TADS
Version 2.1
Release Notes

TADS
The Text Adventure Development System
Version 2.1 Release Notes

Contents

Introduction

Since the initial release of TADS version 2.0, we have fixed many problems that users have encountered, and have added many new features. We haven't yet updated the TADS 2.0 Author's Manual to include documentation of the new features, so we've prepared this set of release notes to provide details on the additions and changes. Many of the features in TADS version 2.1 were added in one of the earlier 2.0.*x* maintenance releases, so you may have seen some of the information in these release notes before.

The TADS 2.0 Author's Manual remains the primary documentation for the system. We have tried to ensure that the changes we've made maintain compatibility with previous versions, which means that the information in the Author's Manual is still mostly correct.

You may notice that the TADS version numbers displayed by the compiler, run-time, and debugger now have greater precision than in the past: the version numbers now have *four* parts. The new fourth number indicates the platform-specific release level. Platform-specific releases will be made only to correct problems specific to a particular port of TADS, and will not involve any changes in the language, features, or file formats.

New keywords: `replace` and `modify`

Most game authors find that they can't avoid modifying `adv.t` in the course of writing their games. While there's nothing intrinsically wrong with this, it creates a problem when a new version of TADS is released: the game author must either continue to use the old version of `adv.t`, which means that any bug fixes or enhancements in the new version are not available, or take the time to reconcile the author's changes to the file with those made in the standard version.

We can't do anything to help you with changes you've made to `adv.t` prior to TADS version 2.1, but we may be able to prevent this from being a problem in the future, thanks to the new `replace` and `modify` mechanism.

These new keywords allow you to make changes to objects that have been previously defined. In other words, you can `#include` the standard `adv.t` file, and then make changes to the 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 the methods already defined in an object.

To replace a function that's already been defined, you simply preface your replacement definition with the keyword `replace`. Following the keyword `replace` is an otherwise normal function definition. The following example replaces the `scoreStatus` function defined in `adv.t` with a new function that customizes the status line score display.

```
#include <adv.t>

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

You can do exactly the same thing with objects. The following example entirely replaces a verb defined in `adv.t`.

```
#include <adv.t>

/* we don't want 'buckle' -- replace fastenVerb */
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 additional vocabulary does *not* replace the original vocabulary, but simply adds to the previously-defined vocabulary. Further note that verb association pseudo-properties, such as `doAction` and `ioAction`, are legal in a `modify` definition.

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 object directly; you can only refer to it indirectly through the new replacement 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` results in this display:

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

You can also replace a property entirely, erasing all traces of the original definition of a 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. In the example above, we could do this instead:

```
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.

`validDoList` and `validIoList`

Some game authors encountered a performance problem when defining a large number of objects with the same vocabulary. For example, if a game had thirty or forty objects that all had the noun `'button'`, the system took a long time to disambiguate a command such as "push button" even when only one button was present. The reason for the delay is that the run-time needed to run each object that matched the vocabulary through the `validDo` or `validIo` method (as appropriate) for the verb; when forty objects all had the same vocabulary, the parser had to call `validDo` forty times, resulting in lengthy parsing delays.

To improve run-time performance in these situations, the parser now calls a method in the verb object to reduce the number of objects that must be considered. The new method is `validDoList` for direct objects, and `validIoList` for indirect objects. This method returns a list of all of the objects for which `validDo` or `validIo` (respectively) would return `true`. The parser limits its checks to the objects returned by this new method; the result is that only a few objects typically need to be checked with `validDo` or `validIo`, eliminating the lengthy parsing delays for objects with common names.

The objects returned by `validDoList` and `validIoList` are matched against the vocabulary words used by the player, then tested through `validDo` or `validIo` as always. Note that this means that `validDoList` and `validIoList` can safely return *too many* objects—that is, they can return objects that would not actually pass `validDo` or `validIo` without causing any problem. This eliminates the need to perform time-consuming processing in these new routines, which further improves performance; the new listing methods can simply return the list of everything that's even remotely possible for the command.

The methods are called as follows:

```
verb.validDoList(actor, prep, iobj)
verb.validIoList(actor, prep, dobj)
```

The new `adv.t` provides suitable `validDoList` and `validIoList` methods for the verbs it provides. Generally, these methods return the list of all of the objects in the actor's inventory plus all their contents (recursively), plus all of the objects in the actor's location and their contents (recursively), plus all of the objects of class `floatingItem` in the game.

The new `floatingItem` class *must* be used with 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 would not be included in the lists returned by `validDoList` and `validIoList`. These methods include the list of all `floatingItem` objects, however, so declaring all such objects to be of class `floatingItem` ensures that they're accessible when appropriate.

One other feature of `validDoList` and `validIoList` is that returning `nil` is equivalent to returning a list of every object in the entire game. This means that if a verb doesn't define these methods at all, the behavior is compatible with prior versions of TADS before the introduction of these new methods. This is also useful for verbs such as "ask" and "tell" that should allow any object (even inaccessible objects) to be mentioned.

`dobjGen` and `iobjGen`

Sometimes, you want to be able to define an object that handles most or all verbs in the same manner. For example, if you wanted to define a distant object, too far away to be manipulated, you'd like the response to almost any verb (other than "look at") to be something like "It's too far away." In previous versions, it has been very tedious to implement such objects, because you had to provide a customized `verDo`*Verb* or other similar method for practically every possible verb.

To make this type of object easier to implement, the parser will now call a special method in the direct and indirect object early in the parser loop. The new property for a direct object is `dobjGen`, and for an indirect object it is `iobjGen`. These methods are called immediately after the `roomAction` method, and just before the first `verIo`*Verb* or `verDo`*Verb* method. These methods are called with these parameters:

```
dobj.dobjGen(actor, verb, iobj, prep)
iobj.iobjGen(actor, verb, dobj, prep)
```

Any parameters that aren't meaningful for the command (for example, `prep` and `iobj` for a command with no indirect object) will be passed as `nil`.

After calling these methods, the parser will proceed to the `verIo`*Verb* or `verDo`*Verb* method as normal. These methods should use `exit` if they want to stop the command without any further processing.

In certain cases, the parser does *not* call `dobjGen` or `iobjGen`. These methods will be called *only* when they effectively override the actual verb handler for the object—that is, when the verb handler is inherited, *not* defined directly in the object defining `dobjGen` or `iobjGen` (or by a subclass of that object). This allows you to exclude verbs from the general handling of `dobjGen` and `iobjGen`, simply by including specific handlers for those verbs in the object defining the general handler.

`dobjGen` and `iobjGen` make it very easy to define a distant object or a similar type of object. In the example below, every verb except for "look at" will respond with the message "It's too far away," because `dobjGen` or `iobjGen` will be called prior to any other processing on the verb. The reason that "look at" is not caught be `dobjGen` is that `verDoInspect` is defined in the object—this effectively overrides `dobjGen`. Any `verDo`*Verb* methods defined by subclasses of `distantItem` would have the same effect for those subclasses.

