David Betz

An Adventure Authoring System

A tour of AdvSys, a tool for writing text adventure games

[Editor's note: *"Interactive Fiction as Literature," a companion piece to this article, begins on page 135.*]

AdvSys is a system I designed for writing text adventure games. In adventure games, the player acts as an adventurer in a simulated world (real or imaginary). Players determine their own course of action by typing commands that trigger events in the simulated world.

You can approach the writing of an adventure game in many ways, and a number of books describe how to use traditional programming languages to write adventures. The most commonly used language is BASIC. But while you can certainly build very complex and interesting adventures using BASIC, it was not designed specifically for that purpose.

Much of the task of building an adventure game program consists of constructing complex data structures that model the game universe. BASIC has no convenient means for describing these data structures. Even Pascal, which is rich in data structuring facilities, has no easy means of constructing complex initialized data structures.

Another approach to writing adventures is to use a special language specifically designed for the purpose. This article describes such a language.

A language for writing adventures must have three essential features: a parser to handle commands typed by the player, an object-description facility, and a language for specifying the events that take place in response to the players' commands.

The Parser

Adventure games generally take place in a world made up of a network of interconnected 'locations.'

The parser is responsible for prompting the player to enter a command. It must read the command from the keyboard and break it into pieces that can be digested by the action code. All commands are broken into one of five different types of phrases:

1. an actor phrase
2. a verb phrase
3. a list of direct-object noun phrases
4. a preposition
5. an indirect-object noun phrase

Not all of these phrases are present in every command, and the parser recognizes only a limited set of command forms. AdvSys recognizes these forms:

1. [actor,] verb
2. [actor,] verb direct-objects
3. [actor,] verb direct-objects preposition indirect-object
4. [actor,] verb indirect-object direct-objects

where "direct-objects" is defined as:

direct-object [conjunction direct-object]*

(In this article, phrases within square brackets are optional. Phrases followed by an asterisk may be repeated zero or more times.)

The terms "actor," "direct-object," and "indirect-object" all represent noun phrases. Each noun phrase is of the form

[article] adjective* noun

In other words, a noun phrase consists of an optional article followed by zero or more adjectives followed by a noun. Here are some examples of noun phrases recognized by the AdvSys parser:

sword
the angry man
the thick red book

Now that we have defined the command forms that are handled by the parser, let's look at some complete commands. I will precede each example with a number indicating the form on which it is based:

1. Look
1. Fred, wake up
2. Drop the book
2. Take the sword and the orange vial
3. Give the book to the librarian
4. Librarian, give me the book
4. Show the librarian the book

The second example illustrates another feature of the parser. Verb phrases can consist of either a single word like “take” or a pair of words like “pick up” or “wake up.” If a verb phrase consists of two words, the words do not have to be immediately adjacent to each other in the command. Either “Pick up the book” or “Pick the book up” will produce the intended result.

After breaking the command into phrases, the parser sets a small number of global variables to communicate the results of its work to the rest of the adventure system. The parser stores each noun phrase in an internal array, indexed by the noun phrase number. The index associated with the actor noun phrase is stored in the global variable `$actor`, the index associated with the first direct-object noun phrase is stored in the variable `$dobject`, and the index of the indirect-object noun phrase is stored in the variable `$iobject`. If a noun phrase is missing from the command, its corresponding variable is set to `nil` (which is the same as zero in this system). These noun phrase numbers will be used later to determine which objects the noun phrases refer to.

The parser uses the verb phrase and the preposition to select an action to handle the command. It stores the selected action in the global variable `$action`. (More details about actions later.) The adventure language statements you use to define the vocabulary used by the parser are

```
(adjective word*)
(preposition word*)
(conjunction word*)
(article word*)
```

In addition, it is useful to define synonyms of some words. This is accomplished by the statement

```
(synonym word synonym*)
```

Objects

Adventure games generally take place in a world made up of a network of interconnected “locations.” Each location has a set of exits that connect it with adjacent locations. The player explores the game world by moving an actor from location to location through these exits.

In the course of exploring these locations, the player encounters “things” and other “actors.” In a fantasy adventure, for example, the player might encounter a magic sword or an angry dwarf.

