an argument higher than 1 is specified, the function runs all of the fuses that are set to turn down within the number of turns specified, but not after that number of turns. Note that the normal incturn() doesn't actually execute any fuses, but simply burns all fuses down by one turn.

For example, if you call incturn(2), the system will first run any fuses that are set to burn down after 1 turn, then will shorten all remaining fuses by one turn. Similarly, incturn(3) first runs any fuses that are set to burn down after 1 turn, then runs any fuses set to burn down after 2 turns, then shortens any remaining fuses by 2 turns.

Fuses set with setfuse() and notify() are both affected by this routine. Note that this function has no effect on daemons.

1.12.18 inputkey() (2.1)

Reads a single keystroke from the keyboard, and returns a string consisting of the character read. inputkey() takes no arguments. When called, the function first flushes any pending output text, then pauses the game until the player hits a key. Once a key is hit, a string containing the character is returned. Note that this function does *not* provide a portable mechanism for reading non-standard keys, such as cursor arrow keys and function keys. If the user hits a non-standard key, the return value is the representation of the key on the operating system in use. To ensu re portability, you should use this function only with standard keys (alphabetic, numeric, and punctuation mark keys). Note that you will encounter no portability problems if you simply ignore the return value and use inputkey() only to pause and wait for a key.

Note that inputkey() takes its input directly from the keyboard. It does *not*look at the command line

1.12.19 intersect(*list1*, *list2*) (2.1)

Returns the intersection of two lists–that is, the list of items in both of the two lists provided as arguments. For example:

```
intersect([1 2 3 4 5 6], [2 4 6 8 10])
```

yields the list [2 4 6].

Note that the behavior for lists with repeated items is not fully defined with respect to the number of each repeated item that will appear in the result list. In the current implementation, the number of repeated items that is present in the *shorter*of the two source lists will be the number that appears in the result list; however, this behavior may change in the future, so you should try not to depend on it.

1.12.20 nocaps()

This function is the opposite of caps()–it specifies that the next character displayed will be converted to lower case. Note that you can use the new escape sequence "`\v`" to achieve the same effect.

Note that calls to caps() and nocaps() override one another; if you call caps(), and then immediately call nocaps(), the next character displayed will be lower-case–the call to caps() is forgotten after the call to nocaps().

1.12.21 objwords(*num*) (2.1)

Provides a list of the actual words the user typed to refer to an object used in the current command. The argument *num* specifies which object you're interested in: 1 for the direct object, or 2 for the indirect object. The return value is a list of strings; the strings are the words used in the command (converted to lower case, stripped of any spaces or punctu ation). If a special word such as "it", "them", or "all" was used to specify the object, the list will have a single element, which is the special word used.

For example, if the player types "take all", then objwords(1) will return ['all'] and objwords[2] will return []. Note that objwords(1) will return ['all'] even if the player typed a variation on "all", such as "take everything" or "take all but the book".

If the player types "put all in red box", the objwords(1) returns ['all'] and objwords(2) returns ['red' 'box'].

If the player uses multiple direct objects, the function will return the *current*object's words only. For example, if the player types "put blue folder and green book in red box", objwords(1) will return ['blue' 'folder'] while the first direct object is being processed, and ['green' 'book'] while the second object is being processed.

This function could potentially be useful in such cases as "ask *actor* about *object*", because it allows you to determine much more precisely what the player is asking about than would otherwise be possible.

You can call objwords() during a verb verification or action method (verDo*Verb*, verIo*Verb*, do*Verb*, io*Verb*), and during a doDefault method. Note that the return value is slightly different during a doDefault method: if the word "all" is used in the command, the function will return the list ['A'] rather than ['all'], due to the internal order of processing of the word list.

1.12.22 objwords(1) words with doDefault

You can now call objwords(1) from within the doDefault method, as described above. Until version 2.2, you could call objwords(1) only from verb verification and action methods.

1.12.23 outhide(*flag*) (2.1)

Turns hidden output on or off, simulating the way the parser disambiguates objects. The parameter *flag* is either true or nil. When you call outhide(true), the system starts hiding output. Subsequent output is suppressed–it won't be shown to the player. When you call outhide(nil), the system stops hiding output–subsequent output is once again displayed. outhide(nil) also returns a value indicating whether any (suppressed) output was generated since the call to outhide(true), which allows you to determine whether any output *would*have resulted from the calls made between outhide(true) and outhide(nil).

This is the same mechanism used by the parser during disambiguation, so it should *not*be called by a verDo*Verb* or verIo*Verb* method. This function is provided to allow you to make calls to verDo*Verb* and verIo*Verb* to determine if they will allow a particular verb with an object, just as the parser does.

There is no way to recover the text generated while output is being hidden. The only information available is whether any text was generated.

1.12.24 outhide(*flag*)–nested calls

When you call outhide(true), the function returns a status indicator, which is a value that can be used in a subsequent call to outhide() to restore output hiding to the state it was in before the outhide(true). This allows you to nest text hiding–you can hide output in a subroutine or method call, without interfering with the routine that called you or any routines you call.

To use the nested form, save the return value of outhide(true), and then use the saved value as the parameter–in place of nil–to the subsequent call to outhide(). The value returned by the second outhide() indicates whether any text output occurred *between the nested calls.* For example:

```
local old_stat1, old_stat2;
local new_stat1, new_stat2;

old_stat1 := outhide(true);
"This is some hidden test.";
old_stat2 := outhide(true);

// don't write any text here
new_stat2 := outhide(old_stat2);
new_stat1 := outhide(old_stat1);
```

Because outhide(old_stat2) indicates whether any output occurred during the *nested* outhide(true), new_stat2 is nil. However, new_stat1 is true, since output occurred after the first outhide(true). Consider another sequence:

```
old_stat1 := outhide(true);

// write no text here

old_stat2 := outhide(true);
"This is some hidden text.";
new_stat2 := outhide(old_stat2);
new_stat1 := outhide(old_stat1);
```

In this case, both new_stat1 and new_stat2 will be true, because hidden output occurred within both nested sections (even though only a single message was displayed, it was within both nested sections).

The general form of a nested outhide() section is:

```
{
        local old_stat;
        local new_stat;

        old_stat := outhide(true);
        // do whatever you want here
        // output will be hidden
        new_stat := outhide(old_stat);
}
```

The new_stat will indicate whether any output occurred between the outhide(true) and the outhide(old_stat). In addition, output hiding will be restored to the same state as it was prior to the outhide(true).

1.12.25 rand() is more random

The random number generator has been improved. Some people noticed that the old random number generator was producing somewhat less than ideally random results, especially when small upper limits were used. The interface to the new random number generator is exactly the same as before.

Note that the old random number generator is still used if you don't call randomize(). This allows test scripts (which require a fixed sequence of random numbers in order to be repeatable) that were written with older versions of TADS to continue to work unchanged. If you want numbers from the improved generator, be sure to call randomize().

1.12.26 restart(*funcptr, param*) (2.1)

This new form of restart() lets you specify a function to be called after restarting the game, but before the init() function is invoked. This new feature has been added because it would otherwise be impossible to pass any information across a restart operation: the restart() function doesn't return, and all game state is reset to its initial state by restart(). You can use this new form of the function if you want a restarted game to have different startup behavior than the game has when it's first started. Note that adv.t passes a pointer to the initRestart function (defined in adv.t when it invokes restart() in response to a "restart" command from the player; the adv.t implementation of initRestart() simply sets the flag global.restarting to true to indicate that the game is being restarted rather than first entered. You can replace the initRestart function defined in adv.t if you want to customize the restart behavior.

The *param* value is simply passed as the parameter to the function to be called; this allows you to pass information through the restart. For example, if you start the game with a questionnaire asking the player's name, sex, and age, you could pass a list containing the player's responses to your restart function, and have the restart function store the information without making the player answer the questionnaire again. The call to restart() in adv.t uses global.initRestartParam as the parameter for the initRestart function; so, if you provide your own version of initRestart, you can simply store the necessary information in global.initRestartParam to ensure that it's passed to your function.

1.12.27 restore(nil) (2.1)

This special new form of the restore() function allows you to choose the time during startup that the player's saved game is restored when the player starts your game with a saved game already specified. When you call restore(nil), the system checks to see if a saved game was specified by the player at startup, and if so, immediately restores the game and returns nil. If no game was specified, the function returns true.

Current, it's possible for a player to start a game in this manner only on the Macintosh, but the new restore() functionality will work correctly on all platforms. On the Macintosh, the operating system allows an application to be started by opening one of the application's documents from the desktop;

the application is started, an d is informed that the user wishes to open the specified file. Saved game files on the Macintosh are associated with the game executable that created them in such a way that the game is executed when a saved game is opened. This is simply a convenience fe ature on the Mac that allows a player to run a game and restore a saved position in a single operation.

You can use restore(nil) in your init function to choose the point at which the saved game will be restored. If your game has extensive introductory text, you could call restore(nil) prior to displaying your introductory text, and then skip the introductory text if the function returns nil.

The reason that the system doesn't restore the saved game prior to calling your init function is that you may want parts of your init function to be invoked regardless of whether a game is going to be restored or not. For example, you may want to display your copyright message, or ask a copy-protection question, even when a saved game is going to be restored.

If you do not make a call to restore(nil) in your init function, TADS will automatically restore the saved game specified by the player immediately *after* your init function returns. Hence, omitting the call to restore(nil) doesn't do any harm; this function is provided to give you greater control.

1.12.28 rundaemons() (2.1)

Runs all of the daemons. This function runs daemons set with setdaemon() and notify(). This function returns no value.

1.12.29 runfuses() (2.1)

Runs all expired fuses, if any. Returns true if any fuses were ready to run, nil otherwise. This function runs fuses set both with setfuse() and with notify().

1.12.30 setit(nil) (2.1)

You can now use nil as the argument to setit(). This prevents the player from using "it" in the next command.

1.12.31 setit(*obj, num*) (2.1)

You can now specify which pronoun you want to set. The new optional parameter *num* specifies which pronoun to set: 1 for “him”, and 2 for “her”. When *num* isn't specified, setit(*obj*) sets “it” as usual. Note that nil can be used as the *obj* argument to clear “him” or “her”, just as it can with “it”.

1.12.32 setit(*list*) (2.1)

You can now set “them” directly, simply by passing a list of objects (rather than a single object) to setit(). Calling setit() with a list argument has the side effect of clearing “it”.

1.12.33 skipturn(*num*)

This function is similar to incturn(*num*); this function *skips* the given number of turns, which must be at least 1. The difference between skipturn() and incturn() is that skipturn() doesn't run any of the fuses that burn down during the turns being skipped–instead, it simply removes them without running them.

Fuses set with setfuse() and notify() are both affected by this routine. Note that this function has no effect on daemons.

1.12.34 verbinfo(*deepverb*)

1.12.35 verbinfo(*deepverb, prep*)

This function lets you retrieve information on the doAction and ioAction definitions for the given deepverb object. This function returns a list giving the verification and actor property pointers for the appropriate verb template. Refer to the section on getting verb information (page 9) for details on this function.

1.13 Debugger Enhancements

1.13.1 Case Insensitivity in Commands

Commands are no longer case-sensitive. The command "BP" is now equivalent to "bp".

1.13.2 Call Logging (2.1)

We've added a mechanism that lets you capture a log of all method and function calls, and then inspect the log. This feature can help you determine the exact sequence of calls that TADS itself makes into your game, and also lets you see how your game and l ower-level classes (such as those from adv.t) interact. Four new command let you control logging (on the Macintosh, these are also accessible from the "Execution" menu):

c+ Begins call logging. Any previous call log is cleared, and all subsequent method and function calls and returns will be added to the new call log.

c- Ends call logging.

cc Clears the current call log.

c Displays the current call log. Shows all function and method calls after the most recent c+ and up to the current time, or the most recent c-.

The reason that commands are provided to turn call logging on and off is that the logging process can slow down your game's execution substantially. When call logging is activated, the system must do extra work every time a function or method is entered or exited. You should enable call logging at the point you're about to execute a command that you want to trace, then turn it off when you're finished with the command.

Note that the call log is limited in capacity. If the log becomes full, the oldest information is discarded to make room for new information. If you leave call logging activated for an extended period of time, information toward the beginning of the log m ay be lost.

The lines of text displayed in the call log will be indented to show nesting. The functions and method called directly by TADS will not be indented at all; anything called by these functions and methods will be indented one space; and so on. If a function or method has a return value, it will be indicated in the log (prefixed by "=¿") at the point where the function or method returns. Each call will show the object and method or function name involved, along with the arguments; t he format is the same as in a stack trace.

1.13.3 Run-time Error Catching (2.1)

The debugger will now take control when a run-time error occurs. The error message will be displayed, and execution will be suspended at the line where the error occurred. When you resume execution, the current turn will be aborted (as though an abort statement had been executed).

1.13.4 Infinite Loop Break-out (2.1)

You can now break out of an infinite loop in your game. Hit Ctrl-Break if you're running on a DOS machine, or Command-Period on a Macintosh. The debugger will take control.

1.13.5 Breakpoints in Inherited Methods (2.1)

The debugger will now stop at a breakpoint in a method inherited by an object. For example, if you set a breakpoint at room.lookAround, execution will stop any time a subclass of room executes the inherited lookAround method. Of course, if a subclass overrides the lookAround method, execution will not stop at room.lookAround.

1.13.6 Memory Size Reductions (2.1)

The run-time memory requirements for the debugger have been reduced, which should allow larger games to be debugged on smaller machines (such as DOS machines) without the out-of-memory conditions that some people have experienced with past versions.

1.14 New Compiler Options

-v *level* (2.1)

This option sets the warning verbosity level. The default level is 0 (minimum verbosity: suppress warnings which are purely informational or generally do not indicate an actual problem). Other levels are 1 (suppress purely informational messages, but disp lay warnings even when they generally do not indicate a problem), and 2 (maximum verbosity: display all warnings and informational messages).

-fv *type*

Sets the .GAM file format. The *type*can be a for file format “A” (used by TADS versions prior to 2.1), b for format “B” (used by versions 2.1 up to but not including 2.2), c for format “C” (introduced in version 2.2), or * for the most recent version (currently “C”). The default is -fv*. You can use this option if you wish to compile your game so that it can be used with an earlier version of the TADS runtime.

Version 2.1 of TADS uses file format “B”, which is slightly different from the original file format, and 2.2 uses format “C”, which differs slightly from “B”. File formats other than “A” will not be accepted by versions of the runtime earlier than 2.1. Th e version 2.2 runtime is able to read any previous file format.

The changes made in version “B” make the .GAM file much more compressible with archiving utilities. If you use file format “A”, your games will not compress very well with programs such as ZIP.

The changes made in version “C” enable the disambigDobjFirst flag. If you compile with an earlier file format, you will not be allowed to use this flag.

-e *file* (2.1)

This option tells the compiler to log all error messages to *file*. The compiler writes all of the messages generated during compilation to the indicated file; the messages will also be displayed on your screen as usual. The file will be created if it doesn’t exist, and it will be overwritten with the new error log if it already exists.

-C

(Note that this is a capital “C”.) This is a toggle option that lets you turn C operator mode on and off (see the section on C operators on page 20). The default is -C-, which enables the standard TADS operators. If you specify -C+, TADS will change the assignment operator to “=” and the equality operator to “==” to match the C language.

-v-abin

Disables the warning message “operator *x* interpreted as unary in list”. (TADS-357). If you’re compiling a game written with a previous version of TADS, and you get this warning, you may want to use -v-abin to disable the warning. This warning is intended to notify you that the *new*meaning of the operator is *not*being used–in other words, TADS is warning you that it's using an interpretation that's compatible with past versions of the compiler. If your game was written for an older version of the system, th is is exactly what you want, so you probably don't need the warning.

Improved error checking for -m

The compiler now checks more carefully to ensure that the memory size settings specified with the various -m parameters (in particular, -mg, -mp, -ml, -ms, and -mh) are valid. These settings must fit within certain limits, which are now enforced more carefully. Previous versions of the compiler responded unpredictably to invalid settings.

-case-

The compiler now lets you turn off case sensitivity. If you don't want to distinguish between upper- and lower-case letters in your symbols, you can use the -case- option to turn off the compiler's case sensitivity. By default, the compiler is set to -case+, which makes the compiler treat an upper-case letter as distinct from the same letter in lower-case. When the compiler is case-sensitive, deepverb and deepVerb are different symbols. If you would prefer not to keep track of the exact case of the letters in the symbols defined in your game (or in adv.t), you can make the compiler treat upper- and lower-case letters the same by using -case-.

Note that adv.t will work either way. All of the references to a particular symbol in adv.t use the same case, so adv.t will work properly with case sensitivity turned on. However, adv.t will also work properly with case sensitivity turned off, because it doesn't have any symbols that depend on being in a particular case (for example, it doesn't define one symbol called "a" and a distinct symbol called "A"–if it did, these symbols would collide when compiling without case sensitivity).

Macintosh Compiler no longer adds ".t" Suffix

The Macintosh compiler no longer adds a default suffix to input files. In past versions, the compiler added ".t" to the name of a source file if the source file didn't have any periods in its name. While adding a default suffix is convenient on other plat forms (since it saves the user the trouble of typing the suffix on the command line), it's obviously not desirable on the Macintosh, since the user specifies the file by pointing at it–the full filename is always given, so a suffix shouldn't be added.

1.15 Changes to adv.t

1.15.1 distantItem (2.1)

The new class distantItem has been added. This new class is similar to fixeditem, except that it's intended to be used to objects that are not actually part of a room, but are *visible* from the room in the distance. For example, you might use this class for distant mountains visible from a location–the player obviously shouldn't be able to do anything to the mountains except look at it them. A distantItem can be inspected, but any other attempts to manipulate the object will receive the response "It's too far away".

