Jerry Cain
CS 106AJ

Handout #44
November 26, 2018

Assignment #6—Adventure

The vitality of thought is in adventure.
— Alfred North Whitehead, *Dialogues,* 1953

Due: Friday, December 7, 5:00 P.M.
Last possible submission date: Sunday, December 9, 5:00 P.M.
Note: This assignment may be done in pairs

This assignment and handout were written by Eric Roberts.

Welcome to the final assignment in CS 106AJ! Your mission in this assignment is to write a simple text-based adventure game in the tradition of Will Crowther's pioneering "Adventure" program of the early 1970s. In games of this sort, the player wanders around from one location to another, picking up objects, and solving simple puzzles. The program you will create for this assignment is less elaborate than Crowther's original game and is therefore limited in terms of the type of puzzles one can construct for it. Even so, you can still write a program that captures much of the spirit and flavor of the original game.

Because this assignment is large and detailed, it takes quite a bit of writing to describe it all. This handout contains everything you need to complete the assignment, along with a considerable number of hints and suggestions. To make it easier to read, the document is divided into the following sections:

Try not to be daunted by the size of this handout. The code is not as large as you might think. If you start early and follow the suggestions in the "Milestones" section, things should work out beautifully.

Section 1
Overview of the Adventure Game

The adventure game you will implement for this assignment—like any of the text-based adventure games that were the dominant genre before the advent of more sophisticated graphical adventures like the Myst/Riven/Exile series—takes place in a virtual world in which you, as the player, move about from one location to another. The locations, which are traditionally called ***rooms*** (even though they may be outside), are described to you through a written textual description that gives you a sense of the geography. You move about in the game by giving commands, most of which are simply an indication of the direction of motion. For example, in the classic adventure game developed by Willie Crowther, you might move about as follows:

Adventure!

```
Welcome to Adventure!
You are standing at the end of a road before a small brick
building.  A small stream flows out of the building and
down a gully to the south.  A road runs up a small hill
to the west.
> WEST
You are at the end of a road at the top of a small hill.
You can see a small building in the valley to the east.
> EAST
Outside building.
>
```

In this example, you started outside the building, followed the road up the hill by typing **`WEST`**, and arrived at a new room on the top of the hill. Having no obvious places to go once you got there, you went back toward the east and ended up outside the building again. As is typical in such games, the complete description of a location appears only the first time you enter it; the second time you come to the building, the program displays a much shorter identifying tag, although you can get the complete description by typing **`LOOK`**, as follows:

Adventure!

```
Outside building
> LOOK
You are standing at the end of a road before a small brick
building.  A small stream flows out of the building and
down a gully to the south.  A road runs up a small hill
to the west.
>
```

From here, you might choose to go inside the building by typing **`IN`**, which brings you to another room, as follows:

Adventure!

```
> IN
You are inside a building, a well house for a large spring.
The exit door is to the south.  There is another room to
the north, but the door is barred by a shimmering curtain.
There is a set of keys here.
>
```

In addition to the new room description, the inside of the building reveals that the adventure game also contains objects: there is a set of keys here. You can pick up the keys by using the **TAKE** command, which requires that you specify what object you're taking, like this:

The keys will, as it turns out, enable you to get through a grating at the bottom of the streambed that opens the door to Colossal Cave and the magic it contains.

In these examples, user input appears in uppercase so that it is easier to see. Your program should recognize commands in lowercase or in any combination of the two.

The code for the teaching machine

The best model for the Adventure assignment is the second of the two teaching machine examples that I presented on the day before the Thanksgiving holiday. The starter project for Assignment 6 includes the code for the revised teaching machine so that you can copy whatever parts of the code you think would be useful.

Like the teaching machine program, the adventure program you create for this assignment is entirely *data driven.* The program itself doesn't know the details of the game geography, the objects that are distributed among the various rooms, or even the words used to move from place to place. All such information is supplied in the form of XML entries in the HTML index file, which the program then uses to control its own operation. If you change the XML description of the game, the same program will guide the player through a different adventure.