AdvSys groups locations, actors, and things in the general category of *objects*. Each object in the adventure has an associated set of *properties*. Each property has an associated *value*.

A location object, for example, has a property for each of its exits. The values of these exit properties are the locations a player will reach by passing through the corresponding exits. Location objects also have *description* properties whose values are text strings describing the location under different circumstances. The concept of objects with properties is very general in this system, leaving you, as you write your game, free to invent new properties appropriate to a particular type of game and to define new classes of objects that share common properties, structure, and behavior.

For instance, a location object might be defined as

```
(location living-room
   (property
```

```
   description "You are in the living room.\n"
   north library
   south entrance
   east dining-room))
```

This is a definition of `living-room`, a location object with the properties `description`, `north`, `south`, and `east`. The description property has the string `"You are in the living room.\n"` as its value. (A back slash followed by the letter `n` instructs the program to start a new line). The property `north` has the value `library` (the lcoation the player reaches when traveling north from the living room), the property `south` has the value `entrance`, and the property `east` has the value `dining-room`.

Things are objects that the player can manipulate. A thing must have a noun associated with it. And since the same noun can refer the different objects, you can associate adjectives with the objects to make references to the objects umambiguous. Here is an example of an object description:

```
(thing sword
   (noun sword weapon)
   (adjective red)
   (property
      description "a red sword"
      weight 20
      value 10
      initial-location entrance))
```

This definition describes the object `sword`, which has the nouns `sword` and `weapon` and the adjective `red`. Thus, a player could refer to this object as “the sword,” “the weapon,” “the red sword,” or “the red weapon.”

The `sword` object also has four properties: `description` has the value `a red sword`, `weight` has the value `20`, `value` has the value `10`, and `initial-location` has the value `entrance`.

The `weight` property might be used to provide the player with a limited carrying capacity. If each object has a weight, you can prevent the player from carrying a set of objects whose combined weight exceeds the player's load capacity. Similiarly, the `value` property could be used for scoring. Each object has an associated value that the game will add to the player's score when the object is carried to some specified location. The meaning of these properties is up to you, the game author.

Defining Objects

Actors are objects that represent characters in the adventure. The player controls a special actor that is the “player character.” The player “sees” through this actor's eyes and takes part in the action by controlling this actor's movements. In AdvSys, the player character is called the “adventurer.” Other actor objects represent nonplayer characters. These nonplayer characters are controlled by the adventure program (the code that you have written) rather than by the player, and they may include both friendly and hostile characters with whom the adventurer must interact to solve the adventure. An example of a nonplayer character might be

```
(actor troll
   (noun troll dwarf)
   (adjective angry)
   (property
      description "There is an angry troll here."
      short-description "an angry troll"
      initial-location "dungeon"))
```

This defines a `troll` that the player can refer to as “the angry troll,” “the dwarf,” and so on, and is initially found in the dungeon, where the adventure will see the words `"There is an angry troll here"` upon entering.

So far we have seen how to describe the static portions of an adventure game. Location objects allow us to build the adventure universe, things allow us to place interesting objects in this universe, and actors allow us to populate the universe with other characters.

I have defined locations, things, and actors as part of a run-time package that comes with AdvSys, but you can define your own objects if you wish. The statements used to define these objects, as shown in the previous examples, are

```
(object object-type
   object-statement*)
```

```
(object-type name
   object-statement*)
```

where the `object-statement` may be defined using one of the following:

```
(noun word*)
(adjective word*)
(property [property-name initial-value]*)
(class-property [property-name initial-value]*)
(method (selector arg-name* [&aux tmp-name*]) expression*)
```

Now we will explore how things happen in the adventure universe.

Handlers

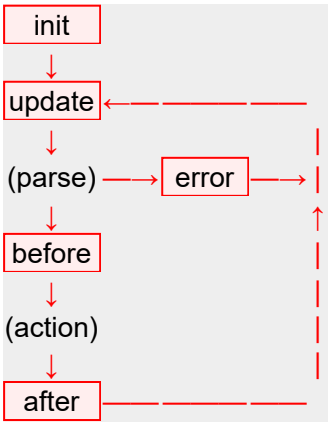

Figure 1: *The action in an AdvSys game is controlled by "handlers," described below.*

All action within an AdvSys adventure is controlled by a set of "handler" and "action" procedures. There are five different handlers that are part of the main control loop. Each of these handlers contains user-defined code written in the adventure language. Figure 1 illustrates the control flow of the adventure system.