1.15.2 scoreStatus (2.1)

The new scoreStatus(points, turns) function has been added. This new function simply calls setscore() with the same arguments. All other calls to setscore() in adv.t have been replaced with calls to this new function, which makes it easy to provide a new scoring format, if you wish, simply by using replace to substitute your own scoreStatus() function for the one in adv.t.

1.15.3 nestedroom.statusPrep, outOfPrep (2.1)

Two new properties have been added to the nestedroom class: statusPrep, which displays an appropriate preposition for status line displays while the player is in the nested room; and outOfPrep, which displays the correct preposition when leaving the nested room. The default values are "in" for statusPrep and "out of" for outOfPrep. The class beditem provides values of "on" and "out of" instead. If you're defining a new subclass of nestedroom, you can override these properties to provide the most appropriate messages for the subclass.

1.15.4 fixeditem and "throw" (2.1)

You can no longer throw a fixeditem at anything

1.15.5 follower changes (2.1)

The follower class's actorAction method now appropriate executes an exit statement. In addition, the follower class now uses dobjGen and iobjGen to provide more sensible responses to most verbs.

1.15.6 clothingItem: "get out of" (2.1)

You can now "get out of" a piece of clothing; this is equivalent to "take off".

1.15.7 "l at" (2.1)

The various verbs which use the word "look" plus a preposition have been modified to allow "l" in place of "look"; this applies to several verbs, including "look at" (which now accepts "l at" as equivalent), "look on", "look in", "look under", "look around", and "look through".

1.15.8 takeVerb.doDefault (2.1)

The doDefault method in takeVerb has been corrected so that it doesn't return the contents of a closed object. In previous versions of adv.t, "take all from object" would succeed even when the object was cl osed.

1.15.9 vehicle manipulation from within (2.1)

The vehicle class has been corrected so that the player cannot generally manipulate the vehicle while occupying it. For example, the player can no longer take a vehicle or put it anywhere while inside it.

1.15.10 initRestart (2.1)

A new function, initRestart(param), has been added. This function is used when adv.t calls the restart() built-in function (to start the game over from the beginning) in response to a "restart" command. The initRestart() in adv.t simply sets the property global.restarting to true. Your game can inspect this flag in your init function (or elsewhere) to take a different course of action when restarting a game than when starting up for the first time. The parameter is not used by adv.t implementation of the function, but if you replace the initRestart function defined in adv.t, you might find the parameter useful for passing information through the restart.

1.15.11 restartVerb (2.1)

The "restart" verb passes a pointer to the initRestart function when it calls the restart() built-in function. This causes initRestart to be invoked after the game has been restarted, but before the system calls init. Note that the call to restart() passes global.initRestartParam as the parameter to the initRestart function. If you replace initRestart with your own implementation, and you need to pass some information to this function from the game running before the restart, simply store the necessary inf ormation in global.initRestartParam at any time before restarting, and the information will automatically be passed to your initRestart function.

1.15.12 pluraldesc

We've added the new property pluraldesc to the class thing in adv.t. The definition simply adds an "s" to the end of the sdesc property. This new property is used by listcont() when multiple equivalent objects are present in a list; for details, see the information on the changes to listcont() described in the section on indistinguishable objects (page 10).

1.15.13 listcont, itemcnt–indistinguishable objects

The listcont and itemcnt functions in adv.t have been enhanced to list multiple occurrences of the same indistinguishable item only once, with the number of such items. See the section on indistinguishable objects (page 10) for more information.

1.15.14 "abort" usage changes

All of the system verbs in adv.t that use the abort statement have been modified slightly to make it easier to augment their behavior with the modify statement. All of the processing other than the abort has been moved out of the do*Verb* (or action) method, and put into a new method. For example, the action method in saveVerb now looks like this:

```
action(actor) =
{
        self.saveGame(actor);
        abort;
}
```

The new method saveGame in saveVerb now performs all of the processing that the action method previously performed.

The benefit of this change is that you can modify the saveGame method, and inherit the original behavior, without having to worry about an abort statement interfering with the order of operations. For example:

```
modify restoreVerb
        restoreGame(actor) =
        {
                // restore the game, and check for success
                if (inherited.restoreGame(actor))
                {
                        // re-randomize the puzzle
                        "The carnival hawker flashes a mischievous
                        smile at you.  \"There's no use trying to
                        guess the answer\", he says. \"I changed
                        around the shells while you were busy
                        restoring!\"";
                        puzzle.answer := rand(100);
                }
        }
;
```

1.15.15 fmtMe

The format mask fmtMe has been added. You can now use "%me%" in messages to refer to the actor. For basicMe, fmtMe is set to the message "me"; for other actors, it's set to the actor's thedesc. In adv.t, thing.ldesc has been changed to use "%me%": "It looks like an ordinary ¡¡sdesc¿¿ to %me%". This makes the default sentence somewhat more adaptable if you ask another actor to describe something:

```
>guard, look at the card

It looks like an ordinary card to the guard.
```

1.15.16 again, wait, sleep are darkverbs

The verbs “again”, “wait”, and “sleep” are now all darkVerb subclasses. None of these verbs logically requires any light (in fact, one could argue that “sleep” works better in the dark).

1.15.17 “there” = “it”

The specialWords definition in adv.t now adds “there” as a synonym for “it”. This allows commands such as: “Take the box and put the ball in there”.

1.15.18 isqsurface correction

The showcontcont() function no longer displays the contents of an object with the isqsurface property set to true.

1.15.19 seethruItem

We added a new class, seethruItem, which the player can look through. This is intended for objects such as windows or magnifying glasses. If the player looks through the object (with a command such as “look through the window”), t he object calls its thrudesc method to display an appropriate description of what the player sees. You should customize this method to display an appropriate message for each seethruItem you create; the default method displays “You can’t see much through the window”, which obviously won’t be what you wanted.

The class thing previously defined a thrudesc method as described above. To preserve compatibility with any existing games that depended on the presence of this method in thing, we left this as it was.

Note that seethruItem is *not*the same as transparentItem. The class transparentItem is for objects whose *contents*are visible, whereas seethruItem is for objects that the player can look through. Use seethruItem for windows, binoculars, magnifying glasses, and other non-containers. Use transparentItem for glass bottles and other transparent containers.

1.16 DOS Color Customization (2.1)

The player can now customize the colors used by the DOS runtime. A small new program, TRCOLOR, lets the player select the colors to use while running a game. The program is self-explanatory: it displays instructions on-screen, and uses the arrow keys to select colors. The program lets a player customize the colors to use for the main text area, highlighted text, and the status line. Type TRCOLOR at the DOS prompt to run the program.

Once you've selected your color scheme, the TRCOLOR program creates a small file called TRCOLOR.DAT in the current directory. The runtime reads this file at the start of subsequent game sessions.

Note that you can use multiple TRCOLOR.DAT files, in the same way that you can use multiple CONFIG.TC files. The runtime looks for TRCOLOR.DAT first in the current directory; if no such file exists, the runtime uses the TRCOLOR.DAT in the directory contain ing TR.EXE. This allows you to set up a separate color scheme for each game you're playing, and in addition set up a default color scheme for games with no color schemes of their own.

1.17 Improvements to MAKETRX

We've made a couple of changes to the MAKETRX program (on Macintosh, this program is called the "Executable Game Builder"). First, in version 2.1, we simplified the command line for the DOS version by letting you leave off arguments when MAKETRX is able to determine a suitable default. Second, in version 2.2, we added the ability to "bind" a set of run-time configuration options into your game executable–both the DOS and Macintosh versions provide this new feature.

1.17.1 DOS Default Arguments (2.1)

In version 2.1, we improved the user interface of the DOS version of MAKETRX. For compatibility with existing makefiles, the old command line syntax is still allowed; however, you can now omit most of the arguments, and MAKETRX will provide suitable defaul ts.

First, you can omit the filename extensions for any of the arguments. The extension assumed for the TADS runtime executable is ".EXE"; for the game file it is ".GAM"; and for the output (executable) file it is ".EXE".

Second, you can now omit everything except the name of the game file, and the program will attempt to provide defaults. If you omit the name of the TADS runtime executable, MAKETRX looks for a copy of TR.EXE in the same directory as the MAKETRX program; so , if you simply keep all of your TADS executables in a single directory, you don't need to specify the location of TR.EXE when running MAKETRX. If you omit the destination filename, MAKETRX will use the same name as the game file, with the extension replac ed by ".EXE".

The new command line formats for MAKETRX are:

```
maketrx mygame
```

Converts MYGAME.GAM into MYGAME.EXE, using the copy of TR.EXE that resides in the same directory as MAKETRX.EXE.

```
maketrx mygame myprog
```

Converts MYGAME.GAM into MYPROG.EXE, using the copy of TR.EXE that resides in the same directory as MAKETRX.EXE.

```
maketrx c:\tads2\tr mygame myprog
```

Converts MYGAME.GAM into MYPROG.EXE, using C:
TADS2
TR.EXE as the runtime executable.

1.17.2 Options File Binding

The MAKETRX command (the Executable Game Builder on the Macintosh) now takes an additional parameter that lets you specify the run-time command options that should be used when the game is executed. MAKETRX still accepts the original command formats; using one of the old-style command formats will not bind any command options into the resulting .EXE file, in which case the game will use the default runtime settings.

To specify command options for your game executable, you must first create a file containing the command options. Use the same format as CONFIG.TR–simply enter your options into the file as you would on the TR command line; separate options by newlines or spaces. For example, to specify a minimal cache size and a swap file of SWAP.DAT, you could make your CONFIG.TR file look like this:

```
-m0 -tf swap.dat
```

For DOS users: Once you've created a file with your command options, specify that file to MAKETRX by using it as the first parameter on the command line, prefixed with an "at" sign (@):

```
maketrx @config.tr mygame
```

The @config option can be used with any of the original command formats for MAKETRX. Once the options file is bound into your executable, its options will be used every time a player runs your game's .EXE file.

For Macintosh users: an extra dialog has been added at the end of the process of building the application. This new dialog asks you for the options file you want to bind into your game. If you don't want to include an options file, simply hit "Cancel". Otherwise, select the file containing the options and hit "OK". Note that the options file for the Macintosh version is in the same format as for the DOS version; refer to the *Author's Manual* for information on the format of the run-time command options.

Note that you may should avoid specifying anything specific to your system in your options file, such as drives or directories, since doing so may prevent the game from working properly on someone else's system. For example, if you specify the swap file as "D:
SWAP.DAT", the game won't work on a system that doesn't have a "D:" disk, since TADS will be unable to create a swap file on a drive that doesn't exist.

1.18 Miscellaneous Runtime Improvements

1.18.1 DOS: Ctrl-Left and Ctrl-Right Arrow Keys

The DOS runtime now lets the player move the cursor to the start of the next word wit control-right arrow, and to the start of the previous word with control-left arrow. This is consistent with many other DOS command-editing environments.

1.18.2 DOS Plain ASCII mode

The DOS runtime now has a plain ASCII mode, in which it uses only DOS character input/output functions. When in plain ASCII mode, the DOS runtime won't use any low-level I/O functions, such as BIOS or memory-mapped video, but goes entirely through DOS. Th e runtime also won't display the status line or any highlighting. If you have a special need for DOS-only I/O (for example, if you want to use a voice synthesizer to read the text generated by the game), you can use this mode.

To tell the runtime to use plain ASCII mode, specify -plain on the runtime command line:

tr -plain deep

You can also specify -plain to an executable game, as long as the game executable was built using a version of the runtime (such as 2.2) that supports the plain ASCII mode:

deep -plain

1.18.3 Hyphenation Improvements

The output formatter's handling of hyphenation has been improved. In particular, the output formatter will not split multiple consecutive hyphens across lines; if you use two or three hyphens together as a dash, the formatter will no longer split these up across lines. In addition, the formatter will keep hyphens at the end of a line, and will not break a line in such a way that it starts with a hyphen.

The Macintosh formatter has been fixed so that it no longer inserts spaces after hyphens when the spaces weren't in the original text.

1.18.4 Escapes in askfile Prompts

Prompt strings passed to askfile can now contain
n and
t sequences; these sequences are converted properly for display in the dialog.

1.18.5 DOS file selector bug fix

The DOS file selector has been corrected so that the Alt-key mappings are correct. In a previous version, the Alt-key mappings were off by one (Alt-B selected the A drive, for example).

1.18.6 Macintosh File Selector "All" Button

The Macintosh runtime's initial file selector dialog, which lets you specify the game you want to run, now has an "All Files" button. Clicking this button makes the file selector show *all* files, rather than just files that have the special signature that indicates they were created by the Macintosh version of the TADS compiler. If you've transferred a .GAM file from a non-Macintosh system, you can use the "All Files" button to o pen the file–you no longer have to use ResEdit or another file utility to set the creator and type flags in the file.

1.18.7 Status Line on Startup

The runtime system (on both DOS and Macintosh) no longer displays anything as the game is being initialized. In previous versions, the runtime displayed "TADS" at the left end of the status line, and "0/0" at the right end, until the game changed the displ ay. This was slightly incongruous for games that changed to a different status line format. The status line is now completely blank during game initialization.

Note that this change may have a slight effect on your game: by default, the score/turn area of the status line will be completely blank until a call to setscore() is made. To compensate for this change, we've added a call to scoreStatus(0, 0) at the beginning of the init function defined in adv.t. If you provide your own init function, you may want to initialize the status line's score display in your init; if you don't, the status line's score/turn area will be empty until the player enters the first command, at which point the normal turn-counting daemon will usually update the status line.

1.19 Compiler Exit Codes

The compiler now sets system exit codes according to whether any errors occurred during compilation. If there are no errors (even if there are warnings), the compiler sets the exit code to 0. If any errors occur, the compiler sets the exit code to a non-z ero value. This makes the compiler work better with "make" utilities, since these utilities usually check the exit code of each command to determine if the operation encountered an error.

In addition, the compiler will no longer generate a .GAM file if any errors occur. In the past, the compiler generated a .GAM file in most cases, even when the source file contained errors. Attempting to run a .GAM file produced from a source file with er rors had unpredictable results.

The Macintosh compiler now displays a different message if a compilation fails or is interrupted than it does when the compilation completes successfully. The compiler displays "Compilation Terminated" if errors occur; it still displays "Compilation Completed" upon successful completion without errors.

Chapter 2

The New Book of the Parser

Chapter Four of the *TADS Author's Manual* describes the player command parser as it was in version 2.0. Since then, we've added substantial new functionality to the parser. For the past few releases, we described the new features as they we re added, but until now there was no single source of information on the parser; as a result, it was often difficult to see how the new features and the original features worked together. This section cover all of the parser's features–new and old–and is intended to replace Chapter Four of the Author's Manual.

If you were going to write a text adventure game, and you didn't have a system such as TADS, you would be faced with the daunting prospect of designing a player command parser. The job is so large that you'd probably spend more time designing and implementi ng your parser than on any other part of the game.

Fortunately, if you're taking the time to read this, you're at least thinking about building your game with TADS, which means that you won't have to write your own parser. Using an existing parser will save you a vast amount of work, but won't eliminate your work entirely: you won't have to write your own parser, but you w ill have to learn enough about the parser you're using to write your game.

With TADS, the amount you need to learn depends on what you want to do. You can do quite a lot with TADS without learning anything at all about the parser–by using existing class libraries (such as the standard library adv.t supplied with TADS), you can cr eate objects whose interactions with the parser have already been defined, so that all you need to do is fill in some simple information on the objects, such as their names and descriptions. Once you've become comfortable with the basics of writing TADS gam es, though, you'll want to create your own object classes, with their own interactions with existing verbs; to do this, you'll have to learn a little bit about how the parser processes commands, so that you can add code to your new objects that lets them re spond to certain commands. You'll probably also want to be able to create your own verbs, which requires that you learn how to define your own commands. After you've done all of this, you may want to start making more complicated changes, which will require that you learn about some of the more esoteric aspects of the parser.

2.1 Parsing Fundamentals

The first part of this chapter is an overview of what you need to know about parsing to get started with TADS: how to create objects and give them names the player can use in commands; how to make customize the way your objects respond to commands; and how to create your own new commands. One of the nice features of TADS is that you can write games almost immediately, and fill in your knowledge with more detail as you go. This section is intended to help you get started–we begin with the minimum information you need, and then move on to the more advanced features.

2.1.1 Using Pre-Defined Objects

The very least you have to learn about the parser to write a game is: nothing! That's because you can write a game using only objects that have been defined already–for example, you can use the objects defined in adv.t, the library of common adventure ob jects included with TADS. To create a box, which can be opened and closed (but starts off open), you would only have to write this code:

```
box:  openable
        sdesc = "box"
        noun = 'box'
        location = startroom
        isopen = true
;
```

The class openable (defined in adv.t) has already been set up to respond to the commands appropriate for a container that can be opened and closed–it automatically knows how to respond to all of these commands from a player:

```
>open the box

>look in the box

>put everything in the box

>the the ball out of the box

>put the book in the box, and close the box
```

Although we said you didn't have to know anything about parsing to use objects defined in adv.t, you actually have to know a little bit about parsing: you have to know how to give an object a name.

In the object above, we defined a property called noun, which we set to 'box'. The noun property is special–it's a *vocabulary* property. Vocabulary properties assign names to an object that the player can use in commands; the vocabulary properties are noun, adjective, verb, preposition, article, and verb. For the most part, you will define noun and adjective vocabulary properties; you won't need to use the others until you start defining new commands of your own. Note a very important feature about vocabulary properties: the value of a vocabulary property must always be a single-quoted string.