As with the Enigma assignment, Adventure is partitioned into several milestones that will allow you to get the program working in stages. It will save you a great deal of time on this assignment if you get each milestone working before you move on to the next.

The starter project includes an **index.html** file that includes much of the geography from Willie Crowther's original game along with a few of the puzzles. The description of the game itself is contained inside the index file in an invisible ** ** tag with the id **"GameData"**. The ** ** tag appears in the **<body>** section of each index file in a structure that looks like this:

```
<div id="GameData" style="display:none;">
   . . . the description of the game is included here . . .
</div>
```

your program can retrieve the parsed XML description of the entire game by calling

```
document.getElementById("GameData")
```

which returns an element from the Document Object Model (DOM) that contains all the information you need.

The various XML tags that can appear inside this particular ** ** element are described later in this handout in conjunction with the milestone in which you implement the relevant tag.

The contents of the starter folder

The starter folder for the Adventure assignment includes the following files:

- **Adventure.js**—This file defines the **Adventure** function itself, which is just a few lines long and looks almost exactly the same as the **TeachingMachine.js** file in the example. We've provided the complete code for **Adventure.js** as shown in Figure 1, and you shouldn't need to change anything in this file at all.
- **AdvGame.js**—This file defines the **AdvGame** class, which implements the game and is therefore analogous to the **TMCourse** class in the teaching machine. The **AdvGame** factory method is responsible for reading the XML data from the index file and storing it in a suitable data structure. This class also exports the **play** method, which is called by the main program to play the game. Although the **play** method is complex, you will have a chance to build it up gradually as you go through the milestones.
- **AdvRoom.js**—This file defines the **AdvRoom** class, which represents a single room in the game and is analogous to the **TMQuestion** class in the teaching machine. The starter file contains the header lines for all the methods exported by **AdvRoom** for any of the milestones, but you don't need to implement these methods until you get to the appropriate point in the assignment.
- **AdvObject.js**—This file defines the **AdvObject** class, which represents an object in the game. As with the **AdvRoom** class, this file specifies the header lines for all the methods that **AdvObject** supports. You will have a chance to implement these methods in Milestone #4.
- **AdvPassage.js**—This file defines the **AdvPassage** class, which represents a passage. This file is provided in its complete form, mostly because doing so gives you an example of a simple class definition. You will use this class starting in Milestone #7.

Figure 1. The `Adventure.js` starter file

```
/*
 * File: Adventure.js
 * ------------------
 * This program plays the CS 106AJ Adventure game.
 */

function Adventure() {
   let game = AdvGame();
   if (game === undefined) {
      console.log("There is no data file for this adventure.");
   } else {
      console.log("Welcome to Adventure!");
      game.play();
   }
}
```

Section 2
Structure of the XML Entries

Although you won't see the implementation of several of these features until you implement the later milestones, it is useful to have a sense of what the internal structure of the relevant XML looks like. The game XML section contains three types of entries:

1. ***Rooms.*** Each room is defined by a **<room>** section, which has the following form:

   ```
   <room name="name used to refer to the room" short="a short description">
      . . . The long description of the room, which may span several lines . . .
      . . . A sequence of passage tags that describe the exits from this room . . .
   </room>
   ```

   The **<passage>** tags that follow the room description look like this:

   ```
   <passage dir="motion verb" room="room name" key="object name"/>
   ```

   The **key** attribute is optional and is used to indicate that the player needs to be holding a specific object to traverse a particular passage. For example, one of the passages for the room above the entrance to the cave is defined as

   ```
   <passage dir="DOWN" room="BeneathGrate" key="KEYS" />
   ```

   which signifies that the player can only go down to the room named **"BeneathGrate"** if the player has the object named **"KEYS"**. Locked passages are discussed in more detail in Milestone #7.