At the beginning of the game, the AdvSys interpreter calls the "init" handler. The init handler is responsible for printing any introductory text explaining the initial situation and for performing any initialization. Here is an example of an init handler definition:

```
(init
   (print "Welcome to the sample adventure!\n")
   (setq curloc nil))
```

This example handler prints a welcome message and sets the variable `curloc` (the current location of the adventurer) to `nil`.

The first handler in the main loop is the "update" handler, which is responsible for handling changes in the game state. If the player has moved to a new location, the update handler should print a description of the new location. Here is an example:

```
(update
   (if (not (= (getp adventurer parent) curloc))
      (progn
         (setq curloc (getp adventurer parent))
         (send curloc describe))))
```

This handler checks to see if the adventurer's new location is different from the current one. If it is, the handler updates the current location and prints a description of the new location by sending the message `describe` to the new location object. (Note that on the first pass through the control loop, the update handler sees the location of the adventurer as being different from that stored in `curloc` and prints a description of the initial location.)

After the update handler has finished, the AdvSys interpreter calls the parser. The parser prompts the player for a new command, allows the player to enter the command, and parses the command according to the description above. The parser communicates its results to the remaining handlers by setting the global variables `$actor`, `$action`, `$dobject`, `$ndobjects`, and `$iobject`. If an error occurs during the parsing of the command, the system prints an error message, calls the error handler, and goes back to the start of the main loop (the update handler).

Assuming that the parser succeeds in parsing a syntactically valid command, the AdvSys interpreter calls the "before" handler, which handles any general preprocessing that needs to be done before the command-specific code is performed.

Next, the action associated with the player's command is performed. This is the action that was stored in the global variable `$action` by the parser. This code is responsible for actually carrying out the player's request (if it is allowed in the current situation).

The last handler in the main loop is the "after" handler, which handles any processing that must happen after the action is complete, such as updating the game clock or the player's score, or anything that should happen only at the end of a successful turn.

The adventure language statements that are used to define handlers are:

```
(init expression*)
(update expression*)
(before expression*)
(after expression*)
(error expression*)
```

Actions

The only part of the adventure system left to describe is the method for defining actions. Each action definition handles a specific command form and verb/preposition combination. Let's look at an example:

```
(action a-take
   (verb take get (pick up))
   (direct-object)
   (code
         (setq %dobject (in-location $dobject))
      (if (getp %dobject takeable)
         (progn
            (if (send %actor carrying? %dobject))
               (complain "You are already carrying the " $dobject "!\n"))
               (send %actor take %dobject)
               (print-noun $dobject)
               (print " taken.\n"))
         (complain "You can't take the " $dobject "!\n"))))
```

This action definition handles commands like “take the lamp” or “pick up the sword.” It handles any command with the verbs “take,” “get,” or “pick up” and at least one direct object. The code begins by determining to which object the direct-object noun phrases refers. The function `in-location` looks for an object in the current location (`curloc`) that matches the noun phrase it receives as its argument. The function returns the matching object or signals an error if no object in the current location matches the noun phrase.

Assuming that `in-location` finds a matching object, the `action` code assigns that object to the variable `%dobject` and then checks to see if it is possible to pick up the object. This is done by checking the value of the property `takeable`. If the result is `true`, the object may be taken. If not, the program prints an error message and the turn ends.

If the object is takeable, the code then checks to see if the player is already carrying it. It does this by sending the message `carrying?` to the `actor` object. This message checks to see if the object is currently being carried by the actor receiving the message. Both actor and location objects support the concept of containment. If the adventurer is already carrying the object, the program prints a message saying so and the action is complete. If not, the program adds the object to the adventurer's inventory by sending the `actor` object the `take` message and the program prints a message indicating the successful completion of the command.

This example illustrates the use of object-oriented programming techniques in the specification of action code. AdvSys lets you define “methods” to handle messages sent to objects. Each message requests that the object perform some operation. The specific operation that is performed in response to a message is determined by a method definition associated with the object that receives the message. AdvSys supports hierarchical inheritence of both properties and methods so you can define classes of objects that share the same property structure and behavior.