Theorem 1. ***A vocabulary string must be in single quotes.***

Vocabulary properties are different from normal properties. Their only function is to define words that the player can use to refer to the object. Unlike ordinary properties, vocabulary properties can't be used from within your game program. For example, this line of code won't display anything at all:

```
say(box.noun);
```

The property box.noun simply has no value at run-time. You can't use the vocabulary properties to get the vocabulary words for an object at run-time; however, you *can* get these properties using the built-in function getwords().

When you assign a vocabulary property to an object, you create a word that the player can use in a command, and you assign the word a particular "part of speech". A word's part of speech determines how the word can be used in a sentence, the same way it do es in ordinary speech. When you define a word in a noun property, you're specifying that the word can be used in a player's commands wherever a noun is grammatically valid. You're also specifying that the word refers to this particu lar object.

The same word can be assigned to multiple objects. For example, you may have several objects in your game that all have the noun "box". In addition, the same word can be used as multiple parts of speech. For example, you might have one object in your game called a "paper towel", and another called a "term paper"; the word "paper" in this case can be used as either a noun and an adjective.

You can also define more than one of the same part of speech for an object. In many cases, you will want to provide several synonyms for an object's words. You can do this easily–simply list all of the synonyms in the vocabulary property for each part of s peech. For example, if you have a booklet in your game, you may want to provide a number of similar words that the player can use to refer to the object:

```
booklet: item
        location = startroom
        sdesc = "booklet"
        ldesc = "It's a small booklet labeled
                \"TADS Release Notes\". "
        noun = 'book' 'booklet' 'notes'
        adjective = 'tads' 'release' 'small'
;
```

Notice that each word in each list is enclosed in single quotes. With this definition, you could refer to the booklet as "release notes", or "small tads booklet", or any other combination of these adjectives and nouns.

2.1.2 Responding to Commands

Adv.t defines many types of objects that are common in adventure games, and you will probably be able to write a substantial part of your game using only these objects. However, a game that only involved objects from adv.t would be very dull, so you'll soon want to be able to create objects of your own that respond in novel ways to certain commands. To do this, you'll need to understand how verbs are applied to objects.

Each command that the parser understands has a set of methods that it calls in the objects making up the command (a method is simply a function or subroutine that's associated with a particular object). When the player types a command that involves a verb a nd a noun, the parser figures out which methods to call based on the verb, and then calls these methods in the object specified by the noun. For example, the verb "open" has two methods that it calls in the object being opened: it has a *verification* method named verDoOpen, and an *action*method called doOpen. (The "do" in these names stands for "direct object"–it's not meant to be the verb "do".) The verification method is used to determine if the object is logical with the verb–for "open", an o bject is logical if it can be opened and closed in the first place, and if it's not already open. The verification method doesn't have to test to see whether the object is accessible to the player, because this testing is done separately; this method only h as to check to determine if the object makes sense with the verb. The action method carries out the verb; it doesn't have to check to see whether the object makes sense with the command, since the verification method has already done this.

Suppose you wanted to make an object that does something unusual when opened. For example, you might want to have a can that contains a spring-loaded snake that jumps out when the can is opened. To do this, you'd customize the can's handling of the "open" c ommand.

If we make the can an openable, as we did with the box, we'd get the automatic handling of opening and closing the can. However, we want some additional behavior–when the object is opened, and it contains the snake, we want the sna ke to jump out. To do this, we'll *override* the verDoOpen method that's defined by the openable:

```
snakeCan:  openable
        sdesc = ''can''
        noun = 'can'
        doOpen(actor) =
        {
                inherited.doOpen(actor);
                if (snake.isIn(self))
                {
```

```
                    " As you open the can, a spring-loaded
                    fake snake leaps out!  You're so
                    surprised that you drop the can''.;
                    self.moveInto(actor.location);
                    snake.moveInto(actor.location);
                }
        }
;
```

The first thing the new action routine does is to *inherit* the default doOpen that was defined by openable in adv.t. When overriding a method, it's often useful to be able to incorporate the original method from the class–the inherited keyword lets you do this. The inherited default doOpen opens the can as usual. After that's been done, we check to see if the snake is in the can; if so, we take our special action: display a message telling the player about the snake, then moving both the can and the snake to t he player's location.

There are a couple more things to notice in this example. One is the parameter actor. Whenever a verification or action routine is called, the system includes the actor to whom the command was directed as an argument. When the player types a command without specifying an actor, the player's actor Me is used by default–Me is a special object that refers to the player. You can always find out what room the player is in with Me.location, and you can find out what the player is carrying with Me.contents. However, by writing our doOpen method in terms of the actor object passed as a parameter, we can make it possible for the player to direct commands to other characters in the game, and have the commands work properly. So, rather than moving the snake and the can to Me.location, we moved them to actor.location (which will be the same thing if the player was opening the can him or herself).

If you wanted to make the command even more adaptable to another character, you should replace some of the text in the message with special "format strings":

```
" As %you% open the can, a spring-loaded
fake snake leaps out!  %You're% so
surprised that %you% drop%s% the can".;
```

The special sequences in percent signs, such as "%you%", are converted by the system automatically to an appropriate word based on the current actor. If the command is being carried out by the player actor, the message displayed is the same as the original; however, if the command is directed to another character in the game, the terms are all replaced by something appropriate to that character. In most games, the non-player characters can only carry out certain selected commands, so you only need to worry about this when you're writing a command that can be directed to a character. All of the command handlers in adv.t are written this way, but you only need to worry about the ones you know can be carried out by a non-player character.

Note that we only had to change the action (doOpen) routine–we didn't do anything to the verification (verDoOpen) routine, because the one we inherited from container already does what we need.

Now suppose we wanted to handle another verb; but this time, it's one that the object doesn't inherit from its class. Let's suppose that we want to make it possible to read the can's label, but only after the can has been cleaned. For this, we'll need two new verbs, "read" and "clean", both of which are defined in adv.t. As with "open", these verbs each have a verification and an action method. We'll add the following code to the can:

```
isClean = nil
verDoClean(actor) =
{
        if (self.isClean)
                "You can't make it much cleaner.";
}

doClean(actor) =
{
        "You wipe off the label. It's much more
        readable now.";
        self.isClean := true;
}

verDoRead(actor) =
{
        if (not self.isClean)
                "It's far too dirty.";
```

```
}

doRead(actor) =
{
        "It's labeled \"Snake in a Can.\"";
}
```

The verification routine is a little strange, because it doesn't seem to do anything–all it does is check to see whether the command makes sense, and displays a message if not. This is, in fact, all that a verification routine should *ever*do. The most important thing to remember about verification routines is that they must *never* modify the game's state in any way–for example, they must never set a property's value. Only the action routine should change game state.

Theorem 2. ***Never modify any game state in a verification routine.***

The reason that verification methods shouldn't change any game state is that these methods are sometimes called "silently" by the parser; that is, the parser turns off the output, and calls a verification method. The player will never know that the method h as been called at all, because any text it displayed was suppressed–this is what we mean when we say that the method is called "silently". The parser makes these invisible calls to verification methods when it's trying to figure out which of several possi ble objects the player means. For example, suppose the player is in a room with two doors, one red and one blue, and the red door is open and the blue door is closed; if the player types "open door", the parser calls the verification routine on each door to determine which one makes the most sense. The red door's verification method will display a message saying that you can't open it because it's already open; the blue door's method will say nothing. These calls are made with the text output turned off, beca use the parser isn't really trying to open anything at this point–it's merely trying to figure out which door makes more sense with the command. Once it's decided which one to open, the parser will run the verification method with output turned on, and the n it will run the action method.

How does a verification method tell the parser that the command is not logical? Simple: it displays a message. So, the verification routine's only function is to display an error. If the verb is logical for the object, the verification method doesn't do a nything at all–since the verb makes sense, there's no error to display. If the verb does *not*make sense, the verification method displays an error message telling the player why the verb can't be used. The parser sees the message, and knows that it indicates that the verb can't be used, so it won't try to run the action routine.

There's another important feature of the verification method: if an object doesn't have a verification method defined at all–which means that the object neither defines the verification method itself nor inherits the method from a superclass–the command automatically fails, and the parser generates the message "I don't know how to *verb* the *object*". So, if you want an object to be able to carry out a command, you *must*either define the verification method directly in the object, or make sure the object inherits the verification method from a class.

2.1.3 Creating New Commands

Once you start writing your game, you'll quickly want to start adding your own new commands. One of the features that makes TADS so powerful is that it has *no*built-in verbs–all of the verbs you've seen so far are defined in adv.t, not within TADS itself, so you can change, replace, or remove any of them, and, of course, add new verbs of your own.

We've already mentioned that there's a vocabulary property called verb; as you might guess, this is the vocabulary property that you use to add a new command. You may wonder what sort of object you attach the verb property to. The answer is that there's a special kind of object defined in adv.t, called deepverb, that you can use as a command object. This isn't an object that the player sees as part of your game–it's merely an abstract, internal object, whose purpose is to contain the data and methods that define a new command.

TADS allows you to create three basic types of commands: a command that consists simply of a verb, such as "look" or "jump"; a command that has a verb and a direct object, such as "take book" or "turn on the flashlight"; and a command that has a verb, a di rect object, a preposition, and an indirect object, such as "put the ball in the box" or "clean the lens with the tissue". The same word can be used in more than one form of command; for example, you might define commands such as "lock door" and "lock door with key".

For the first type of sentence, which has only a verb, you define the entire command in the deepverb object. You specify the action that the verb carries out with a method called action that you define in the deepverb object. For example, to define a verb called "scream" that displays a message, you could do this:

```
screamVerb: deepverb
        verb = 'scream'
        sdesc = "scream"
        action(actor) =
        {
                "Aaaaaiiiiieeeee!!!\n";
                "(You feel much better now.)";
        }
;
```

The presence of the action method in the object that defines the verb allows the command to be used without any objects. If you define a deepverb that doesn't have an action method, and the player types the verb without any objects, the parser asks the player to supply a direct object, because the parser can't execute the command without any objects when no action method is present.

For the second type of sentence, which has a verb and a direct object, you must set up a deepverb object–but, as you have already seen, the command is carried out by the direct object, not by the verb itself. The deepverb object serves to tell the parser about the verb and how it works–it contains the verb property, and also has the doAction property. The doAction property is a special property that tells the system that the command can be used with a direct object (the "do" in doAction stands for "direct object", just as it did with methods such as doOpen), and tells the system what the verification and action methods are called.

Here's how it works: you define the property doAction in the deepverb as a single-quoted string. The parser takes the string 'verDo' and puts it in front of your string–this becomes the name of the verification property. The parser then takes 'do ' and puts it in front of your string–this becomes the name of the action property. For example, if you define a verb to scream at someone, and you specify doAction = 'Screamat', then the verification method for your new command is verDoScreamat, and the action method is doScreamat. Here's how your deepverb definition would look:

```
screamatVerb: deepverb
```

```
        sdesc = "scream at"
        verb = 'scream at'
        doAction = 'Screamat'
;
```

If the player types "scream at the phone", the parser first calls the verDoScreamat method in the phone object, to verify that the command is sensible; if the verification method is defined, and succeeds (which means that it doesn't display any messages), the parser proceeds to call the doScreamat method, which should carry out the action of the command.

There are two more things to notice in this example.

The first is that the verb property we defined has two words. This is a special ability of the verb property–you can't do anything similar for noun, adjective, or any of the other vocabulary properties. If you define a two-word verb, you must separate the two words by a space, and you must define the second word as a preposition. (In this case, "at" is already defined as a preposition in adv.t, as are most English prepositions. If you add a two-word verb, be sure that the second word is defined as a preposition; if it's not so defined in adv.t, you must define it in your game's source code.) Two-wor d verbs are useful, because many English verbs take this form. When a verb has a direct object, TADS lets the player use the second word immediately after the verb, or at the end of the sentence. With "scream at", the latter doesn't make much sense, but for many verbs it does–for example, you might say "pick up the book", but it would be equally valid to say "pick the book up". TADS understands both.

The second thing to notice is the use of capitalization in the doAction string. The convention used by adv.t is to capitalize the first letter of a preposition, but *only for verbs that take an indirect object*. For example, adv.t defines an ioAction of GiveTo for the command "give *object*to *actor*"; the To is capitalized because "to" is the preposition that introduces an indirect object. For our verb here, we don't have an indirect object–the preposition is part of the verb, rather than a word that introduces an in direct object–so we do *not* capitalize the "at" in Screamat. You don't have to use this convention, since the string that you define in a doAction is entirely up to you, but it may help you to make sense of adv.t if you know the convention it uses.

The third type of sentence has both a direct object and an indirect object, and has a preposition that introduces the indirect object. For example, you might want to create a command such as "burn paper with torch". To define this command, you use the ioAction property of the deepverb. This property is similar to doAction, but it defines *three* new methods, and also associates a preposition with the command:

```
burnVerb: deepverb
        sdesc = "burn"
        verb = 'burn'
        ioAction(withPrep) = 'BurnWith'
;
```

Notice that 'BurnWith' follows the capitalization convention described earlier. Since the preposition introduces an indirect object, and isn't merely a part of the verb, we capitalized its first letter.

The association of the preposition is done with the withPrep in the parentheses–withPrep is an object, defined in adv.t, which specifies a preposition property that defines the vocabulary word "with". The withPrep object is of class Prep, which is similar to deepverb: it's an abstract type of object that isn't visible to a player, and its only function is to define the vocabulary word and associate it with an object that can be referenced from within your game program.

This definition tells the parser that sentences that start with the verb "burn" must have a direct object, and an indirect object, and that the objects are separated with the preposition "with". When a sentence matching this pattern is entered by the playe r, the parser will use this verb definition to execute the command.

We mentioned that an ioAction defines three methods. Recall that doAction defines two methods from its root string: a verification method (starting with verDo), and an action method (starting with do). The ioAction property similarly defines verification and action methods, but since two objects are involved, it must generate not one but two verification methods: one for the direct object, and anothe r for the indirect object. The methods are called verIoBurnWith and verDoBurnWith. As you might guess from the names, verIoBurnWith is sent to the indirect object, and verDoBurnWith goes to the direct object–and they're called in that order, with the indirect object going first. If *both*verification methods are defined and successful, then the parser calls the action method, ioBurnWith–this is called only for the indirect object. The parser never calls an action method in the direct object for a two-object command.

The verification and action methods for verbs with two objects are a little different than those for single-object verbs. Whereas a single object verb's verification and action methods only take the actor as a parameter, two of the two-verb methods get an a dditional parameter: the other object. The remaining method only gets the actor. The definitions should look like this:

```
verIoBurnWith(actor) = { ... }

verDoBurnWith(actor, iobj) = { ...}

ioBurnWith(actor, dobj) = { ... }
```

This may seem strange, but there's a reason. Since the parser calls verIoBurnWith first, it doesn't always know what the direct object is when it calls verIoBurnWith. As a result, it can't provide it as a parameter. By the time it calls verDoBurnWith, though, the indirect object is known, so the extra parameter is provided. ioBurnWith is called later still, so the extra object is also provided as a parameter there.

Theorem 3. ***verIoXxxx does nothave a dobj parameter.***

You may notice that there are some methods defined in adv.t that might make you think that the parser also calls doBurnWith. In fact, the parser will never call doBurnWith itself, but this is a perfectly valid method name which you can call from your own code. Sometimes, you will want the action of a two-object verb to be carried out by the indirect object, and other times you'll want to us e the direct object. Which way is better depends entirely on the situation–in general, the better way is the one that requires you to write less code. If you find a situation where you want to handle a two-object command's action in the direct object rathe r than in the indirect object, simply write an indirect object action routine that looks like this:

```
ioBurnWith(actor, dobj) =
{
        dobj.doBurnWith(actor, self);
}
```

The parser will never call doBurnWith, but there's nothing to stop you from calling it.

Theorem 4. ***The parser never calls doXxxx for a two-object verb.***

If you find yourself in a situation where the standard order of verification for a two-object verb is wrong, TADS does let you change this default ordering. We're moving into advanced territory here, so you probably won't need to know this for quite a while , but you might at least want to note that it's possible. When you define an ioAction, you can include a special modifier that tells TADS to change the ordering:

```
ioAction = [disambigDobjFirst] 'BurnWith'
```

The modifier is the strange-looking word in square brackets; it tells TADS that you want to reverse the normal ordering. When you include this modifier, the parser will process the verification and action methods in a different order than usual:

```
verDoBurnWith(actor) = { ...}

verIoBurnWith(actor, dobj) = { ... }

ioBurnWith(actor, dobj) = { ... }
```

Note that the parameters changed along with the order. Since the direct object is processed first for this command, the direct object verification routine is called before the indirect object is known, and hence no indirect object is passed to it; and since the indirect object verification routine is called after the direct object is already known, the direct object is supplied as a parameter to this routine.

2.1.4 Adding to an Existing Verb

As you add your own new commands, you will probably find that you want to augment the definition of one of the verbs defined in adv.t. For example, you may want to add a new command such as "open crate with crowbar". Since a verb for "open" is already defi ned in adv.t, you can't create another "open" verb in your own game. Fortunately, TADS lets you supplement an existing verb without having to change the original definition's source code.

To change an existing verb, you can use the TADS modify statement. This statement lets you add properties to an object that has already been defined elsewhere in your game (even in a separate file that you include, such as adv.t). For example, to modify the "open" verb defined in adv.t to add an "open with" command, you would do this:

```
modify openVerb
        ioAction(withPrep) = 'OpenWith'
;
```

We strongly recommend that you use the modify mechanism whenever possible, rather than making source changes to adv.t. If you edit adv.t directly, you may find that you have a lot of work to do when you upgrade to a new version of T ADS, because we will probably modify adv.t–if you also modify it, you will have to figure out how to apply your changes to the new version of adv.t If you use the modify mechanism instead, you should be able to use future versions of adv.t without any additional work.