2. ***Objects.*** Each object is defined by an **<object>** section, which looks like this:

   ```
   <object name="name used to refer to the object" location="room name">
      . . . A single line describing the object . . .
   </object>
   ```

   Objects are introduced in Milestone #4.

3. ***Synonyms.*** The adventure game allows the player to enter abbreviations for many of the more common commands. For example, the compass points **N**, **E**, **S**, and **W** are defined to be equivalent to **NORTH**, **EAST**, **SOUTH**, and **WEST**. Similarly, if it makes sense to refer to an object by more than one word, you can use a **<synonym>** tag to define the two as synonyms. As you explore the cave, you will encounter a gold nugget, and it makes sense to allow players to refer to that object using either of the words **GOLD** or **NUGGET**. The **<synonym>** tag has the following form:

   ```
   <synonym word="the synonym" definition="the canonical form of the word" />
   ```

   For example, the **index.html** file includes the following **<synonym>** tags:

   ```
   <synonym word="N" definition="NORTH" />
   <synonym word="GOLD" definition="NUGGET" />
   ```

   Synonyms are introduced in Milestone #6.

Section 3
Milestones

For a project of any reasonable complexity, it is important to implement the project in stages rather than trying to get it going all at once. As with the Enigma assignment, we've given you a set of milestones that will lead you through the process in a series of manageable steps.

Milestone #1: Modify the teaching machine code so that it fits with Adventure

As I showed at the end of Friday's lecture, the **`TeachingMachine.js`** program works as a rudimentary Adventure-style game if you simply change the HTML index file. The result of doing so, however, does not constitute a useful basis for building up a more sophisticated Adventure game. If nothing else, the metaphors used in the code are entirely inappropriate to the new context. The teaching machine program talks about courses, questions, and answers, none of which make sense in the Adventure world. The corresponding concepts in Adventure are games, rooms, and passages. Your first step is to take the code for the teaching machine and adapt it so that it makes sense for the Adventure-game model.

You have two starting points for this phase of the project. The **`TeachingMachine`** folder contains the code for the teaching machine application presented the Friday prior to Thanksgiving break. The **`Adventure`** folder contains the starter versions of the files you need to implement the classes used in the Adventure game. Your task for this milestone is to adapt the code from the **`TMCourse.js`** and **`TMQuestion.js`** files into their **`AdvGame.js`** and **`AdvRoom.js`** counterparts (you don't have to do anything with the other files until later milestones).

The code you need to complete this milestone is entirely there already, at least in a functional sense. All you have to do is copy the code out of the classes for the teaching machine application and add it back to the corresponding classes in the Adventure game, changing the names of fields and methods so that they fit the Adventure game metaphor. The new names of the exported methods are given to you as part of the starter files, but you will also need to change the names of helper functions and local variables so that they make sense in the context of the game.

This milestone has two primary purposes:

1. To ensure that you understand what's going on in the teaching machine application.
2. To give you some practice in debugging. Even though the structure of the code remains exactly the same, this milestone is not as easy as you might think. Nearly all the variable and method names will have to change, and you'll need to be careful to make sure that your changes are consistent. Since you'll probably make some mistakes along the way, you'll need to polish up your debugging skills to figure out exactly what you did wrong.

When you finish this milestone, you should be able to wander a bit around the surface of the Adventure world, heading up to the top of the hill, inside the building, and down to the grate. You won't, unfortunately, be able to get past the grate until Milestone #7.

Milestone #2: Implement short descriptions of the rooms

The Adventure game would be tedious to play—particularly when output devices were as slow as they were in the 1970s—if the program always gave the full description of the room every time you entered it. Crowther's game introduced the idea of short descriptions, which were one-line descriptions for rooms that the player has already visited. The long description appeared the first time a room was entered, and the short description appeared thereafter.