The default run-time environment in AdvSys adventures (contained in the file OBJECTS.ADI) defines methods that implement the default behavior for the built-in object classes. But the power of the system is that it lets you define subclasses of these default classes that implement either objects or classes of objects whose behavior and properties are unique to a particular adventure. This allows you to build on the existing classes rather than “reinventing the wheel.” The default run-time environment is thus a framework for building an adventure rather than merely a sample program.

If a command contains multiple direct objects, the parser will store the first direct object noun phrase number in `$dobject` and the number of direct objects in `$ndobjects`. If your action code doesn't touch the value of `$ndobjects`, at the end of the after handler, the system will assign the next direct object ot the variable `$dobject`, decrement the count stored in `$ndobjects`, and loop back to the before handler again. This means that you don't need to do anything special to handle commands with multiple direct objects. However, if you have a reason to want to handle all of the objects yourself, you can do so on the first pass through the action code and then set the variable `$ndobjects` to nil to prevent the system from looping back to handle the additional direct objects.

The adventure language statements used to define actions are:

```
(action name
   action-statement*)
```

with `action-statement` defined as one of the following:

```
(actor [flag])
(verb [word | (word1 word2)]*)
(direct-object [flag])
(preposition word*)
(indirect-object [flag])
(code expression*)
```

and where `flag` is one of the following:

```
required
optional
forbidden
```

Expressions

Handlers, actions, and methods all contain executable statements called expressions. The complete list of expressions allowed in AdvSys is shown in table 1. Each of the forms shown computes a result, which is returned as the value of the expression.

Here is an example of an expression derived from table 1:

```
(setq x (+ (* a b) (* c d)))
```

This executable statement is an arithmetic expression that multiplies `a` times `b` and `c` times `d`, adds the two products, and stores the result in the variable `x`. Even the `setq` form returns a value. Its value is the new value of the variable after the assignment is done.

Table 1: *The executable statements used in actions and handlers to control an adventure game written with AdvSys.*

(+ expr1 expr2)	Add two numbers	(expr [expr]*)	Call a user-defined function
(− expr1 expr2)	Subtract two numbers	(class obj)	Get the class of an object
(* expr1 expr2)	Multiply two numbers	(send obj sel [expr]*)	Send a message to an object
(/ expr1 expr2)	Divide two numbers	(send-super sel [expr]*)	Send a message to the superclass of an object
(% expr1 expr2)	Remainder after dividing two numbers	(randomize)	Initialize the random-number generator
(& expr1 expr2)	Bit-wise AND of two numbers	(rand expr)	Generate a random number between 0 and n-1
(: expr1 expr2)	Bit-wise OR of two numbers	(yes-or-no)	Prompt the user and accept YES or NO
(expr)	Bit-wise complement of a number	(print expr)	Print a string
(< expr1 expr2)	Is expr1 less than expr2?	(print-number expr)	Print a number
(= expr1 expr2)	Is expr1 equal to expr2?	(print-noun expr)	Print a noun phrase
(> expr1 expr2)	Is expr1 greater than expr2?	(terpri)	Start a new print line
(setq sym value)	Set the value of a variable	(match np obj)	Does this noun phrase match this object?
(getp obj prop)	Get the value of a property	(finish)	Finish this turn (go to the AFTER handler)
(setp obj prop val)	Set the value of a property	(chain)	Exit this handler and go to the next
(and [expr]*)	Logical AND of a set of expressions	(abort)	Abort this turn (go to the UPDATE handler)
(or [expr]*)	Logical OR of a set of expressions		
(not expr)	Logical NOT of an expression		

(cond [clause]*)	LISP style conditional	(exit)	Exit the adventure (back to DOS)
(if expr then-expr [else-expr])	Traditional “IF” statement	(save)	Save the current game state to a file
(while expr [expr]*)	Traditional “WHILE” statement	(restore)	Restore the game state from a file
(progn [expr]*)	Group expressions into a block		
(return [expr])	Return from a function		

Run-Time Functions

Not all of the functions that I have used in the examples are listed in table 1. The missing functions are part of the run-time package OBJECTS.ADI (see table 2) and are not built into the language. These functions are defined in adventure code and can be changed by the adventure author to suit a variety of tasks.