Theorem 5. ***Don't edit adv.t directly–use modify instead.***

2.1.5 Overview of Disambiguation

As you build your game, you will probably add multiple objects that all go by the same name. When you give the same noun to two or more objects, you create the possibility that a player can type in a command with an ambiguous object name–that is, the words that the player uses refer to more than one object. For example, if you create several books in your game, you will probably give them all the noun "book"; a command such as "take book" could refer to any of these objects. Fortunately, the TADS parser has a powerful system that can resolve ambiguous object names; in many cases, the parser can disambiguate objects automatically, without needing any help from the player.

The first step in disambiguating an object name is to apply the visibility and access rules. These rules, which are defined by objects in your game program, determine whether an object is visible to the player, and if so whether it can be used with a partic ular verb. If an object is neither visible, nor accessible for use by the player with the given verb, the object isn't considered further. If it's visible but not accessible, it will be considered only to the extent that there isn't another object which *is* accessible.

Visibility is determined by the method isVisible, which is defined for each game object. The general-purpose class thing defined in adv.t provides a definition of this method suitable for most objects; you will probably not need to make any changes to this method unless you need some special effect. To determine if an object is visible to the player, the parser makes this call:

```
object.isVisible(Me)
```

Accessibility is determined by the deepverb object. For a direct object, the parser first calls the method validDoList, which returns a list of all objects which are *possibly* valid for the verb; the objects in the list aren't necessarily valid, but any objects *not*in the list are *not*valid. If validDoList returns nil, it means that *all* objects are possibly valid and should be considered. A similar method, validIoList, returns a list of possibly-valid indirect objects.

After eliminating any objects that aren't in the validDoList (or validIoList, as appropriate), the parser submits each surviving object to the method validDo, also defined in the deepverb object. This method takes the object as a parameter, and returns true if the object is valid for the verb, nil if not. The parser keeps only objects that are valid.

If any objects remain, the parser turns off output and calls the appropriate validation method in each object. This is the "silent" validation call that we mentioned earlier. If only one of the objects passes the validation test (that is, the validation met hod doesn't attempt to display any messages), that object is chosen. Otherwise, the parser gives up and asks the player which of the objects to use.

The silent validation test lets TADS be very intelligent about disambiguation. For example, if there's an open door and a closed door in the room, and the player types "open door", the validation test lets the system determine that the player means to open the closed door–since the verDoOpen method fails for the open door (it displays "It's already open" if the default verification method from adv.t is used), but succeeds for the closed door, the parser knows to choose the closed doo r.

The second half of this chapter, "The Parser in Detail", has full details on how disambiguation works.

2.1.6 Default objects and "All"

The parser also tries to be intelligent about supplying defaults when the player provides only partial information. The parser supplies default direct and indirect objects for commands that omit one of these objects, using rules defined by the objects in yo ur game program.

To determine a verb's default direct object, the parser calls the method doDefault in the deepverb object. This method returns a list of default direct objects. The verbs in adv.t generally return a list of all accessible objects; a couple of verbs return more restricted lists (f or example, "take" returns a list of accessible objects that aren't already in the player's inventory). Similarly, the parser calls the deepverb method ioDefault to get a list of default indirect objects. When doDefault (or ioDefault, as appropriate) returns a list that consists of exactly one object, the parser uses this object as the default for the command. If the list is empty, or consists of more than one obj ect, the parser will ask the player to supply the object, because the command doesn't imply a particular object.

The parser's automatic default system can be especially convenient for indirect objects. Certain commands imply the use of particular indirect objects; for example, "dig" implies a shovel or similar tool. For such commands, it's a good idea to define ioDefault methods that return a list of accessible objects that satisfy particular criteria; for "dig", you would return a list of accessible objects that are usable for digging.

The doDefault method has another use: the parser calls these methods to determine the meaning of "all". When the player uses "all" as a direct object, the parser replaces "all" by the list returned by doDefault. Since the parser doesn't allow using a command to include multiple indirect objects, "all" cannot be used as the indirect object, hence ioDefault will never be used to supply a list for "all". Note that you can disallow using "all" (and multiple direct objects in general) for a particular verb by defining the property rejectMultiDobj to be true in the deepverb. Here's an example of using rejectMultiDobj to prevent the player from being able to look at everything in the room all at once:

```
modify inspectVerb
        rejectMultiDobj(prep) =
        {
                ''You can only look at one thing at a time''.;
                return true;
        }
;
```

2.2 The Parser in Detail

The rest of this chapter is devoted to the details of how the parser works. The sections are organized in roughly the same order as the steps the parser executes as it analyzes and executes a player's command.

When the player types a line of text into your game, the TADS parser goes through a series of operations to convert the command line into simple actions on objects in the game. In the course of these operations, the parser makes calls to your game program f or advice; in most cases, the parser has default actions for these calls, so you don't need to include any code in your game program for the default behavior. You can, however, customize the parser by providing definitions for these "hooks" in your game. Th e main purpose of this chapter is to describe the way the parser interacts with your game program, so that you can customize the parser by overriding the defaults. We also describe many of the internal operations of the parser so that you can understand how to make it behave the way you want it to.

2.2.1 Reading a Command Line

The first step in parsing is getting a command line from the player. The parser does this automatically, without any action by your game program, at the beginning of each turn. The parser displays its prompt, then waits for the user to enter a line of text.

The prompt is the first parser default action that you can override: if your game defines a function called commandPrompt, the parser will call this function when it needs to display a prompt; otherwise, it will display its default prompt, which is a greater-than sign (¿). The commandPrompt function is called with an argument that lets you display distinct prompts for different situations, if you wish; the default prompt is always the same, regardless of the type of input being requested, but you can use different prompts for the different types of input if you prefer. The commandPrompt argument values are:

0 Used for normal commands.

1 The player typed an unknown word, and the parser is reading a new command that can be an "oops" command. If the player types a command line that begins with the word "oops" (or "o"), the parser will substitute the next word for the unknown word in the pre vious command, and go back to parsing the previous command. Otherwise, the parser treats the new line as a brand new command.

2 The player has referred to an ambiguous object, and the parser wants the user to provide more information. The parser displays a message such as "Which book do you mean, the red book, the blue book, or the green book?", and then reads a command with prompt 2. If the player types something that looks like an answer to the question (for example, "the blue one" or "green book" or simply "red"), the parser uses that information to disambiguate

the object; otherwise, it treats the new line as a brand new command.

3 The player has typed a command that has no direct object, but the verb requires a direct object. The parser displays a message such as "What do you want to open?", then prompts with prompt 3. If the player responds with an object name, it is used as the d irect object of the original command, otherwise the text is treated as a new command.

4 The same as 3, but used to prompt for indirect objects.

After the player has typed a command, the parser will tell your game about it by calling your preparse function. Immediately after reading the command line, the parser calls this function with the original text of the player's command as the argument. If you don't provide a preparse function in your game program, the parser skips this step. If you do provide a preparse function, it can return one of three values: it can return true, in which case the command is processed as normal; it can return nil, in which case the command is skipped entirely, and the parser immediately asks the player for a new command; or it can return a string, in which case the returned string is used as the command instead of the line of text typed by the player.

Here's an example definition of commandPrompt that displays a long prompt for the first few commands, then changes to a shorter prompt. The longer prompt is not displayed when the parser asks the player a question.

```
commandPrompt: function(typ)
{
        if (global.promptTurns = nil)
                global.promptTurns := 0;

        "\b";

        if (typ = 0 or typ = 1)
        {
                global.promptTurns++;
                if (global.promptTurns < 5)
                        "What do you want to do now?\n";
                else if (global.promptTurns = 5)
                        "Aren't you tired of this long prompt?
                         I sure am.  From now on, the prompt
                         will just be like this:\n";
        }
```

```
        ">";
}
```

If the player enters a blank line in response to a command prompt, the parser calls a function called pardon in your game program. You must provide a function with this name. This function takes no arguments; its only purpose is to provide an opportunity for you to display an error message when the player enters an empty command. Here's an example:

```
pardon: function
{
        ``I beg your pardon?";
}
```

2.2.2 Breaking the command into words

Once the parser has read a line of text from the user, the next step is to break the command into individual words. The parser does this based entirely on the characters in the command line–at this stage, it doesn't pay any attention to whether it recogniz es any of the words. People who write compilers call this step "lexical analysis" or "tokenization", because it involves breaking up a string of characters into individual units (or "tokens") by classifying the characters and grouping related characters.

First, the parser converts the entire command to lower-case. Next, the parser simply goes through the command line, and takes each group of letters and numbers to be a word, and keeps any strings enclosed in double quotes together.

For example, assume the player types this:

```
Joe, Go West, then type ``hello, world!" on
the Acme-Tech computer's keyboard.
```

The parser converts this to a string of words and symbols:

```
joe
,
go
west
```

```
,
then
type
"hello, world!"
on
the
acme-tech
computer's
keyboard
.
```

Note that the punctuation marks are all treated as separate words, and the spaces are all ignored. Note also that apostrophes and dashes are considered to be equivalent to letters, so things like "Acme-Tech" and "computer's" are treated as single words.

2.2.3 Checking for Special Words

After breaking the command into words, the parser checks the word list for the special words listed in your specialWords directive. If your game doesn't provide a specialWords directive, the parser has a built-in list that it uses (since adv.t provides a specialWords directive, your game probably has one whether you've entered one or not). Any word in the word list that matches one of the special words is converted to a flag indicating that it is that special word.

Note that this conversion doesn't apply to the special word in the "of" slot. This special word is matched later, and not converted along with the other special words, because it is often useful for "of" to be an ordinary preposition, which wouldn't be poss ible if it were converted at this point.

As with the previous step, where the command was broken up into words, this step is done automatically by the parser without any calls to your game.

2.2.4 Assembling the Words into Commands

The next step is to find the individual commands on the command line. The command line can be made up of multiple commands, separated by the words "then" or "and", or by various punctuation marks: commas, periods, exclamation points, and question marks.

The word "then", and the period, exclamation point, and question mark are unambiguous–they always separate commands. So, the first thing the parser does is scan through the command line, looking for one of these symbols. If it finds one or more at the begi nning of the command, it simply discards them. Otherwise, it takes all of the words up to the first occurrence of one of these symbols, and treats that as a command. For example, consider this command:

```
Joe, go west, then open the door and the window and go east.
```

The parser takes the part up to the "then" as the first command. It keeps doing the same thing through the rest of the command, which means that it will at this point break the line up into two commands:

```
Joe, go west
```

```
open the door and the window and go east
```

Note that the word "and" and the comma are ambiguous, since they may separate entire commands, or objects within a single command. The parser delays any decisions about these symbols until it has further analyzed the command; so, although the second line ab ove actually consists of two separate commands, the parser at this point is considering it to be a single command.

The parser now processes these commands individually. It processes each part of the command in sequence, until either it has processed all of the commands, or an error occurs, or your game program executes an exit or abort command.

2.2.5 Checking for an Actor

Now the parser starts processing the individual commands making up the command line. The first thing it does is to check for an actor prefix. The player can specify that a command is to be given to a particular actor by putting the actor's name followed by a comma at the very start of a command. The sentence "Joe, go west" starts with an actor's name and a comma, so the command "go west" is directed to Joe.

Checking for an actor is the first processing of the command that involves the vocabulary words defined by your game (other than the special words).

To determine if the group of words in front of the comma is indeed the name of an actor, the parser looks at each word in turn, and applies the noun-checking rules; see the section on noun phrases (page 80) for details on these rules.

If the words do form a noun phrase, the parser will next attempt to determine if the noun phrase refers to an object that can be used as an actor. To do this, it first forms a list of all of the objects in the game that match all of the words in the noun ph rase. For example, if the sentence starts with "black knight", the parser builds a list of all of the objects in the game that have both the adjective "black" and the noun "knight". If the list has more than one object, the noun phrase is ambiguous, so the parser must "disambiguate" the object to determine what the player intended.

First, the parser goes through the list and notes which objects in it are visible. To do this, it calls the method isVisible(Me) on each object in the list–this method returns true if the object is visible from the given vantage point (Me, the player actor), nil if not. This step is used entirely to determine how to report any errors that happen later; an object may be visible, but not valid as an actor, in which case a different message is required than if the object isn't present at all.

Next, the parser determines whether each object can be used as an actor. For each object, the parser calls the validActor method, which returns true if the object can be used as an actor. The default definition of this method in adv.t returns true for any object if the object can be reached by the player. You can override this method to achieve different effects; for example, if the player is carrying a walkie-talkie, you could make any actors that have the other walkie- talkie valid as actors, since they can hear the player through the radio, even though they may not be in the same room.

Note that validActor is intended to determine whether an object is *valid* as an actor, not necessarily whether the object is *logical* as an actor. Hence, the validActor method defined in adv.t applies to *all* objects, not just actors. This method is used in determining whether an object can be addressed by the player at all; if a head of lettuce is present in the same room, the player should be able to address it, even though doing so may not do any good.

For any object which passes the validActor test, the parser notes whether the object is a "preferred" actor, by calling the preferredActor method on each object. This method returns true if the object is generally suitable for use as an actor, nil otherwise. The default definitions in adv.t return true for this method for all objects of class Actor, nil for other types of objects.

After removing all objects that failed the validActor test, the parser looks at its list to see what's left.

If no objects remain, the player has tried to talk to something that is not currently accessible as an actor, or which doesn't exist at all. If none of the objects in the original list were visible, the parser issues error 9: "I don't see any %s here". (Note that the "%s" is replaced with the text of the player's noun phrase. So, if the original command was "Joe, Go West", the error message reads: "I don't see any joe here".) The reason the parser uses the player's original words in the error message is that the parser can't determine what object the player is referring to–it can only use the words the player originally entered. Note that the parser always converts the player's words to lower-case.

If more than one object remains, the parser goes back and looks at the results of the preferredActor method for each object remaining. If any objects returned true from this method, the parser considers only those objects; if all returned nil, the parser ignores the preferredActor results and keeps the original list. If exactly one object returned true from preferredActor, the parser uses that object as the actor. Otherwise, it must ask the player which of the remaining objects was intended; this process is the same as for any other object, and is des cribed below.

If exactly one object remains, the parser uses that object as the actor. The process of determining the actor is completed.

2.2.6 Identifying the Verb

Once the actor (or absence of an actor) has been determined, the parser finds the verb. This is one of the simpler parts of parsing, because the verb must always be the first word of the command. The parser takes the first word and checks to make sure it ca n be used as a verb; if not, message 17 ("There's no verb in that sentence!") is displayed, and the command is aborted.

When you define a verb, you can specify one word or two. For example, the verb "take" is specified with a single word, whereas "pick up" is specified with two words. In English (and some other languages), a preposition can be associated with a verb in such a way that it effectively becomes part of the verb–the preposition's presence changes the meaning of the verb so much that the verb-preposition combination is effectively a whole new verb: "throw away" has a meaning entirely different from "throw". TADS supports this construct with two-word verbs. Note that when you define a two-word verb, the second word in the two-word verb must be separately defined as a preposition–the parser does not automatically create a preposition for the word. For example, a ver b defined as "pick up" requires that the preposition "up" be defined as well (these particular examples are defined in adv.t).

If the word following the verb is defined as a preposition, the parser checks to see if it's defined as part of a two-word verb with the verb in the sentence. If it is, the parser takes the pair of words as the verb–if the player types "pick up" as the fir st two words of a command, the parser takes the pair as the verb-preposition combination. If the preposition does *not*go with the verb, and the same word can be used as a noun or an adjective, the parser takes the verb to be the first word of the command only, and assumes the second word is being used as part of a noun phrase.

Once the verb is identified as the first one or two words of the sentence, the parser checks to see if anything remains. If the next word is a sentence separator ("and" or a comma), or if no more words follow, the parser takes it as the end of the sentence, and executes the verb without any objects.

If the next word starts a noun phrase, the reads the noun phrase that follows, and then checks to see what follows that. If the next word is a sentence separator, the parser takes it as the end of the sentence, and executes the verb with the noun phrase as the direct object.

If, instead, the next word is a preposition, the parser checks what follows the preposition. If the next word starts another noun phrase, the parser reads this second noun phrase, then executes the command with both a direct and an indirect object, with th e preposition separating them.

If the word following the preposition is a sentence separator, the parser takes the preposition as part of the verb. With verb-preposition combinations, the preposition is sometimes placed immediately after the verb, as in "pick up the book", but can also b e put at the end of the sentence, as in "pick it up". So, if the parser finds a preposition at the end of the sentence, it treats the preposition the same way it would have if the preposition had immediately followed the verb.

Note that many of these cases can be ambiguous–based on the parts of speech of the words in a sentence, more than one interpretation is possible. The parser's rules as described above are designed to choose the most sensible interpretation, but sometimes t he results will not be exactly what you may have intended. For example, if your game has an object defined as an "up button" (in an elevator, for example), the word "up" will end up defined as a verb, preposition, and adjective. If the player types "push u p button", and no verb is defined as "push up", the parser will know that "push" and "up" don't go together as a verb and will interpret this as applying the verb "push" to the object "up button". However, if the player types "pick up button", the parser w ill interpret this as applying the verb "pick up" to the object "button"; if you also have a down button, the parser will ask the player which button to take–which would be confusing if the player had intended to apply the verb "pick" to the object "up but ton".

Ambiguous word definitions are very difficult to anticipate, because there are so many possible combinations of words in even a small game. The best way to find these is to test your game, and have other people test it.