```
class distantItem:  item
  dobjGen(actor, verb, iobj, prep) =
  {
    "It's too far away.";
    exit;
  }
  iobjGen(actor, verb, dobj, prep) =
  {
    self.dobjGen(actor, verb, dobj, prep);
  }
  verDoInspect(actor) =
  {
    pass verDoInspect;
  }
;
```

New and improved built-in functions

Several new built-in functions have been added, and a few of the existing functions have been enhanced to provide additional features.

`getfuse(`*funcptr, parm*`)`: This function 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 (either it has already fired, or it has been removed with a call to `remfuse`), this function returns `nil`. Otherwise, it returns the number of turns before the fuse is activated.

`getfuse(`*obj, &msg*`)`: This form of `getfuse` lets you check on a fuse set with the `notify` function. If the fuse has already been fired, or has been removed with `unnotify`, this function returns `nil`. Otherwise, it returns the number of turns before the fuse is activated.

`gettime()`: Returns the current system clock time. The time is returned as a list of numeric values for easy processing.

[1]	year	calendar year (e.g., 1992)
[2]	month	month number (1 for January, 2 for February, etc.)
[3]	day	day of the month
[4]	weekday	day of the week (1 for Sunday, 2 for Monday, etc.)
[5]	yearday	day of the year (1 for January 1)
[6]	hour	hour of the day on 24-hour clock (midnight is 0)
[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

`inputkey()`: 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-standardized keys, such as cursor arrow keys and function keys. If the user uses a non-standard key, the return value is the representation of the key used by the computer being used. To ensure portability, you should use `inputkey()` only with standardized keys (alphabetic, numeric, and punctuation 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.

`intersect(`*list1, list2*`)`: 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:

```
[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 future versions.

`objwords(`*num*`)`: 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 punctuation). 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.

If the player types "take all," then `objwords(1)` will return `['all']` and `objwords(2)` will return `[]`.

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

If multiple direct objects are used, 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 determine much more precisely what the player is asking about than would otherwise be possible.

`outhide(`*flag*`)`: 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 is not seen by 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.

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

`restart(`*funcptr, param*`)`: This new form of `restart()` allows you to 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 does not return, and all game state is reset to its initial state by `restart()`. You can use this 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; 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.

The *param* value is simply passed as the parameter to the function to be called; this allows you to pass information through the reset. 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()` that makes use of the parameter information, you can simply store the necessary information in `global.initRestartParam` to ensure that it's passed to your function at the appropriate time.

`restore(nil)`: 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 started 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`.

Currently, it is 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, and 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 feature on the Macintosh 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 is restored. If your game has extensive introductory text, you could call `restore(nil)` (and `return` if the function returns `nil`) prior to displaying the introductory text, since the player has presumably already seen it anyway.

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 wish to display your copyright message, or ask a question for copy protection, every time the game starts, even when a saved game is going to be restored.

If you do not make a call to `restore(nil)` in your `init` function, the system will automatically restore the saved game specified by the player at startup immediately after your `init` function returns. Hence, omitting the call to `restore(nil)` does no harm; it merely delays the restore.

`rundaemons()`: Runs all of the daemons. The function runs daemons set both with `setdaemon()` and `notify()`. This function returns no value.

`runfuses()`: Runs all expired fuses, if any. Returns `true` if any fuses expired, `nil` otherwise. This function runs fuses set both with `setfuse()` and `notify()`.

`setit(nil)`: You can now use `nil` as the argument to `setit()`. This prevents the player from using "it" in subsequent commands.

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

`setit(`*list*`)`: 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 clears "it."

Prompt customization

You can now customize the appearance of the command prompt. If you provide a function named `commandPrompt`, the system will call this function rather than displaying its usual `>` prompt. The function is called with a single argument, a number, which indicates what type of prompt should be displayed.

The possible values of the prompt type parameter are:

0	normal command
1	command after an invalid word (allowing “oops” to be used)
2	disambiguation (after “which *obj* do you mean...”)
3	command after `askdo` (after “what do you want to *verb*?”)
4	command after `askio`

Note that the default prompt is the same in every case; the reason that `commandPrompt` receives the information on the type of prompt is that you may want to use a custom prompt that would not be appropriate in all cases.

This function has no return value. If the function is defined by your game, no default prompt is ever displayed regardless of whether your `commandPrompt` displays anything or not. So, if you provide this function at all, it must handle all cases. Note also that the blank line that the system displays before each prompt is part of the default prompt, and therefore is *not* displayed if a user `commandPrompt` function is defined. If you want to write a version of this function that emulates the default prompt, it would look like this:

```
commandPrompt:  function(arg)
{
  "\b>";
}
```

National Language Support

We’ve made several enhancements to the system to better support non-English character sets. The most basic improvement is that the system should now be fully “8-bit clean,” which means that extended 8-bit character sets (which use character codes above 127) should work just like normal ASCII characters in source code, displayed text, vocabulary words, and player commands. Note that extended characters used in player commands are treated the same as alphabetic characters; no provision has been made for extended punctuation codes, so players will still have to use the standard ASCII punctuation characters in their commands.