I wrote the run-time package so that you would not have to start from scratch when writing adventures. The run-time package defines commonly used object types such as locations, actors, and things; common actions such as look and take; game control commands like save and restore; and methods for handling common messages. These methods define the default behavior of objects, but can be easily supplemented or overrideen by methods defined in objects or subclasses defined by the adventure author. In this way, you can build your adventure game on the basic framework provided by the run-time package. For example, the run-time package supplies a method for the message `leave`, which allows an actor to leave a location.

Table 2: *The functions included in the basic run-time package for the AdvSys adventure writing system, OBJECTS.ADI.*

BASIC-THING	
superclass:	
object	
properties:	
initial-location	Initial location of the object.
parent	Current location of the object.
sibling	Next object in the location.
methods:	
(none)	

ACTOR	The adventurer or non-player characters.
superclass:	
basic-thing	
properties:	
child	First object carried by the actor.
methods:	
(move dir)	Move in the specified direction.
(take obj)	Take an object.
(drop obj)	Drop an object.
(carrying? obj)	Is actor carrying the specified object?
(inventory)	Show the actor's inventory.

PORTAL	Connections between locations.
superclass:	
basic-thing	
properties:	
other-side	Counterpart in the other location.
closed	Is it closed?
locked	Is it locked?

THING	Things that can be taken.
superclass:	
basic-thing	
properties:	
takeable	Can the thing be taken? (defaults to T)
methods:	
(none)	

STATIONARY-THING	Things that cannot be taken.
superclass:	
basic-thing	
properties:	
(takeable)	Can the thing be taken? (Because this property is missing, GETP will return NIL as its value.)
methods:	
(none)	

LOCATION	Locations in the adventure.
superclass:	
object	
properties:	
description	Long description.
short-description	Short description.
visited	Has the player been here?

key	Key to lock and unlock.
methods:	
(knock? obj)	Can this object enter?
(enter obj)	Cause this object to enter the location.
(open)	Open the portal.
(close)	Close the portal.
(lock key)	Lock the portal.
(unlock key)	Unlock the portal.

child	First object in this location.
(exit directions)	Exits.
methods:	
(knock? obj)	Can this object enter?
(enter obj)	Cause this object to enter the location.
(leave obj dir)	Leave in the specified direction.
(describe)	Describe the location.

Associated Definitions

The following method definition could be associated with a particular location and would require the actor to be carrying the "rusty key" in order to leave the location:

```
(method (leave obj dir)
   (if (send obj carrying? rusty-key)
      (send-super leave obj dir)
      (print "You seem to be missing
              something!\n")))
```

This example also illustrates the use of the `send-super` form. Send-super passes a message to the parent (or super) class of the current object. This definition says that if the actor (the value of `obj`) is carrying the rusty key, the `leave` message should be handled normally. If not, the program prints a message and the action is aborted.

The adventure language statements used to define constants, functions, variables, and property-names are:

```
(define symbol value)
(define (function-name symbol*) expression*)
(variable symbol*)
(property symbol*)
```

Summary

AdvSys is a tool for writing adventure games, much as a word processor is a tool for writing novels. It is not a substitute for good creative writing, but a tool for the writer. I hope the availability of this system will inspire potential adventure authors to write adventure games and share them with the rest of us. (See the article "Interactive Fiction as Literature," which begins on this page.) ■

Editor's note: *The source code for AdvSys, the adventure game writing system, was written in C and includes the following files: ADVCOM, the adventure game compiler; ADVINT, the adventure game interpreter; OBJECTS.ADI, a run-time package containing the basic definitions needed for a game; and the AdvSys documentation.*

The files are available on disk, in print, and on BIX. See the insert card following page 324 for details. Listings are also available on BTYEnet. See page 4. In order to run the programs, you will need a C compiler appropriate for your computer system.

David Betz is a BIX senior editor. He can be reached at BIX, One Phoenix Mill Lane, Peterborough, NH 03458.

MAY 1987 • B Y T E **125–135**