Note that in the course of identifying the verb, the parser has also identified the noun phrases that make up the direct and indirect objects, and the preposition that separates the direct and indirect objects. For the command "open the door and the window, " the parser identifies the following sentence components:

```
actor:  Me

verb:  open

direct object:  the door and the window

preposition:  none

indirect object:  none
```

For the command "joe, pick up the ball with the tongs", the parser will identify the following elements:

```
actor:  Joe

verb:  pick up

direct object:  the ball

preposition:  with

indirect object:  the tongs
```

2.2.7 Noun Phrases and Objects

A noun phrase is made up of an optional article, one or more optional adjectives, a noun or a plural, and optionally the word "of" (or an equivalent defined with specialWords) and another noun phrase. A word is a noun if it matches a word defined in the noun property of an object in your game; likewise for adjectives, plurals, and articles.

Certain special words can be used as noun phrases. The word "all" is by itself a valid noun phrase, as are the pronouns (it, him, her, and them). "All" or "all of" followed by a plural noun phrase (which is a noun phrase whose last word is a plural rather t han a noun) is a noun phrase, equivalent to the plural noun phrase without the "all" or "all of". Similarly, "both" and "both of" can be used in exactly the same way. "Any" followed by a noun phrase, or "any of" followed by a plural noun phrase, can also b e used; these tell the parser to arbitrarily pick one of the objects indicated.

The player can also specify a count with a plural or with "any". For example, phrases such as "3 books", "any 3 books", "3 of the books", and "any 3 of the books" work the same as "any book", but the parser will (arbitrarily) choose three of the named obje cts in this case, rather than just one. If the number is "1", the parser allows this format with a singular noun phrase as well: "1 book" or "any 1 book", which are equivalent to "any book".

The player can also use multiple noun phrases, by separating each with "and" or a comma.

During the first phase of parsing, before starting to execute the command, the parser identifies *all* of the possible objects for each noun phrase. After finding the words involved in the noun phrase (which is done entirely on the basis of their defined parts of speech, as described above), the parser makes a list of all of the objects which mat ch all of the words in the noun phrase. For example, if the noun phrase is "pile of red paper", the parser finds every object which has "pile" defined as a noun, "red" defined as an adjective, and "paper" defined as a noun. The parser intersects these lists of objects to arrive at a list of objects that have all three words defined.

In most circumstances, a number entered as part of a command serves as a noun, and is assigned to numObj (see the section on object resolution on page 94). In some cases, though, you may want to use a number as part of the name of an object. For example, you might have a five dollar bill, in which case you would want "5" to serve as an adjective for the object.

Similarly, if you have an elevator with a button for each floor, the buttons will be called “button 3”, and so on.

The parser allows you to enter numbers as adjectives. For example:

```
button3:  floorButton
        adjective = '3' 'three'
        floor = 3
;
```

When a number is defined as an adjective, and the object is present, the parser allows the number to be used as either a prefix or a suffix in the noun phrase: “button 3” and ”3 button” (as well as “three button”) are valid noun phrases referring to the bu tton.

2.2.8 Command Execution: Part One

Now that the command has been identified, the parser begins to execute it.

The first step in execution is to call a user function called preparseCmd. This functional is optional; if your game doesn't define one, the parser skips this step. The parser calls the function with a single parameter: a list of (single-quoted) strings, one string per word in the command. For example, for the command “open the door and the window”, the parser calls the function like this:

```
preparseCmd(['open' 'the' 'door' ',' 'the' 'window'])
```

Note that the word “and” became a comma. “And”, and certain other words, are transformed internally before this function is called:

```
and     ,
all     A
but     X
it      I
them    T
him     M
her     R
any     Y
```

Your preparseCmd function can return one of three values: true, nil, or a list. If the function returns true, processing continues as normal. If it returns nil, the command is abandoned–the parser discards the current command, and goes back to the beginning of the parsing process to get a new command. If the function returns a list, the list must consist entirely of single-quoted stri ngs; the parser will go back to the step described above under "Checking for an Actor", substituting the returned list for the original command.

Note that the argument to preparseCmd doesn't include the actor specification. For example, if the player types "joe, go west", preparseCmd will receive only "go west". In addition, the list doesn't include the words separating the command from the previous or next command on the same line. For example, if the player types "go west, then pick up the box", t he function is called first with simply "go west", then with "pick up the box".

In addition, this command is called for each command just prior to the steps described below. For example, if the player types "go west, then pick up the box", the parser first calls preparseCmd with "go west", and then proceeds through the rest of the processing for this command as described below; after that's all done, the parser comes back to this step, calls preparseCmd with "pick up the box", and then proceeds through the steps below once again.

The preparseCmd function is allowed to change a particular command only once. If preparseCmd returns a list value, the parser goes back and starts processing the list as a new command–as a result, the new command will itself be submitted to preparseCmd when the parser gets back to this step. On this second call, preparseCmd is not allowed to return a list; if it does, the parser displays an error that indicates the preparseCmd appears to be stuck in a loop (message 402) and cancels the command.

Here's a sample implementation of preparseCmd. This implementation converts commands in the format "tell *actor*to *command*" to the standard TADS notation for these commands, "*actor*, *command*".

```
preparseCmd: function(cmd)
{
        local i, tot, to_loc, actor, the_rest;

        /* see if there's a ''to'' */
        for (i := 1, tot := length(cmd) ;
                        i <= tot ; ++i)
```

```
        {
           if (cmd[i] = 'to')
           {
                /* note the location */
                to_loc := i;

                /* we're done looking */
                break;
           }
        }

        /* check for TELL actor TO command */

        if (tot >= 1 and cmd[1] = 'tell' and
               to_loc <> nil and to_loc > 2)
           {
           /* get the actor */
           for (i := 2, actor := []; i < to_loc ; ++i)
                actor += cmd[i];

           /* get the command */
           for (the\_rest := [],i := to\_loc+1 ; i <= tot ; ++i)
                the\_rest += cmd[i];

           /* change it to ``actor, command'' */
           return actor + ',' + the\_rest;
        }
        else
        {
           /* process the command unchanged */
           return true;
        }
}
```

After preparseCmd returns true, the parser makes sure it has a valid deepverb object for the verb phrase. If no object is found that defines the verb phrase as one of its verb property values, the parser displays an error (message 18, "I don't understand that sentence") and cancels the commands.

Next, the parser runs the roomCheck method in the object Me. The roomCheck method is called with a single argument: the deepverb object. For example, if the player types "take book", the parser calls this method:

```
Me.roomCheck(takeVerb)
```

This method should return either true or nil: a return value of true indicates that the command is allowed to proceed. If roomCheck returns nil, the command is canceled. The roomCheck method is expected to display a message indicating why the command is not allowed, so the parser doesn't display any additional information when roomCheck returns nil–it simply cancels the command without further comment.

The roomCheck method is intended to be a first, coarse check of whether a command is valid in a room. This method is called prior to any disambiguation, and once for the entire command (regardless of how many direct objects may be i nvolved in the command). The main use of this method is to disallow entire verbs in a room based on the room's condition; for example, you could use roomCheck to prevent the player from being able to take any objects while in the dark. Since this method is called prior to disambiguation, you can prevent the disambiguation process from displaying any information about the room th at the player should not be able to see.

The default roomCheck defined in the basicMe class in adv.t simply returns the value of the roomCheck method in the player's location. (The roomCheck defined in the movableActor class does the same thing for the actor's location.) The roomCheck defined in the room class in adv.t always returns true, and the roomCheck defined in the darkroom class in adv.t returns nil unless the location is lit, or the verb is a "dark verb" (i.e., its isDarkVerb property is true), which means that it can be used in the dark. All "system" verbs (such as "save" and "restore"), as well as a few other verbs that don't logically require light (such as "wait"), are defined in adv.t as dark verbs.

2.2.9 Verb Templates

The parser must now determine how to execute the command. The first step is to identify the deepverb object associated with the word or words making up the verb; to do this, the parser simply looks up the verb phrase in its dictionary, and finds the object that defined the phrase as one of its verb property values.

A single deepverb can have multiple interpretations; these interpretations are called "templates", because they serve as models for the possible sentences that can be made with the verb.

TADS verb templates are a subtle concept, because templates are not explicitly stated in a deepverb; rather, they are *implied* by the presence of the properties action, doAction, and ioAction. If an action method exists for the verb, the template sentence that consists only of the verb is implicitly defined. If a doAction property exists for the verb, the template sentence that consists of the verb and a direct object is implicitly defined. If an ioAction property is defined for a particular preposition object, the verb implicitly defines the template sentence that consists of the verb, a direct object, the specified preposition, and an indirect object.

The "wait" verb defined in adv.t defines only the no-object template:

```
waitVerb:  darkVerb
        verb = 'wait'  'z'
        action(actor) =
        {
           "Time passes...\n";
        }
;
```

The "lock" verb defined in adv.t defines both a one-object and two-object template, but doesn't have a no-object template:

```
lockVerb:  deepverb
        verb = 'lock'
        sdesc = ''lock''
        ioAction(withPrep) = 'LockWith'
        doAction = 'Lock'
        prepDefault = withPrep
;
```

Note that the ioAction definition can contain certain special flags that affect the way commands matching the template are processed. For example, you could define a template as follows:

```
ioAction(aboutPrep) = [disambigDobjFirst]'TellAbout'
```

The [disambigDobjFirst] flag (which is the only flag currently defined) specifies that the parser is to disambiguate the direct object first when a sentence involving this template is executed. Normally, the parser disambiguates the indirect object first, then disambiguates the direct object with the indirect object known. For some commands this is undesirable, because it means that the direct object isn't known when the indirect object is being disambiguated. This flag provides control over the sequence of disambiguation, so that you can determine which object is known first, which allows you to use that object while disambiguating the other.

To choose a template, the parser uses the components of the command, which were determined earlier (see the section "Identifying the Verb" on page 78). If the only non-empty component of the command is the verb, the parser chooses the no-object template; if the deepverb defines an action method, it has a no-object template, and therefore allows no-object sentences.

If the sentence has a verb and a direct object, but no preposition or indirect object, the parser chooses the one-object template. If the deepverb has a doAction property, the verb has a one-object template, and thus allows single-object sentences. The doAction property specifies the verification and action methods for the command: TADS appends the string defined by doAction to the string "verDo" to form the verification method, and "do" to form the action method. For example, if doAction = 'Take', the verification method is called verDoTake, and the action method is called doTake.

If the sentence has a verb, direct object, preposition, and direct object, the parser uses a two-object template. A single deepverb object can define multiple two-object templates; these multiple templates are distinguished from one another by their prepositions. The templates are defined by the ioAction property–each ioAction specifies the preposition which identifies it. If the player types "dig in dirt with shovel", the parser looks in digVerb for an ioAction(withPrep) definition. (Note that digVerb. and withPrep could actually be called anything–we're following the naming conventions used in adv.t here, but you can call verb and preposition objects anything you want. The parser doesn't identif y these objects based on their names, but rather based on their vocabulary words, defined by their verb and preposition properties, respectively.)

The parser must now match the components of the sentence entered by the player with the templates available for the verb. Before going on, the parser checks to make sure that the verb object is properly defined: it must define at least one sentence templat e. If the deepverb object fails to define any templates (that is, it has no action, doAction, or ioAction properties), the parser displays an error (message 23) and cancels the command.

Case 1: A Command with No Objects

If the player's command has no objects at all, but consists only of a verb, the parser checks to see if the deepverb has an action method; if so, the parser chooses the verb-only sentence template and proceeds with the execution.

If the command has no objects, and the deepverb object does *not* define an action method, the parser tries to provide a default direct object by calling the doDefault (which stands for "direct-object default") method in the deepverb object. This method is called with the actor, the preposition, and nil (since the indirect object is not yet known) as arguments. For example, if the player types simply "pick up", the parser chooses takeVerb as the deepverb object, sees that it has no action property, so attempts to obtain a default object:

```
takeVerb.doDefault(Me, nil, nil)
```

This method is meant to return a list of objects that should be used by default for the command; the definition of this method for the takeVerb defined in adv.t returns a list of all movable objects in the player's location. If this method fails to return a list, the parser simply asks the player to supply a direct object (see below). Otherwise, the parser now gets the value of the prepDefault property in the deepverb object: if this value is not nil, it must refer to a preposition object, which becomes the preposition for the command.

For example, if the player simply types "dig", the parser gets the default list from the doDefault method defined in digVerb, then gets the value of digVerb.prepDefault. Suppose that prepDefault has a value of withPrep: the command's components now look like this:

```
actor:  Me

verb:  dig

direct object:  (the list returned by doDefault)

preposition:  with
```

```
indirect object:  nil (because it's not yet known)
```

The parser checks that the template specified by the new components exists for the verb. If it doesn't, the parser skips checking the doDefault return value, and goes on to ask for a direct object.

If a valid list resulted from doDefault, and the new template is valid, the parser goes through the doDefault list to check each item with the direct object verification method defined by the doAction or ioAction method for the new template. If the template is a single-object sentence, which means that the doAction definition is used, the parser calls the verification method with the actor as a parameter. If the template is for a two-object sentence, which means that the ioAction definition is used, the parser calls the verification method with the actor and nil (because the indirect object is not yet known) as parameters. If the ioAction is used, but the ioAction template specifies the [disambigDobjFirst] flag, the second (nil) parameter is omitted. Some examples:

```
object.verDoTake(Me)

object.verDoLockWith(Me, nil)

object.verDoTellAbout(Me)
```

In this third example, we're assuming that tellVerb's ioAction property has the [disambigDobjFirst] flag.

The verification methods are called silently, which means that any messages they display are hidden from the user (the same as during the disambiguation process). However, the parser notes whether these methods attempt to display any messages; any verifica tion method that attempts to display a message is considered to fail. The parser discards any objects from the list whose verification method fails.

If exactly one object from the doDefault list passes the silent verification test, that object is chosen as the default direct object for the command. The parser displays the object to the player: if your game program defines the function parseDefault, the parser calls this function with the chosen default object as the parameter, and expects that the function will display a message to indicate that the object has been chosen as the default. Otherwise, the parser displays message 130 ("("), then calls the method thedesc in the default object, then displays message 131 (")"). It then starts this step over again with the new set of command components.

Here's a sample implementation of parseDefault. This definition generates the same message that the parser would if parseDefault weren't defined.

```
parseDefault: function(obj, prp)
{
        "(";
        if (prp) "<<prp.sdesc>> ";
        obj.thedesc;
        ")";
}
```

If no objects survive the verification test, the parser doesn't have any way to supply a default object. If more than one object survives, the parser still does not supply a default object, because the command doesn't imply a specific object. In either ca se, the parser now asks the player to supply a direct object; see the description below.

Case 2: A Command with One Object

If the player's command has a direct object, but no preposition or indirect object, the parser checks the verb for a single-object sentence template. If the deepverb object has a doAction property, the single-object sentence is allowed. The parser first disambiguates the direct object list (see the section on resolving objects on page 94). If this succeeds, the parser saves the direct object (or list of direct objects) for future use as "it" or "them" (or "him" or "her", if appropriate), and proceeds to execute t he command.

If a single-object sentence is not allowed, the parser checks the prepDefault property of the deepverb object; if it's nil, the sentence is not allowed, so the parser displays message 24 ("I don't recognize that sentence") and terminates the current command. Otherwise, the parser uses the prepDefault value as the preposition for the command, and calls the ioDefault method of the deepverb with the actor and preposition as arguments. For example, if the player types "put ball", and the verb has a default preposition of "in", the parser calls ioDefault like this:

```
putVerb.ioDefault(Me, inPrep)
```

If this method returns something other than a list, the parser ignores it, and asks the player to supply an indirect object (see below). Otherwise, the parser checks to see if the direct object must be disambiguated prior to the indirect object–if the ioAction property defines the [disambigDobjFirst] flag, the direct object must be disambiguated first. If so, the parser disambiguates the direct object using the normal process (see the section on disamb iguation on page 94). Next, the parser goes through the ioDefault list, and silently calls the verification method for each object. For normal verbs, these calls look like this:

```
object.verIoPutIn(Me)
```

When the ioAction property has the [disambigDobjFirst] flag, these calls look like this:

```
object.verIoPutIn(Me, directObject)
```

These calls are made silently–any messages they display are hidden from the player. However, the parser notes whether they attempt to display anything; any verification method that attempts to display a message is considered to have failed. The parser re moves from the ioDefault any objects whose verification method fails at this point.

If exactly one object survives this test, the parser uses this object as the default indirect object and proceeds with the command. First, though, it shows the player that the object is being assumed. If your game defines a function called parseDefault, the parser calls this function with the verb and the preposition as arguments. Note that you must define parseDefault to take a variable argument list, because it can be called with either one or two arguments: when called with one argument, it means that the a default direct object is being assumed; when called with two arguments, it means that a default indirect object is being assumed. Note further that the preposition supplied as the second argument can be nil, which will be the case when a two-object command is entered that has no preposition (for example: "give joe the bottle"). If your game doesn't define a parseDefault function, the parser generates a message by displaying message number 130 ("("), then calling the preposition object's sdesc method, then displaying message 132 (a space), then calling the default indirect object's thedesc method, and then displaying message 131 (")").

If no objects survive the verification test, the parser can't supply a default indirect object. If more than one object survives, the parser doesn't supply a default indirect object, because the command doesn't imply a particular object. In either case, t he parser asks the player to supply an indirect object (see below).

Case 3: A Command with Two Objects

If the player specifies a command with two objects and no preposition, the parser treats the first object as the indirect object, and the second as the direct object, and uses the two-object sentence format. When this type of sentence is entered, the parse r gets the value of the deepverb object's nilPrep property–this property should return a preposition object that is to be used for a sentence of this type. If nilPrep doesn't return a preposition object, the parser assumes that the preposition is "to", and finds the preposition object with that vocabulary word. In any case, the parser transforms the sentence from the format VERB IOBJ DOB J to the standard two-object sentence format, VERB DOBJ PREP IOBJ, then proceeds with the command as though it had been entered that way in the first place.