We have also made it possible to use 16-bit characters in displayed text. For the most part, this has always been possible, because the display of 16-characters is done by the operating system or video device. However, one problem that some people encountered is that certain 16-bit character sets include the code for the backslash as one of the two bytes of certain characters; this created a problem, because the TADS output formatter interpreted the backslash byte and the following byte as a formatting code sequence. To address this problem, we've added a new code sequence to the output formatter: `\-` (a backslash followed by a hyphen). This sequence tells the formatter to pass the following two bytes as-is, without any interpretation. If you wish to display a 16-bit character that contains a backslash as one of its bytes, simply prefix the 16-bit character with the `\-` sequence; this will prevent the formatter from interpreting either of the bytes of your 16-bit character.

In addition, we have expanded the functionality previously provided by the `parseError` function. Some users have found that way that `parseError` was used to construct certain messages did not provide enough information to allow proper translation into non-English languages. To improve translatability of the parser messages, we have added several new functions that are similar to `parseError`. As with the other optional user-provided functions, the system will use its old default behavior if these functions are not provided. The functions are independent, so you can provide or omit them in any combination.

The new function `parseError2` is called to generate the default parser message stating that the verb attempted isn't accepted by the objects involved; this happens when either the indirect object doesn't define an appropriate `verIo`*Verb* method, or the direct object doesn't define an appropriate `verDo`*Verb* method. The `parseError2` function receives four arguments: the verb object, the direct object, the preposition object, and the indirect object; however, only one of the direct or indirect object will be non-`nil`. If the indirect object is `nil`, the preposition will be `nil` as well. The verb will never be `nil`. Note that the preposition may be `nil` even when the indirect object is not, in which case you should assume that the preposition is "to." The function below implements the default behavior that the system uses when `parseError2` is not defined by your game.

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

The parseDefault function is called when the parser is assuming a default object. This function receives two arguments: the object being defaulted, and the preposition object. If a default direct object is being assumed, the preposition will be nil; if an indirect object is being defaulted, the preposition argument will be the preposition object associated with the preposition in the player's command. The implementation below provides the default behavior.

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

The parseDisambig function is called when objects need to be disambiguated. This function should generate an appropriate question to ask the player which of several objects is intended. The function receives two arguments: the string that the user typed that is in need of disambiguation, and a list of objects that match the string. The implementation below emulates the parser's default behavior.

```
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 ";
  }
  "?";
}
```

The parseAskobj function is called when the parser needs to ask the player for a direct or indirect object to complete a command. For example, if the player types "take," and several objects are present that could be taken, the parser must ask the player what to take. The parseAskobj function can be called with either one or two arguments, so it should be declared using the variable argument list notation. When the parser wants to ask for a direct object, parseAskobj will be called with only one argument: the verb object. When the parser is asking for an indirect object, the function will be called with two arguments: the verb object, and the preposition object. Note that the preposition could be nil, in which case you should assume that the preposition is toPrep. The example below implements the parser's default behavior.

```
parseAskobj:  function(v, ...)
{
  "What do you want to <<v.sdesc>>";
  if (argcount = 2)
  {
    local p := getarg(2);
    " it << p ?  p.sdesc :  "to" >>";
  }
  "?";
}
```

Debugger Enhancements

The debugger has a new set of commands that allows you to 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 lower-level classes (such as those from adv.t) interact. Four new commands have been added to support this new feature; these are also accessible from the "Execution" menu on the Macintosh.