Your job in this milestone is to implement this feature in your program. You will need to implement the **`printShortDescription`** method in the **`AdvRoom`** class, making sure that it correctly implements the functionality described in the comments. It is also important to keep in mind that some rooms do not include the **`short`** attribute, which means that they have no short description. In that case, which you can determine if calling **`getAttribute`** returns **`null`** (not **`undefined`**, as is true for missing properties), calling **`printShortDescription`** should simply print the long description.

You will also need to implement the **`setVisited`** and **`hasBeenVisited`** methods that keep track of whether the room has been visited earlier in the game. You will then need to modify your code for the **`play`** method so that it checks the visited flag before printing the long or the short description, as appropriate.

Once you have completed this milestone, your program should be able to generate the following sample run:

Adventure!

```
Welcome to Adventure!
You are standing at the end of a road before a small brick
building.  A small stream flows out of the building and
down a gully to the south.  A road runs up a small hill
to the west.
> IN
You are inside a building, a well house for a large spring.
The exit door is to the south.  There is another room to
the north, but the door is barred by a shimmering curtain.
> OUT
Outside building.
>
```

Note that the second time the player reaches the starting room, the program displays the short description.

Milestone #3: Implement the `QUIT`, `HELP`, and `LOOK` commands

Most of the commands entered by the player are words like **`WEST`** or **`EAST`** that indicate a passage to another room. Collectively, these words are called ***motion verbs.*** Motion verbs, however, are not the only possible commands. The Adventure game allows the player to enter various built-in commands called ***action verbs.*** The six action verbs you are required to implement (although you only need to implement **`QUIT`**, **`HELP`**, and **`LOOK`** as part of this milestone) are described in Figure 2 at the top of the next page.

The first thing you need to do to implement this milestone is to subdivide the user's input into individual words. Once you've done that, you need to look at the first word to

Figure 2. The built-in action verbs

QUIT	This command signals the end of the game. Your program should stop reading and executing user commands.
HELP	This command should print instructions for the game on the console. You need not duplicate the instructions from the stub implementation exactly, but you should certainly give users an idea of how your game is played. If you make any extensions, you should describe them in the output of your **HELP** command so that we can easily see what exciting things we should look for.
LOOK	This command should type the complete description of the room and its contents, even if the user has already visited the room.
TAKE *obj*	This command requires a direct object and has the effect of taking the object out of the room and adding it to the set of objects the user is carrying. You need to check to make sure that the object is actually in the room before you let the user take it.
DROP *obj*	This command requires a direct object and has the effect of removing the object from the set of objects the user is carrying and adding it back to the list of objects in the room. You need to check to make sure that the user is carrying the object.
INVENTORY	This command should list what objects the user is holding. If the user is holding no objects, your program should say so with a message along the lines of "You are empty-handed."

see if it is one of the action verbs before checking whether a motion verb applies. You then need to implement the first three action verbs. The **QUIT** command stops the program from reading any more user commands, just as a new room with the name **"EXIT"** does in the code you adapted from the teaching machine application. The **HELP** command should print the contents of the **HELP_TEXT** constant on the console using **console.log**. The **LOOK** command calls **printLongDescription** for the current room.

Once you have finished this milestone, your program should be able to produce this sample run:

Adventure!

```
Outside building
> LOOK
You are standing at the end of a road before a small brick
building.  A small stream flows out of the building and
down a gully to the south.  A road runs up a small hill
to the west.
> HELP
Welcome to Adventure!
Somewhere nearby is Colossal Cave, where others have found fortunes in
treasure and gold, though it is rumored that some who enter are never
seen again.  Magic is said to work in the cave.  I will be your eyes
and hands.  Direct me with natural English commands; I don't understand
all of the English language, but I do a pretty good job.

It's important to remember that cave passages turn a lot, and that
leaving a room to the north does not guarantee entering the next from
the south, although it often works out that way.  You'd best make
yourself a map as you go along.

Much of my vocabulary describes places and is used to move you there.
To move, try words like IN, OUT, EAST, WEST, NORTH, SOUTH, UP, or DOWN.
I also know about a number of objects hidden within the cave which you
can TAKE or DROP.  To see what objects you're carrying, say INVENTORY.
To reprint the detailed description of where you are, say LOOK.  If you
want to end your adventure, say QUIT.
> QUIT
```