If the player enters a command in the format VERB PREP OBJ1 OBJ2, the parser converts this to VERB OBJ2 PREP OBJ1–in other words, the original sentence is interpreted as VERB PREP IOBJ DOBJ. After the conversion, the parser proceeds with the sentence as t hough it had originally been entered as VERB DOBJ PREP IOBJ. This conversion is most useful for some non-English languages; English speakers don't typically use this construction.

If the player specifies a command with a preposition and one or two objects, the parser finds the ioAction definition that matches the given preposition. If the verb doesn't have an appropriate ioAction property, the parser displays message 24 ("I don't understand that sentence"), and abandons the command.

If only one object was provided, the parser takes the object that was given as the indirect object, and attempts to find a direct object. The parser uses the same mechanism to find the direct object–first by trying to find a default direct object, then by asking the player–as described above in Case 1. This type of sentence is useful when the *direct*object is implied by the verb; for example, if a single actor is present in the room with the player, the direct object of "ask about the murder" would implicitly be the actor.

If two objects are provided, the parser disambiguates the two objects. If the ioAction has the [disambigDobjFirst] flag, the parser disambiguates the direct object first; otherwise, it disambiguates the indirect object first. It then disambiguates the other object. See the section on resolving objects (page 94) for details.

The parser never allows a command to use multiple indirect objects. If the command contains more than one indirect object, the parser displays message 25 ("You can't use multiple indirect objects") and abandons the command. When the direct object is disam biguated first (which will happen only when the ioAction has the [disambigDobjFirst] flag), the parser also restricts the direct object to a single object; if multiple direct objects are specified in this case, the parser displays message 28 ("You can't use multiple objects with this command") and abandons the command.

Once the objects have been disambiguated, the parser saves the direct object or objects as "it" or "them" (or "him" or "her", as appropriate), then executes the command.

Asking for an Object

If the verb requires a direct or indirect object, but the command doesn't specify one and the parser can't find a suitable default, the parser must ask the player to supply an object for the command. First, it must display a message asking for the object.

To do this, the parser checks to see if your game program defines a function called parseAskobjActor; if so, it calls this function with the actor as the first argument, the deepverb object as the second argument, and the preposition as an optional third argument–this third argument is present only if an indirect object is required, and even then may be nil. Your parseAskobjActor function is expected to display a message asking the player to supply a direct object for the verb. (Note that if you don't define a parseAskobjActor function, but you *do*define a function called parseAskobj, the function parseAskobj will be called instead. parseAskobj has the same purpose as parseAskobjActor, but does not receive the actor as a parameter. parseAskobj is called for compatibility with games written prior to version 2.2; you should use parseAskobjActor for new games, since this function gives you more information.)

Note that the parseAskobjActor function is called with a third argument only when the parser is asking for an indirect object. The third argument is the preposition object, which may be nil. Because the parser calls parseAskobjActor with two *or* three arguments, depending on whether it's asking for a direct or indirect object, your parseAskobjActor function must be defined as taking variable arguments. You can determine if the function is being called with two or three arguments by inspecting the argcount pseudo-variable.

If your game does not have a parseAskobjActor (or parseAskobj) function, the parser generates its own message. If no actor (other than Me) is specified in the command, the parser displays message 140 ("What do you want to"); if an actor is specified, the parser displays message 148 ("What do you want"), then calls the actor's thedesc method, then displays message 149 ("to"). Next, the parser calls the sdesc method in the deepverb object. If an indirect object is required, the parser next displays an appropriate pronoun for the direct object, then calls the preposition's sdesc method (if the preposition is nil, the parser displays message 142, "to"). Finally, the parser displays message 143 ("?").

Here's a sample definition of parseAskobjActor, which generates the same prompt that the parser would if parseAskobjActor weren't defined.

```
parseAskobjActor: function(a, v, ...)
{
        if (argcount = 3)
        {
                "What do you want ";
                if (a <> Me) a.thedesc;
                " to <<v.sdesc>> <<getarg(3).sdesc>> it?";
        }
        else
        {
                "What do you want ";
                if (a <> Me) a.thedesc;
                "to <<v.sdesc>>?";
        }
}
```

If a pronoun for the direct object is required, the parser checks each object in the direct object list. If the player explicitly specified multiple direct objects (by using "all" or a list of noun phrases), the parser displays message

144 ("them"). Other wise, if every object in the list returns consistent values from isHim and isHer, the parser displays message 147 ("them") if all objects return true for *both*isHim and isHer; message 145 ("him") if only isHim returned true; message 146 ("her") if only isHer returned true; or message 141 ("it") if neither isHim nor isHer were true for all objects.

After displaying a message (either with parseAskobjActor or through a set of messages as described above), the parser allows the user to enter a new command, using commandPrompt code 3 if asking for a direct object and code 4 for an indirect object. If the new command appears to consist only of a noun phrase, the parser uses that noun phrase for the missing object, and goes back to the beginn ing of this step to process the command with the new noun phrase filling in the missing component. Note that the player can use anything in this noun phrase that would have been allowed in the command itself, including multiple objects and special words su ch as "all".

If the new command does not appear to consist of only a noun phrase, the parser abandons the current command and starts over from the beginning with the new command. So, when the parser asks the player to supply an object, the player can choose either to a nswer the question, or to type a brand new command.

2.2.10 Resolving the Objects

After the parser has made a final determination of the form of the sentence, and identified the verb template, it must resolve the noun phrases in the command to game objects. Up to this point, the parser has been keeping a list of the objects matching eac h noun phrase; these lists include all objects in the *entire game*that match each noun phrase. At this point, the parser must determine which specific object or objects the player meant for each noun phrase. This process is called "resolving" o r "disambiguating" the object references.

Certain special noun phrases can be resolved with very little work. The player can include a string enclosed in double quotes in a command:

```
>type "hello, world!" on the terminal
```

Strings are always mapped to the special object strObj, which must be supplied by your game program (adv.t provides a class called basicStrObj, which provides a suitable definition that you can use a superclass of your own strObj object). In this case, the parser uses strObj as the direct object to the command, and sets the value property of strObj to the (single-quoted) string value 'hello, world!'.

Likewise, the player can use numbers:

```
>dial 8056 on the phone
```

Numbers are mapped to the special object numObj; as with strObj, this must be supplied by your game (adv.t provides a basicNumObj class). In this example, the direct object will be numObj, and its value property will be set to the numeric value 8056.

The special words for the pronouns (it, them, him, and her) are taken from the direct object of the previous command. (Note that you can also set the meaning of the pronouns from within game code, using the setit() built-in function. When you don't use setit() to set the pronouns yourself, the system will automatically set the pronouns to the direct object of the previous command.) Before accepting the pronoun, the system uses the access-checking function (va lidDo for a direct object, validIo for an indirect object) to ensure that the object is still accessible.

The special word for "all" is replaced with the object list returned by the doDefault property of the verb. The parser calls doDefault, and includes all of the objects from the resulting list in the direct object list. If a list of objects is excluded with "but" (for example: "take everything except the key and the idol"), the parser removes the objects in the exception list from the doDefault list.

Ordinary noun phrases require more analysis. In particular, we must identify the actual objects to which the noun phrases refer.

First, we identify which objects are accessible. To do this, the parser calls validDoList (or validIoList, as appropriate) in the deepverb. This method is called with the actor, preposition, and other object as parameters; the preposition will be nil for one-object commands, and the other object may be nil if it's not known yet or isn't appropriate (when calling validDoList for a one-object command, the other object will always be nil). This method returns either a list of objects, or nil; if it returns a list, this list is intersected with the list of objects that the parser found for the noun phrase, otherwise the entire noun phrase object list is retained. Note that validDoList and validIoList must return every valid object, but they need not return *only* valid objects–since the remaining objects will be checked for validity with validDo or validIo, validDoList and validIoList can harmlessly include invalid objects in their results.

Next, we call validDo (or validIo) in the deepverb for each surviving object. The arguments to this method are the actor, the object whose validity is being tested, and a "sequence number". The sequence number starts off at 1, and is incremented for each object in the lis t. The sequence number can be used when you wish to arbitrarily select a single object from a list of ambiguous objects–simply return true only for the first object, and nil for all of the other objects. If this method returns true, it means that the object is valid for the current command–in other words, the object is accessible to the actor for the purposes of the command. If the player wishes to take an object, for example, the object should be valid for the command if and only if the player can reach the object. For other commands, accessibility may not necessitate that the object is reachable; for example, to look at an object, it need only be visible–if the object is inside a locked glass case, it will be visible but not reachable. The parser eliminates from consideration any objects for which validDo (or validIo) returns nil.

We now check each surviving object to see if it's logical with the verb. To do this, we make a "silent" call to the verification method. Recall that the verb template we chose earlier specifies a pair certain methods–a verification method, and an action method–that we will call in the direct and indirect objects. For example, the verb "take" may define a doAction = 'Take', which means that the verification method for the direct object is verDoTake, and the action method is doTake. The call is silent because we hide any messages that the method displays–the player will not be aware that we are making this call. However, we *do* take note of whether the method attempts to display any messages; if it does, the verification fails for the object. We make a list of all of the objects that survived the validity test *and* the verification test.

We now have two lists: a list of *valid* objects (objects which passed the validity test), and a list of *logical* objects (objects which passed the verification test).

If both lists are empty, there's no way to apply the verb to the player's noun phrase–we can't identify any objects for which the command is valid. We check to see if the player's noun phrase referred to any *visible*objects; if not, we simply display message 9 ("I don't see any %s here", where the "%s" is replaced by the words from the player's noun phrase) and abandon the command. Otherwise, we have a situation where the player referred to one or m ore visible objects, but those objects can't be accessed for the purposes of this command. In this case, we check to see if the verb object has a cantReach method–if so, we call the verb's cantReach method with four parameters: the actor, a list of inaccessible direct objects, a list of inaccessible indirect objects, and the preposition. Actually, only one of these lists will ever be supplied, and the other will alw ays be nil; when the direct object is inaccessible, the first argument has the list of inaccessible (but visible) direct objects, and the second argument is nil. When the indirect object is inaccessible, the arguments are reversed. The preposition argument will be nil if the command is a one-object sentence.

For example, if a closed glass case contains a red book and a blue book, and the player types "take book", the word "book" will refer to the two books inside the case; neither of these books will be accessible for the "take" command, because they're inside a closed container, but they're both visible. The parser will call the cantReach method defined in the verb:

```
takeVerb.cantReach(Me, [redBook blueBook], nil, nil)
```

If the verb object doesn't define or inherit a cantReach method, the parser uses a different mechanism: for each visible object, it calls the object's sdesc method, then displays message 200 (":"), then calls the cantReach method in the object itself–*not*in the verb–with the actor as the single argument.

If the list of logical objects is empty, but the list of valid objects is not empty, the parser considers only the list of valid objects. Otherwise, it considers only the list of logical objects.

At this point, the parser needs to determine whether the objects in the list are uniquely specified–in other words, it determines whether it has identified a single object for each singular noun phrase that the player entered. If the player types "take bo ok", the parser must find a *single object*to which "book" refers. Likewise, if the player types "take book and box", the parser must identify a single object for "book", and a single object for "box".

If any of the noun phrases were plural, they're now finished–the player explicitly asked us to use every object matching the phrase by using a plural. For example, if the player types "take books", we will apply the command to every object which matches t he vocabulary word "books" in its plural property.

Similarly, if a noun phrase was specified with "any", as in "take any book" or "drop any of the coins", the player has explicitly asked us to choose one of the objects matching the phrase. The parser simply picks one of the remaining objects (it picks the first object in its internal list, but the order of objects in the internal list is unpredictable).

In addition, if the objects themselves are all indistinguishable from one another, the parser also picks one of the objects arbitrarily. Objects are indistinguishable if two conditions are true: first, that the objects are all derived from the exactly the same class–that is, every object's immediate superclass is the same; and second, that the objects all have the property isEquivalent set to true. If these two conditions hold, the parser considers the objects to be indistinguishable, and simply picks one from the list, exactly as though the user had used "any" with the noun phrase.

For a singular noun phrase, we must now have a single object. If so, we're done with the object. If more than one object remains, however, we have no way of knowing which of the remaining objects the player meant, so we must ask the player to tell us. To do this, we check to see if the game program provides a function called parseDisambig. If so, we call this function with two arguments: a (single-quoted) string giving the text of the noun phrase that the player typed; and a list composed of the surviving objects. If the game doesn't define a parseDisambig function, the parser generates a message: it displays message 101 ("Which %s do you mean", with the "%s" replaced by the text entered by the player for the ambiguous noun phrase), then calls thedesc for each surviving object, then displays message 104 ("?"). Between adjacent objects, the parse displays message 102 ("",); and it also displays message 103 ("or") after the second-to-last object. Note one unusual case: i f the ambiguous list contains a set of objects that are mutually indistinguishable, as described earlier, the parser displays only one description for the whole set of such objects, and uses adesc instead of thedesc for the object. For example, if the player types "take coin", and the room contains a gold coin and three silver coins which are all indistinguishable from one another, the message looks like this: "Which coin do you mean, the gold coin, or a silver coin?"

Here's an example implementation of parseDisambig, which generates the same message that the parser would generate if parseDisambig weren't defined.

```
parseDisambig: function(str, lst)
{
        local i, tot, cnt;
        "Which <<str>> do you mean, ";
```

```
        for (i := 1, cnt := length(lst) ;
                 i <= cnt ; ++i)
        {
                lst[i].thedesc;
                if (i < cnt) ", ";
                if (i+1 = cnt) "or ";
        }
        "?";
}
```

After displaying prompting message, the parser lets the user enter a command using commandPrompt code 2. If the player types something that looks like a noun phrase, the parser tries to use the new information to choose from the surviving objects. If the player simply types "all" or "both", the parser chooses al l of the surviving object. If the player types "any", the parser chooses one of the surviving objects arbitrarily. In either case, this completes disambiguation, since the parser has identified the objects the user wishes to use in the command.

If the player types adjectives and/or nouns, the parser limits the surviving list to objects that are associated with all of the new vocabulary words. If no objects remain, the parser displays message 16 ("I don't see that here"). If exactly one object re mains, disambiguation is complete, because the noun phrase has been resolved to a particular object. Similarly, if the player typed a noun phrase that consists of multiple objects (for example: "the red one and the blue one"), and the remaining list has t he same number of objects as the number of noun phrases the player entered, disambiguation is completed. Otherwise, the phrase is *still* ambiguous, so the parser goes back and asks the user once again which of the remaining objects is to be used. Note that if parseDisambig isn't defined, the parser displays message 100 ("Let's try it again:") before asking again.

If the player types something that doesn't look like a noun phrase, the parser abandons the current command, and starts over at the beginning, using the new input as the next command.

2.2.11 Command Execution: Part Two

At this point, everything about the command is known. The parser has converted the player's words into a set of objects, with each object having a specific function in the sentence. Each component of the command–the actor, the verb, the direct object, the preposition, and the indirect object–has been resolved to an object or a list of objects. We have also identified the verb template, which specifies the methods that are to be called in the objects at certain points during execution. We now have all the information necessary to execute the command.

2.2.12 Processing Multiple Objects

A command can refer to multiple direct objects; for example, the player can type "take the ball and the box". Fortunately, the details of handling multiple objects are entirely handled by the parser; you write the verb handlers in your game for single obje cts only, and the parser calls these handlers as many times as necessary.

When the player uses multiple direct objects, either by listing several objects or by using "all", the parser checks to make sure the verb allows multiple direct objects. (In certain cases, multiple objects will be disallowed before this point. The parser never allows multiple indirect objects, and it doesn't allow multiple direct objects for verb templates that specify the [disambigDobjFirst] flag.) To determine if the verb allows multiple direct objects, the parser calls the rejectMultiDobj method in the deepverb object, passing the preposition as the argument (the argument is nil for single-object commands). If this method returns true, the parser terminates the command without further processing. Note that the parser doesn't display anything if rejectMultiDobj returns true–the parser expects this method to display an appropriate error message if it returns true. You can use rejectMultiDobj if you want to discourage the player from guessing for certain verbs.

The remainder of the processing for the command is done for each object in the direct object list. The parser goes all the way through the rest of the command processing for the first direct object, then comes back to this step and goes through the same pr ocess for the next direct object, and so on.

Before proceeding, the parser displays the name of the current direct object if multiple direct objects are being used. If only a single object is used in the command, the parser doesn't display the object name–this is only done for multiple objects, so t hat the player can see the results of the command for each individual object. Note that the parser displays the object name when "all" is used even if "all" ended up referring to only a single object, because the player didn't say exactly which object he o r she meant. For the same reason, we'll display the object name when the player used "any".

To display the object name, the parser first calls the multisdesc method in the current object (the parser calls sdesc instead if the object doesn't define or inherit multisdesc–this is for compatibility with games written with previous versions of TADS, since multisdesc is a recent addition); then, it displays message 120 (":").

2.2.13 "Again"

Before going on, we check to see if the verb is "again", which requires special handling. This verb is used to repeat the last command. To identify this verb, TADS requires that you provide an object named againVerb–this allows you to define the vocabulary words that are to be used for this special internal command. When the deepverb object for the command is againVerb, the parser knows to perform the special handling necessary to repeat the last command.

If the player types "again" as the first command of the game, the parser displays message 26 ("There's no command to repeat"), and abandons the command. Otherwise, the parser gets the information from the last command–the actor, verb, direct object, prepo sition, and indirect object–and uses this as the new command information. Before proceeding, the parser calls the validDo method in the deepverb to ensure that the direct object is still valid for the command, and calls validIo to ensure that the indirect object is still valid. If either of these methods indicates that the objects are not valid, the parser displays message 27 ("You can't repeat that command") and terminates the command.