- `c+` Turn on 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-` Turn off call logging.
- `cc` Clear current call log.
- `c` Show current call log. All function and method calls after the most recent `c+` (up until the current time or the most recent `c-`) are displayed.

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, because the system must do extra work every time a function or method is entered and exited. You should enable call logging at the point you're about to execute a command that you want to trace through its execution, 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 the new information. If you leave call logging activated for an extended period of time, information toward the beginning of the log may be lost.

The lines of text displayed in the call log will be indented to show nesting. The functions and methods 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 when the function or method returns. Each call will show the object and method or function name involved, along with the arguments; the format is the same as in the stack trace.

In addition, the debugger has several small improvements:

- The debugger will get 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 command will be aborted (as though an `abort` statement had been executed).

- It is now possible to break out of an infinite loop in your game. Hit Ctrl-Break on a DOS machine, and Command-period on a Macintosh, if your game goes into an infinite loop. Control will return to the debugger.

- 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`.

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

File Format Compatibility

Version 2.1 of TADS uses a slightly different game file format than previous versions. The new format will *not* be acceptable to older versions of the run-time, although the new run-time is able to read `.GAM` files produced by older versions of the compiler.

The version 2.1 compiler generates `.GAM` files with format "B"; prior versions generated format "A." The version 2.1 run-time is able to read files in either format; prior versions can read only "A."

If for some reason you wish to generate a `.GAM` file that can be read by an older version of the run-time, the compiler has a new option, `-fva` (Format Version "A") which generates `.GAM` files in the older format. Unless you have a specific need to generate the older format, we recommend using the newer format (which the compiler will use by default). Another switch, `-fvb`, is provided to tell the compiler explicitly to use format "B"; and `-fv*` tells the compiler to use the most recent format (currently "B").

Note that some incompatible file format changes have been made in past versions in such a way that the run-time is unable to detect the incompatibility. It is therefore not always safe to mix different versions of the compiler and run-time with versions prior to 2.1.

One of the changes to the game file format makes the files much more compressible with archiving and compression utilities. The `.ZIP` and `.SIT` files that you make from your `.GAM` files should now be much smaller.

New compiler options

Several new compiler options have been added. Macintosh users will find new menu choices that provide access to these options.

`-v` *level* — Set warning verbosity level. The default level is 0 (minimum verbosity; suppresses certain warnings which are either purely informational or generally do not indicate an actual problem). Other levels currently available are 1 (suppress purely information messages, but display warnings even when they generally do not indicate a problem), and 2 (maximum verbosity; displays all messages).

`-fv` *type* — The *type* can be `a` for format "A" (used by TADS versions prior to 2.1), `b` for format "B" (used by version 2.1.0), or `*` for the most recent version. The default is `-fv*`, which currently selects format "B." Unless you have some specific reason for using an older file format, you should use the most recent format (`-fv*`).

`-e` *file* — Log errors to *file*. All error messages generated during compilation will be saved 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, or overwritten with the new error log if it already exists.

Improved error recovery

Several improvements have been made to the compiler's error reporting and recovery mechanism. The changes are intended to make it easier for you to track down syntax and other errors in your source based on the error information generated by the compiler.

- New logic has been added to help you track down unterminated strings. When the compiler encounters a semicolon or a right brace ("`}`") in the first column of a line while inside a string (either double-quoted or single-quoted), it will generate a warning message that you have a possible unterminated string. This is purely a guess by the compiler, but if you are careful to format your code using the same convention as `adv.t`, in which every function ends with a right brace on a line by itself, and every object ends with a semicolon on a line by itself (without any spaces preceding it), the compiler will be able to find your unterminated strings almost every time. Note that the unterminated string will often not be the last string, but this warning will at least isolate the object or function where the string is coded.
- The compiler will now generate an error message if you attempt to use `self` in a function. This has always been illegal, but previous versions of the compiler did not detect the error, so the problem was not detected until the code was actually executed.
- The warning messages that the compiler generates for optional objects and functions (such as `parseError` and `commandPrompt`) are now suppressed at warning verbosity levels less than 2. Many users found these warnings confusing or annoying, and they almost never actually indicated a problem in your game, so they are no longer generated unless you specifically ask for them.
- The warning message for multiple inclusions of the same file is now suppressed at verbosity levels less than 1. This message is almost

always spurious if you use precompiled headers, because it is generated for every header that you precompiled. You will no longer see this warning unless you specifically ask for it by setting a higher warning verbosity level.