When the verb is something other than "again", the parser saves all of the information for the current command. If the player uses "again" for the next command, it is this information that is used to repeat the command. Note that this information is saved once for each object in a multi-object command, so if the player types "again" after typing "take the box and the ball", the command repeated is "take the ball".

2.2.14 Checking with the Actor

The next step is to check to see if the actor will allow the command. Every command is directed to an actor–if the player doesn't explicitly direct a command to an actor, it implicitly goes to the player actor, Me. The parser calls the method actorAction in the actor object with four arguments: the deepverb object, the current direct object, the preposition, and the indirect object. For example, for the command "take the red book", the call would look like this:

```
Me.actorAction(takeVerb, redBook, nil, nil)
```

The actorAction method can either accept the command or disallow it. To accept the command, the method need not do anything at all. To reject the command, the method should use the exit or abort statement. If you use the exit statement, you will disallow the command, but the turn will still be counted–that is, daemons and fuses will run as though the turn succeeded. If you use abort, the turn doesn't count–the daemons and fuses will be skipped. In most cases, you should use exit if you want to disallow the command in actorAction, because the attempt to command the actor should still count as a turn. The abort statement is intended more for "system" commands, such as "save", that are intended to happen outside of the game's world, and so shouldn't count as turns within the game.

Note that the parser doesn't display any messages when actorAction uses exit or abort to disallow the command, so your actorAction method must display an appropriate error message before exiting or aborting. Note also that this method has no return value.

2.2.15 Checking with the Room

If the actor allows the command, the next step is to determine if the actor's current location allows the command. To do this, the parser get the value of the location property from the actor (if this isn't an object, it skips this step), then calls the roomAction method in that object. roomAction takes five arguments: the actor, the verb, the direct object, the preposition, and the indirect object. For example, if the player types "joe, put the ball in the box", the parser calls this:

```
joe.location.roomAction(joe, putVerb, redBall, inPrep, smallBox)
```

As with actorAction, the roomAction method can disallow the command by using the exit statement. This method must display a message if it disallows the command, because the parser will simply abandon the command if roomAction executes an exit statement. This method has no return value.

2.2.16 General Verb Processing

The next step is to call a general-purpose method in the indirect object, and then in the direct object. These general-purpose routines allow you to define certain behavior for an object for any verb from within a single method. You will sometimes want an object to act a certain way for all (or nearly all) verbs; for example, if you want to create an object that's off in the distance, you will want to handle every verb except "inspect" the same way: "That's too far away". You can do this through the gener al-purpose methods.

First, if the command has an indirect object, the parser calls the method iobjGen (indirect object general processing) in the indirect object. This method takes four arguments: the actor, the verb, the direct object, and the preposition.

Next, if the command has a direct object, the parser calls dobjGen in the direct object. This method also takes four arguments: the actor, the verb, the indirect object, and the preposition.

These methods, like roomAction and actorAction, can cancel the command by using the exit statement. These methods should display an appropriate message if they do cancel the command, since the parser will simply abandon the command.

Note that the general-purpose handlers are *not*called if the specific verb "overrides" them. If the object defines–not inherits, but actually defines–either a verification method or an action method for the verb, the general-purpose method is *not*executed. These methods are intended to be "catch-all" methods; if a verification or action method for a particular verb is defined for the object, the parser assumes that the verb is *not*to be handled by the catch-all method, so it skips calling the general-purpose routine.

Note that a verification or action method will "override" the general-purpose handler if it's defined in the same object as the general-purpose handler, or if it's defined in any subclass of the object that defines the general-purpose handler. Consider this example:

```
distantItem: fixeditem
```

```
        dobjGen(a, v, iobj, p) =
        {
                "It's too far away!";
                exit;
        }

        verDoInspect(actor) =
        {
               inherited.verDoInspect(actor);
        }
;

targetDistantItem: distantItem
        verIoThrowAt(actor) =
        {
               inherited.verIoThrowAt(actor);
        }
;
```

Now, any subclass of targetDistantItem will have verIoThrowAt override the presence of dobjGen. Since verIoThrowAt overrides dobjGen in targetDistantItem, the same applies to any subclass. The same is true of verDoInspect, because it overrides dobjGen by virtue of being defined in distantItem itself.

2.2.17 Verification and Action

If the command survives all of the steps so far, the parser carries out the command on the objects involved.

The first step is to verify that the command is possible. We have already determined that the objects involved in the command are accessible for the purposes of the command, but we don't know that the command can actually be applied to the object. For exa mple, if a door is already open, trying to "open the door" should fail.

Verification is done by calling the verification method, which is specified by the verb template as discussed earlier. If there are no objects in the command, there is no verification step; we move on immediately to the action step. Otherwise, we verify f irst the direct object, then, if the command has one, the indirect object.

Case 1: No Objects

If the command has no objects, we simply call the action method defined in the deepverb object. This method takes one argument: the actor. This method is responsible for carrying out the command; it should display an appropriate message–on either success or failure, since this is the entire command process or.

The action method has no return value. It can use the exit statement to skip any remaining commands on the same line, and it can use abort to terminate the turn and skip running fuses and daemons. Generally, you should only use abort with "system" commands that shouldn't count as turns. System commands, such as "save" or "restart", happen outside of the game world, so they shouldn't consume any time within the game world. For an ordinary command, even a failed command should usually count as a turn.

Note that the action method will generally use neither exit nor abort, even if the command fails. You should only use these statements when you want to skip the remaining commands on the same line, which should only be done if something really important happens in the command–ordinary failures in an action routine don't usually warrant canceling the remainder of the command.

Another way of exiting an action routine is to use the askdo statement. Executing an askdo statement lets you ask the player for a direct object, using exactly the same mechanism that the parser uses when it determines that a direct object is required– askdo even supplies a default direct object using the same techniques as the parser. See the description of default objects (page 70) details.

For example, when the player types "wait", the parser makes this call:

```
waitVerb.action(Me)
```

Case 2: Direct Object Only

If the command has one object, the parser starts by calling the verification method in the object. The verification method, which is named verDo*Verb* (where *Verb* is replaced by the doAction property value–for example, for "take", the verification method is called verDoTake), takes one parameter: the actor.

If the verification method is not defined or inherited for the object, the parser disallows the command and displays an error. To display the error, the parser checks to see if the game defines a function called parseError2. If so, it calls this function with four parameters: the verb, the direct object, the preposition, and the indirect object (note that the preposition and indirect object may be nil); this routine is responsible for displaying an appropriate message. If this function isn't defined, the parser generates a message: it displays message 110 ("I don't know how to"), then calls the sdesc method in the deepverb object. Then, if there's only a direct object, it displays message 111 (a space), then calls the thedesc method in the direct object; if there's an indirect object, the parser displays message 112 ("anything"), then calls sdesc in the preposition object (or displays message 113 ("to") if there's no preposition), then displays message 114 (a space), then calls thedesc in the indirect object.

Here's an example implementation of parseError2, which generates the same message that the parser would use if parseError2 were not defined.

```
parseError2:  function(v, d, p, i)
{
        "I don't know to how <<v.sdesc>> ";
        if (d)
                "<<d.thedesc>>".;
        else
                "anything <<p ? p.sdesc : ''to''
                >> <<i.thedesc>>".;
}
```

Apart from this extra message, the absence of a defined or inherited verification method is equivalent to the verification method failing by displaying a message.

If the verification method displays a message, the command is terminated; the parser proceeds with the next direct object in the same command, or moves on to the next command on the command line. The verification method doesn't have a return value–it indi cates failure by displaying a message, and indicates success by displaying nothing.

Note that verification methods must *not*make any changes to game state–they must never set the object of a property, or call any built-in functions with side effects (for example, never call restart or askfile from within a verification method). Verification methods can be called "silently", which means that the parser hides any messages they generate from the player–see the section on resolvi ng objects (page 94) for details. The parser is simply testing various possibilities when making these silent calls, and hasn't committed to a course of action yet, so these calls m ust not have any effect on the game state.

If the verification test succeeds–in other words, the verification method doesn't display any messages–the parser calls the action method in the direct object. The action method is named do*Verb* (for example, the action method for "take" is called doTake). Note that the do in doTake stands for "direct object"–it's not the word "do". The action method is called with one parameter: the actor.

The action method is responsible for carrying out the command. It should display an appropriate message–both on success and on failure, because this is the entire processing of the command.

Note that it's possible for the action method to reject the command, even though the verification method allowed it. If the action method wants to apply a tougher standard to the conditions under which the command can succeed, it can do so without any prob lem.

For example, when the player types "take red book", the parser calls these methods:

```
redBook.verDoTake(Me)

redBook.doTake(Me)
```

Case 3: Direct and Indirect Objects

When both a direct and an indirect object are specified, the verification sequence involves *both*objects. First, the parser checks to make sure that the verification methods are defined or inherited for both objects; if not, the command fails, and the parser displays a message as described above for Case 2.

The parser first verifies the direct object. It does this by calling verDo*Verb* in the direct object. For a normal verb, whose the verb template does *not* define the [disambigDobjFirst] flag, this method takes two parameters: the actor, and the indirect object. Otherwise, the only parameter is the actor.

If the direct object verification is successful, we verify the indirect object by calling verIo*Verb* in the indirect object. For a normal verb, whose template does not specify [disambigDobjFirst], this method takes only one parameter: the actor. For verb templates that *do* specify the [disambigDobjFirst] flag, this method takes two parameters: the actor, and the direct object.

If either verification method displays a message, the command fails, and we proceed to the next direct object of the current command, or to the next command on the command line if there are no more direct objects for the current command.

If both verification methods succeed, we call the action method for the verb template. The action method for a two-object command is *always* called in the *indirect* object. The method is called io*Verb*, and takes two parameters: the actor, and the direct object. (Note that the direct object is always included in this call, regardless of any special flags in the verb template.) As with a single-object action metho d, this action method is responsible for carrying out the command, and must display an appropriate message, whatever the outcome of the command.

Note that the parser will *not*call any action method in the direct object for a two-object command. Your action method in the indirect object is free to call a method in the direct object if you wish to have a direct object method process the co mmand, but the parser will never call such a method directly.

For example, if the player types "put ball in box", the parser makes these calls:

```
ball.verDoPutIn(Me, box)

box.verIoPutIn(Me)

box.ioPutIn(Me, ball)
```

2.2.18 Daemons and Fuses

After processing the verification and action methods, the command is finished. The parser goes back and processes the next direct object in the same command, as described earlier. The steps above are repeated for each direct object in the list.

Once all of the direct objects have been processed for a particular command, the turn is over. Even if the command line has additional commands following the current command, the parser considers the current command to be a complete turn–the additional co mmands will be processed, but will be counted as additional turns. So, once all of the direct objects have been processed for a command, the parser executes the daemons and fuses.

The parser first executes the daemons. The order in which the daemons are executed is arbitrary; it depends on the (unpredictable) order in which the system builds its internal lists as daemons are added and removed. The parser executes all active daemons set with the setdaemon() built-in function, then executes all active daemons set with the notify() built-in function.

Next, the parser executes any active fuses that has "burned down". As with daemons, the order in which fuses are executed is arbitrary and unpredictable. The system first executes any fuses set with the setfuse() function, then executes any fuses set with the notify() function. Before executing each fuse, the parser removes it from the list of active fuses. Only fuses that have burned down are executed; others are left to be processed when they're ready.

Note that the fuse timer is entirely under the control of your game program. The only time that fuse timers are advanced is when you call the incturn() built-in function. This function advances all fuse timers by one turn (or by a specified number of turns, if you provide an argument), and marks any fuses that burn down as being ready for execution. The incturn() function doesn't actually execute any fuses, but merely advances their timers.

2.2.19 Verb Synonyms and Verb Redirection

TADS has a couple of convenience features that make it easy to create certain common definitions.

First, TADS lets you define object-level verb synonyms. It's often desirable to make a number of verbs work the same way with a particular object, because different users will try to do the same thing with slightly different commands. For example, if you have a touch-pad on a control panel in your game, the player may try a number of different verbs with it: touch, push, tap, press. "Push" and "press" are already synonymous in adv.t (they both refer to pushVerb), but "touch" is a separate verb, and "tap" isn't even defined, so you would have to add this verb in your game program. Since "touch" and "tap" are separate verbs from "push", you will need to make the touch pad object re spond to all three verification methods and all three action methods the same way. You *could*do this with code like this:

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

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

This is tedious, though–especially if you have more than two or three verbs that you want to make synonymous. Instead of these lengthy definitions, you can use the TADS object-level verb synonym feature. To do this, you use the special pseudo-property doSynonym:

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

This simple definition has the same effect as the much lengthier definitions above. The way to read this definition is: the direct-object verb synonym for "touch" is "push".

You can specify that more than one verb is a synonym for another verb. To do this, simply list all of the different verb synonyms on the right-hand side of the equals sign:

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

Read this definition as: the direct-object verb synonyms for "touch" are "push" and "tap". Note that the fact that you can use multiple verbs on the right-hand side may make it easier to remember which way this direction works. All of the verbs in the co mmand are redirected to a single verb, so the "target"–the verb handler that's actually called when any of these verbs are used–must be in the position where only one verb can go, which is inside the parentheses. So, if the player types "push pad", the p arser will call verDoTouch and doTouch.

ioSynonym works the same way, but maps the verIo*Verb* and io*Verb* handlers of one or more verbs to those of another verb. Whereas doSynonym makes one or more verbs synonymous when the object is used as a direct object, ioSynonym makes verbs synonymous when the object is used as an indirect object.

Note that verb synonyms created with doSynonym and ioSynonym apply only to the object in which they are defined. A verb synonym created for one object doesn't affect any other objects, so "touch" and "push" are still separate verbs for other objects. Of course, verb synonyms are inherited just like any other verb handler.

The second convenience feature lets you specify that when a verb is applied to a particular object, it should instead be applied to a different object. This is often convenient when you are composing a complex object in your game from several internal obje cts. For example, you might want to include a desk with a drawer. The desk and drawer are separate objects, but the player may type "open desk" to open the drawer. You could code this by redirecting the verb verification and action methods for "open" from the desk to the drawer:

```
desk:  fixeditem
        noun = 'desk'
        sdesc = "desk"
        location = office
        verDoOpen(actor) =
        {
                deskDrawer.verDoOpen(actor);
        }

        doOpen(actor) =
        {
                deskDrawer.doOpen(actor);
        }
;
```

This works, but redirecting more verbs would be tedious–and you'd probably want to redirect at least "close", and probably "look in" and "put in" as well. To avoid the large amount of typing this would involve, you can use the verb handler redirection syntax:

```
doOpen -> deskDrawer
```

(Note that the symbol "-¿" is made up of a hyphen followed by a greater-than sign.) This single line replaces the verDoOpen and doOpen definitions above. It indicates that both the doOpen and verDoOpen methods, when called on the desk, should be redirected to the drawer.

2.2.20 Format Strings

For the most part, TADS doesn't treat the player actor's object, Me, any differently from any other actor in the game. When the player types a command that isn't explicitly directed to another actor, TADS assumes that the actor is Me, and from that point on acts the same as it would if the command were being directed to any other actor.

All of the command handling methods that the parser calls have the current actor object as a parameter, which allows them to be written independently of actor–or, if desired, to take special actions for certain actors. The verb handlers in adv.t generally treat all actors the same. For example, the definition of doTake in the class thing doesn't move the object being taken into the player's inventory, but rather into the current actor's inventory. This allows commands such as "take the box" and "joe, take the box" to be processed using the same code.

If a command handler is written independently of the actor, it needs to be able to display its messages in terms of the actor carrying out the command, rather than simply in terms of "you". It would be very inconvenient if you had to call actor.thedesc every time you wrote a message in a verb handler, 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 t he same thing any place where an actor other than the player may be taking some action that results in a message.

A format string associates a special character sequence with a property name. When a message includes a format string, enclosed in percent signs (%), the output formatter removes the format string and replaces it by calling the property associated with the format string. For example, one of the format strings defined in adv.t is:

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

This tells the system that whenever it sees "%you%" in a message being displayed, it removes the "%you%" sequence, and replaces it with the text that results from evaluating the property fmtYou in the current actor.

You can define new format strings of your own, although adv.t defines most of the format strings you'll probably need.

In adv.t, most messages are phrased in terms of format strings. For example, when the player tries to travel a particular direction but can't, adv.t doesn't display "You can't go that way", but instead displays "%You% can't go that way". When the player tries to go east but can't, the system displays "You can't go that way", just as it would have if the format string hadn't been used. But, when the player types "joe, go east", the system displays "Joe can't go that way", because joe.fmtYou displays "joe" rather than "you".

Note that the capitalization of the replacement text follows the capitalization of the format string itself. So, "%you%" becomes "you", while "%You%" becomes "You". ("Joe" is displayed capitalized in either case if the message displayed by joe.fmtYou is itself capitalized.)

Note that many format strings are defined for other parts of the sentence. When writing a message, you will usually have to use several format strings to keep the sentence grammatical. For example:

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

This message will become "You aren't wearing your spacesuit" when displayed for the player character, but will become "Joe isn't wearing his spacesuit" when displayed for Joe.

2.2.21 Compound Words

TADS can only handle prepositions that consist of a single word. Unfortunately, English and other human languages have constructions in which multiple words are used for the function of a single preposition in a sentence; for example, "take the book out of the bag" has two prepositions in a row, "out of", serving the function of a preposition in the sentence.

Although TADS doesn't have any way to specify a sentence template that uses multiple prepositions like this, it *does*provide a mechanism that lets you specify that certain pairs of words are "glued together" into a single word when they occur together. In the example above, you could define "out of" as such a pair of words: whenever the parser sees "out" followed directly by "of", it can be made to put these two words together and treat the pair as a single word.