adv.t changes

Several improvements have been made to `adv.t`. Most are corrections, but some functional improvements have been made, and a few new items have been added.

- A new class, `distantItem`, has been added. This class is essentially the same as a `fixeditem`, except that it's intended to be used for objects that are not actually part of a room, but are *visible* from the room. For example, the player might be able to see a distant mountain from a certain location, even though the player can't do anything to the mountain (except look at it). A `distantItem` object can be inspected, but any other attempts will receive the response "It's too far away."
- A new function, `scoreStatus(`*points, turns*`)`, has been added, which simply calls `setscore()` with the same arguments. All other calls to `setscore()` in `adv.t` have been replaced by calls to `scoreStatus()`, which makes it easy to provide a new scoring format simply by using the `replace` keyword to substitute your own implementation of `scoreStatus()`.
- 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 `statusPrep = "in"` and `outOfPrep = "out of"`. The class `beditem` provides default values of `"on"` and `"out of"` instead. If you're defining new subclasses of `nestedroom`, you can override these properties to provide the most appropriate messages for the subclasses.
- The `fixeditem` class has been corrected so that a `fixeditem` cannot be thrown at anything.
- The `follower` class's `actorAction` method now appropriately executes an `exit` statement. In addition, the `follower` class now uses `dobjGen` and `iobjGen` to provide more sensible responses to most verbs.
- The `clothingItem` class has been modified so that "get out of" is equivalent to "take off."

- The various verbs which use vocabulary including "look" plus a preposition have been modified to allow "l" in place of "look"; this applies to several verbs, including "look at," "look on," "look in," "look under," "look around," and "look through."

- The `doDefault` method of `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 closed.

- The `vehicle` class has been corrected so that the player cannot generally manipulate the vehicle while occupying the vehicle. For example, the player cannot now take a vehicle or put it anywhere while inside it.

- 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. The `initRestart()` in `adv.t` simply sets the property `global.restarting` to `true`. Your game can inspect this flag in the `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 the `adv.t` implementation of the function.

- 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 `init()`. Note that the call to `restart()` passes `global.initRestartParam` as the parameter to the `initRestart` function. If you replace `initRestart()` with your own function, and you need to pass some information to this function, simply store the necessary information in `global.initRestartParam` at any time before restarting, and the information will automatically be passed to `initRestart()` when it's invoked.

DOS Color Customization

The player can now customize the colors used by the runtime. A small new program, `TRCOLOR`, is provided to set up the runtime screen colors. The program is self-explanatory (it displays instructions on-screen, and its operations are very simple). Type `TRCOLOR` at the DOS prompt to run the program.

Once you've selected your color scheme, the `TRCOLOR` program will create a small file called TRCOLOR.DAT in the current directory. The runtime will read 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 containing 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.

Improved DOS MAKETRX Interface

The user interface of the DOS MAKETRX program has been improved. 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 use convenient new defaults.

First, you can now omit the extensions on all of the arguments. The extension assumed for TR.EXE 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 use reasonable defaults. If you omit the name of the TR.EXE program, MAKETRX attempts to find TR.EXE in the same directory as MAKETRX.EXE; so, if you simply keep all of your TADS executables in a single directory, you won't need to specify the location of TR.EXE when running MAKETRX. If you omit the name of the destination file, MAKETRX will use the same name as the game file, with the extension replaced by ".EXE".

The 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.exe mygame myprog
```

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