To define a pair of words that should be treated as a single word, you use the compoundWord statement. For example:

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

This specifies that when the parser sees the word "out", followed directly by the word "of", it should take the pair of words and replace it with the single word "outof". (Note this replacement word follows the convention in adv.t, which is to simply conca tenate the two words to form the replacement word, but the replacement can be anything at all–you could have made the replacement word, for example, "out_of".)

Note that you must still define "outof" as a preposition. Even though "out" and "of" are separately defined as prepositions, the result of a compound word replacement must *also*be defined as a vocabulary word. Once the pair of words is replaced with the compound word, the parser forgets ever knowing about the pair as separate words.

You can't directly define three-word (or longer) compound words, but you can *indirectly*do so by constructing the longer words from two-word directives. For example, to convert "out from under" into a single preposition, you would use two compound word definitions:

```
compoundWord 'out' 'from' 'outfrom';

compoundWord 'outfrom' 'under' 'outfromunder';
```

When the parser finds the sequence "out from under", it first converts "out from" into "outfrom". Then, it checks the sentence again, and finds the pair "outfrom under", which is converts to "outfromunder".

2.2.22 Special Words

The parser treats a number of words as special reserved words. These words aren't ordinary vocabulary words, because they don't fit in to the parts of speech used for normal vocabulary words–they aren't nouns, plurals, adjectives, articles, verbs, or prep ositions. Since they aren't ordinary vocabulary words, you can't define them using the normal vocabulary definitions; instead, you use a separate command to specify these words: the specialWords statement.

This command can appear anywhere that one of the other special commands (such as formatstring or compoundWord) is allowed. Following the specialWords keyword, you list 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. You can provide more than one word for each position, by listing each synonym separated by an equals sign (=). The default list, which matches the built-in list if no specialWords command is used in your game, looks like this:

```
specialWords
        'of',
        'and',
        'then',
        'all' = 'everything',
        'both',
        'but' = 'except',
        'one',
        'ones',
        'it',
        'them',
        'him',
        'her',
        'any' = 'either'
;
```

Except for the words in the "of", "one", and "ones" positions, the words specified in the list can't be used as ordinary vocabulary words at all–they're always considered to have their special meaning as listed here. The slots for "of", "one", and "ones" are *not*reserved in ordinary use; their special meanings are in effect only when the parser expects them according to context.

Note that the slot occupied by "any" is a recent addition. You can omit this position in a specialWords, in which case the system will use the default ("any" and "either" will be used as synonyms for this position).

You can use the special replace and modify commands with specialWords. If your statement starts off with replace specialWords, any previous special words definitions are discarded, and the new set is used. (Note that the same thing happens if you specify a second specialWords directive in your game program with neither a replace nor modify prefix, but the compiler will warn you that you should use replace to make your intentions explicit.) If your statement starts off with modify specialWords, the compiler *adds*the special words in the new list to the words that have already been defined in previous specialWords statements. When using modify, you can use nil in any slot to indicate that you don't wish to add any words to that position. You can therefore use modify as a means of adding a new special word or words to one or more slots, without having to specify the entire list again.

2.2.23 Error Messages

When an error occurs during parsing, the parser tells the user what went wrong with as specific an error message as possible. One of the design goals of the parser is to be friendly to players; although error messages are inherently somewhat unfriendly, the y certainly can't be eliminated entirely, so we at least try to make them informative. The player should always be able to understand why the game rejects a command, so that he or she can figure out what to say instead.

The parser lets your game program provide a custom message for every error. Most errors are handled through the function parseError–if your game defines this function, the parser will call it whenever it wants to display an error. A few parser messages are more specialized, and use different functions that you can optionally provide.

The parser calls the parseError function with two parameters: a message number, and the text of the default message. The message number identifies the reason the parser is displaying a message–each situation that requires a messag e is given its own message number. The default message is the text that the parser would have displayed if you did not define a parseError function in your game at all.

You may have noticed references to message numbers elsewhere in this chapter–these were referring to parseError message numbers.

Your parseError function should return either nil or a (single-quoted) string value. If your parseError function returns nil, it means that you want to use the default message–this is the same as not defining a parseError function at all. If your function returns a string value, the string is displayed in place of the parser's default message.

Note that a few of the default messages contain the sequence "%s". This special sequence is replaced in the actual message displayed by a string value of some sort; for example, in message 2, the "%s" is replaced with the unknown word. Similarly, the specia l sequence "%c" in message is replaced with a single character, and "%d" is replaced by a number. You can use a "%" sequence in any message you provide to replace a message where the default text uses the same special sequence.

Messages with numbers below 100 are complete messages–they're not part of more complex messages built out of pieces. Messages 100 and above are different: they're fragments of complete messages; several such fragments are combined (sometimes with other t ext as well) to form a complete message. So, if you want to do some special formatting, such as enclosing any parser error messages in parentheses, you can easily do so for any of the regular messages, and avoid doing so for the complex messages, you can s imply check the message number to make sure it's less than 100, and apply your special formatting if so.

Most of the messages in the first group are self-explanatory, although a few require some explanation.

Messages 3, 10, and 11 are generated if a word the player uses refers to more than a fixed number of objects in the game. Note that the limit applies regardless of whether the objects are all present or not–the limit is for the total number of objects in t he entire game with the same vocabulary word. (For this reason, it's probably not a good idea to define a noun or adjective in a very general class such as "thing" or "item".) The limit in version 2.2 is 200 objects; prior to 2.2 it was 100 objects.

Message 9 is generated if the words the player uses to refer to an object are not all defined for at least one object. For example, if your game has a blue book, and a red box, the words "red" and "blue" will both be recognized as adjectives, and "book" and "box" as nouns; if the player types "look at red book", the phrase "red book" will be recognized as a valid noun phrase in form, but it doesn't refer to any object defined in the game. The parser will respond with message 9.

Messages 13 and 14 are used for pronouns. When the player refers to one of the singular pronouns (it, her, and him), and the object that the pronoun represents is no longer accessible, message 13 is used. Message 14 is used with the pronoun "them".

Message 15 is used when the player uses "all" as the object of a command, but there are no suitable objects present.

Message 28 is displayed when the player enters a command with multiple direct objects for a verb that requires the direct object to be disambiguated first. Verbs that process the direct object first accept only one direct object.

Message 30 is displayed when the player enters a command like "take 3 balls", but fewer such items are present.

When the player addresses a command to an actor, and the actor is visible to the player (the actor's isVisible(Me) returns true), but the actor is not a valid actor (the actor's validActor method returns nil), the parser displays message 31.

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.`
	`(Note: message 8 is no longer used.}`
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 are 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.`
27	`You can't repeat that command.`
28	`You can't use multiple objects with this command.`
29	``` I think you left something out after ``any of''. ```
30	`I only see %d of those.`
31	`You can't talk to that.`
32	`Internal game error: preparseCmd returned an invalid list`
33	`Internal game error: preparseCmd command too long`

```
34      Internal game error:  preparseCmd loop
```

The next set of messages is used to ask questions when the player refers to an object with words that could refer to more than one object. The parser must in these cases ask the player which of the possible objects was meant. Note that these messages will o nly be used if your game does not define a parseDisambig function, which will be used instead if defined.

```
100     Let's try it again:
101     Which %s do you mean,
102     ,
103     or
104     ?
```

The next set is used when an object is not suitable for a verb. These messages are needed when a player uses a verb with an object, but the object does not define (or inherit from a superclass) an appropriate "verification" method (verDo*Verb* or verI o*Verb*). Messages 111 and 114 consist of a single space.

Note that these messages are not used if your game defines a parseError2 function, since that function will be called instead to display this error.

```
110     I don't know how to
111
112     anything
113     to
114
116     .
```

The next message is used after each object when multiple objects are listed. For example, when the player enters the command "take all", the parser will apply the verb "take" to each accessible object in sequence; before applying the verb to each object, the parser displays the name of the object, followed by this message, so that the player can see the results of the command for each object.

```
120     :
```

The next messages are used to note objects being used by default. When the player types a command that omits an object, and the parser can determine that the verb implies that a particular object is being used, the parser will display the name of the object with these messages. For example, if the player types "dig", the parser might determine that the direct object is implicitly the ground, the preposition is "with", and the indirect object is implicitly the shovel; in this case, it will display message 130, then the name of the ground, then message 131; then it will display message 130 again, then the name of the assumed preposition, then message 132, then the name of the shovel, then message 131. Message 132 consists of a single space.

Note that these messages won't be used if your game defines a parseDefault function.

```
130     (
131     )
132
```

When the player leaves out the object in a command, but the parser is *not* able to find a suitable default object, the parser will ask the player to supply an object using these messages.

Note that these messages aren't used if your game defines a parseAskobj or parseAskobjActor function.

```
140     What do you want to
141     it
142     to
143     ?
144     them
145     him
146     her
147     them
148     What do you want
149     to
```

When the player uses words that could refer to more than one object, and the objects in question are visible but not accessible, the parser will call the cantReach method in each of the objects after displaying the name of the object and message 200.

```
200       :
```

Here's a sample of a parseError function that encloses any of the regular messages in square brackets. It ignores any messages that aren't in the simple message range (below 100), since messages outside this range are fragments of more complicated messages, so can't be formatted as though they were entire messages.

```
parseError: function(str, num)
{
        if (num < 100)
                return '[' + str + ']';
        else
                return nil;
}
```

2.2.24 Summary of Properties, Methods, Objects, and Functions

As described throughout this chapter, the parser calls certain methods that objects in your game can define, and calls certain functions that your game can provide. All of these methods and functions are described in detail in this chapter. This section s ummarizes these properties and function; for full information, refer to the appropriate section earlier in this chapter.

Properties and Methods

action(*actor*): Called in a deepverb object during command execution to execute a command with no objects.

actorAction(*verb, dobj, prep, iobj*): Called in the actor object during command execution.

adesc: Used to display an object's short description with an indefinite article.

adjective: Defines words as adjectives, and associates them with the object.

article: Defines words as articles.

cantReach(*actor*): Called in an object (only if cantReach isn't defined in the verb) when the object is visible but not accessible; this method is meant to display an explanation of this condition.

cantReach(*actor, dolist, iolist, prep*): Called in a verb when the list of objects (only one of *dolist* or *iolist* will be non-nil) is visible but not accessible; this method is meant to display an explanation.

construct: Called immediately after an object is created with the new operator. Minimally, his method should install the object into its container's contents list.

contents: The list of an object's contents.

destruct: Called immediately before an object is destroyed with the delete operator. Minimally, this method should remove the object from its container's contents list, and should make sure that the object is not referenced from any other lists or properties.

doAction: Direct Object Action; associated with a deepverb object Defines the verb template for a single-object command for a verb, and specifies the suffix for to be used for the action and verification methods for the verb (do*Suffix* and verDo*Suffix*, respectively).

dobjGen(*actor, verb, iobj, prep*): General direct object handler, called during command execution on the direct object prior to processing the verb's verification and action routines.

doDefault(*actor, prep, iobj*): Direct Object Default. Associated with a deepverb object. Returns the list of default objects for the verb.

ioAction(*prep*): Indirect Object Action; associated with a deepverb object. Defines the verb template for a two-object command for a verb with the given preposition, and specifies the suffix for to be used for the action and verification methods for the verb (io *Suffix* and verIo*Suffix*, respectively).

iobjGen(*actor, verb, dobj, prep*): General indirect object handler, called during command execution on the indirect object prior to processing the verb's verification and action routines.

ioDefault(*actor, prep*): Indirect Object Default; associated with a deepverb object. Returns the list of default objects for the verb.

isEquivalent: Used to determine if an object should be treated as equivalent to other objects of its immediate superclass. If this property is true, the parser treats all objects of the same immediate superclass as this object as interchangeable during disambiguation.

isHim: If true, the object can be the antecedant for the pronoun "him".

isHer: If true, the object can be the antecedant for the pronoun "him".

isVisible(*vantage*): Used to determine if an object is visible from the given object. The parser uses this to determine how to complain when an object isn't accessible: if it's visible, but not accessible, the parser will use cantReach to complain, otherwise it will simply say "I can't see that here".

location: The object which contains the object.

locationOK: If this property is true, it tells the compiler that you know that the location property is not an object constant, and the compiler therefore suppresses the warning it would otherwise generate.

multisdesc: Used by the parser to display the current object when iterating through a list of direct objects in a command.

nilPrep: Associated with a deepverb object; used by the parser to determine the preposition to use when a command of the form VERB IOBJ DOBJ is entered. If undefined, the preposition object associated with the word "to" is used by default.

noun: Defines words as nouns, and associates them with the object.

plural: Defines words as plurals, and associates them with the object.

preferredActor: Used by the parser to determine if an object is a "preferred" actor; if this property is true, the object is chosen as the actor in cases where the actor is ambiguous and none of the other objects are preferred actors.

prepDefault: Associated with a deepverb object. The parser uses this property to find the default preposition object to use when the player enters a partial command that starts with the verb and a direct object, but omits the preposition and indirect object.

preposition: Defines words as prepositions, and associates them with the object.

rejectMultiDobj(*prep*): Associated with a deepverb object. If this method returns true, the verb doesn't allow multiple direct objects or "all".

roomAction(*actor, verb, dobj, prep, iobj*): Called in the actor's location during command execution.

roomCheck(*verb*): Called in the actor at the beginning of execution, before object disambiguation.

sdesc: Short description. Used by the parser to display an object's name.

statusLine: Called in the actor's location to update the statue line.

thedesc: Displays the object's name with a definite article.

validActor: Called by the parser to determine if the object is valid as an actor. This method doesn't check to see if the object is *logical* to use as an actor, but simply if the object is *accessible* as the target of a command.

validDo(*actor, obj, seqno*): Associated with a deepverb object. Returns true if the object is accessible as a direct object for the verb.

validDoList(*actor, prep, iobj*): Associated with a deepverb object. Returns a list of possibly-valid direct objects for the verb. The objects returned will be tested for validity with validDo, so the list can contain invalid objects as well, but it must contain *all*of the valid direct objects for the verb.

validIo(*actor, obj, seqno*): Associated with a deepverb object. Returns true if the object is accessible as an indirect object for the verb.

validIoList(*actor, prep, dobj*): Associated with a deepverb object. Returns a list of possibly-valid indirect objects for the verb. The objects returned will be tested for validity with validIo, so the list can contain invalid objects as well, but it must contain *all*of the valid indirect objects for the verb.

value: The parser sets this property of the strObj or numObj object to the actual value of the string or number (respectively) entered in a command.

verb: Defines words as verbs, and associates them with the object. Strings in this list can consist of one or two words; the second word must also be separately defined as a preposition.

Objects

againVerb: The deepverb object with the vocabulary for "again". The parser internally handles the "again" processing when this verb's vocabulary word is entered as the command.

Me: The player actor object. When a command is not explicitly directed to another actor, this object is the actor to which the command is sent.

numObj: A special object which is used as the direct object when a number is entered in a command (such as in "enter 30 on calculator", in which numObj is the direct object). The value property is set to the number actually entered in the command.

strObj: A special object which is used as the direct object when a quoted string is entered in a command (such as in 'type "hello" on keyboard,' in which strObj is the direct object). The value property is set to the string actually entered in the command.

takeVerb: Formerly required, but no longer directly used by the parser.

Functions

commandPrompt(*type*): The parser calls this function, if defined, to display the prompt before asking for a command. The *type*code specifies the type of input being requested.

init: The system calls this immediately after loading the game, and again whenever the game is restarted. This function can perform any startup operations, such as setting fuses and daemons, and displaying an introductory message.

pardon: The parser calls this function whenever the player enters an empty command. You can display an error message, if you wish.

parseAskobj(*verb, ...*): The parser calls this function, if defined, to prompt for a missing direct or indirect object. If parseAskobjActor is defined, parseAskobj is ignored. If two arguments are provided, an indirect object is being requested, and the second argument is a preposition (which may be nil); otherwise, a direct object is being requested.

parseAskobjActor(*actor, verb, ...*): The parser calls this function, if defined, to prompt for a missing direct or indirect object. If three arguments are provided, an indirect object is being requested, and the third argument is a preposition (which may be nil); otherwise, a direct object is being requested.

parseDefault(*obj, prep*): The parser calls this function, if defined, to display a message indicating that the object (and the preposition, if not nil) are being used by default.

parseDisambig(*nameString, objList*): The parser calls this function, if defined, to prompt for more information when the player enters an ambiguous object name. The *nameString* is the string of words entered by the player (given as a single-quoted string), and the *objList* is a list of the objects matching the name.

parseError(*num, str*): The parser calls this function, if defined, to get the text of an error message. *num* is the message number, and *str* is the text of the message that the parser would display by default if this function weren't defined.

parseError2(*verb, dobj, prep, iobj*): The parser calls this function, if defined, to display a message indicating that the verb can't be applied to these objects (because one of the objects doesn't define or inherit a verification method for the verb).

preinit: Called by the *compiler*, after compilation is complete, to perform any desired initialization. This function can't set any fuses or daemons, or display any messages; these functions are not saved in the compiled game file, so they must wait until the init function, which is invoked at run-time. The preinit function is intended to perform any time-consuming computation–for example, setting up lists of objects or initializing property values–that would otherwise be done in init every time the game is started. By performing these functions in preinit, the work can be done at compile-time, saving players from having to wait for it each time they run the game. Note that this function won't be called until run-time if the game is compiled for debugging; while this doesn't affect the behavior of the game (apart from slowing down run-time startup), it allows you to use the debugger to step through your preinit function.

preparse(*cmd*): The parser calls this function with a (single-quoted) string giving the full text of the command line each time the player enters a new command.

preparseCmd(*wordList*): The parser calls this function with a list of (single-quoted) strings giving the words of the current command. This function is called for each individual command on a command line.