Milestone #4: Read in the objects and distribute them to their initial locations

The most important extension that separates the Adventure game from the teaching machine application is the introduction of objects like keys and treasures. The objects are specified in the **`index.html`** file using **`<object>`** tags that look like this:

```
<object name="name used to refer to the object" location="room name">
  . . . A single line describing the object . . .
</object>
```

For example, the first object definition in the **`index.html`** file looks like this:

```
<object name="KEYS" location="InsideBuilding">
  a set of keys
</object>
```

This entry shows that there is an object named **`"KEYS"`** that starts off in the room named **`"InsideBuilding"`** whose one-line description is **`"a set of keys"`**.

As a special case, specifying the location as **`"PLAYER"`** is interpreted as indicating that the object starts off in the player's possession. For example, the entry

```
<object name="WATER" location="PLAYER">
  a bottle of water
</object>
```

indicates that the water bottle should be in the player's possession when the game begins.

To implement this milestone, you need to complete the following tasks:

1. Implement the **`AdvObject`** class, which is similar in structure to **`AdvRoom`**. The **`AdvObject`** factory method takes the name, the description between the **`<object>`** and **`</object>`** tags, and location as parameters and returns an object that encapsulates these three data values. The **`AdvObject`** class also exports getter methods for the three internal fields—**`getName`**, **`getDescription`**, and **`getLocation`**—each of which is specified in the starter file. As is often the case with getter methods, each of these methods can be implemented in a single line of code.
2. In the **`AdvRoom`** class, you will need to add a local variable that keeps track of the objects in the room, presumably as an array of **`AdvObject`** values. The **`AdvRoom`** class also exports the following methods for manipulating the list of objects:
   - **`room.describeObjects()`**, which describes the objects in the room.
   - **`room.addObject(obj)`**, which adds the object to the room.
   - **`room.removeObject(obj)`**, which removes the object from the room.
   - **`room.contains(obj)`**, which returns **`true`** if the object is in the room.
3. In the **`AdvGame`** class, you need to make the following changes:
   - Add a call to **`readObjects`** to read in the data for the objects.
   - Add a local variable to keep track of the objects the player is holding.
   - Add a new function to distribute the objects to their appropriate initial locations.
   - Call the **`describeObjects`** method whenever you describe a room.

Given that Milestone #4 does not yet allow you to pick up and drop objects, the only thing you can do to test whether this part of the assignment works is to see whether the objects are listed as part of the room descriptions. For example, you should make sure that the keys are listed inside the building, as shown in the following sample run:

Adventure!

```
> IN
You are inside a building, a well house for a large spring.
The exit door is to the south.  There is another room to
the north, but the door is barred by a shimmering curtain.
There is a set of keys here.
>
```

Milestone #5: Implement the `TAKE`, `DROP`, and `INVENTORY` commands

The next step is to add the **`TAKE`**, **`DROP`**, and **`INVENTORY`** commands to the command processor you implemented for Milestone #3. The **`TAKE`** command looks up the object name, checks to see if that object is in the room, and if so, removes it from the room and adds it to the player's inventory. The **`DROP`** command reverses the process, removing an object from the player's inventory and then adding that object to the room. The **`INVENTORY`** command goes through the player's inventory and prints the description of each object. If the player's inventory is empty, the **`INVENTORY`** command should display the string **`"You are empty-handed"`**. These behaviors are illustrated in the following sample run from the beginning of the game:

Adventure!

```
Welcome to Adventure!
You are standing at the end of a road before a small brick
building.  A small stream flows out of the building and
down a gully to the south.  A road runs up a small hill
to the west.
> INVENTORY
You are carrying:
  a bottle of water
> DROP WATER
Dropped.
> INVENTORY
You are empty-handed.
> IN
You are inside a building, a well house for a large spring.
The exit door is to the south.  There is another room to
the north, but the door is barred by a shimmering curtain.
There is a set of keys here.
> TAKE KEYS
Taken.
> OUT
Outside building
There is a bottle of water here.
> DROP KEYS
Dropped.
> LOOK
You are standing at the end of a road before a small brick
building.  A small stream flows out of the building and
down a gully to the south.  A road runs up a small hill
to the west.
There is a bottle of water here.
There is a set of keys here.
>
```

Milestone #6: Implement synonyms

At this point in your implementation, your debugging sessions will have you wandering through the Adventure game more than you did in the beginning. As a result, you will almost certainly find it convenient to implement the synonym mechanism, so that you can type **N**, **S**, **E**, and **W** instead of the full names for the compass directions. The complete set of **<synonym>** entries in the **index.html** file looks like this:

```
<synonym word="Q" definition="QUIT" />
<synonym word="L" definition="LOOK" />
<synonym word="CATCH" definition="TAKE" />
<synonym word="RELEASE" definition="DROP" />
<synonym word="I" definition="INVENTORY" />
<synonym word="N" definition="NORTH" />
<synonym word="S" definition="SOUTH" />
<synonym word="E" definition="EAST" />
<synonym word="W" definition="WEST" />
<synonym word="U" definition="UP" />
<synonym word="D" definition="DOWN" />
<synonym word="BACK" definition="OUT" />
<synonym word="GOLD" definition="NUGGET" />
<synonym word="BAG" definition="COINS" />
<synonym word="NEST" definition="EGGS" />
<synonym word="BOTTLE" definition="WATER" />
```

To implement the synonym processing, you need to read through the **<synonym>** tags in the **index.html** file and use them to create an object that maps alternative word forms into their standard definition. The easiest place to implement this feature is in the **AdvGame** class. Whenever you read a word—which might be a motion verb, an action verb, or the name of an object—you need to see if that word exists in the synonym table and, if so, substitute the standard definition.

Milestone #7: Implement locked passages

When you modified the teaching machine code for Milestone #1, you presumably defined the structure representing passages to be a map from direction names to room names. That is, after all, how the teaching machine worked, and we didn't suggest you change it.

Unfortunately, using a map doesn't quite work for the Adventure game. If you look closely at the list of passages for certain rooms, you will discover that the same direction name can occur more than once in the list. For example, the **<room>** entry for the room above the grate that leads to the underground part of the cave has the following contents:

```
<room name="OutsideGrate" short="Outside grate">
  You are in a 20-foot depression floored with bare dirt.
  Set into the dirt is a strong steel grate mounted in
  concrete.  A dry streambed leads into the depression from
  the north.
  <passage dir="NORTH" room="SlitInRock" />
  <passage dir="UP" room="SlitInRock" />
  <passage dir="DOWN" room="BeneathGrate" key="KEYS" />
  <passage dir="DOWN" room="MissingKeys" />
</room>
```

As you can see, the motion verb **"DOWN"** appears twice in the list of passages. The first one is associated with a **key** field that has the value **"KEYS"** and takes the player to a room named **"BeneathGrate"**. The second has no **key** field and sends the player off to a room named **"MissingKeys"** that you'll have a chance to see in Milestone #8. This definition is an example of a ***locked passage,*** which is one that requires the player to be holding a specified object that unlocks the passage. In this case, the key is literally the set of keys that starts off inside the building. If the keys are in the player's inventory, applying the motion verb **DOWN** uses the first passage; if not, applying **DOWN** skips over that passage and follows the one to the room named **"MissingKeys"**.

This new interpretation requires you to change the implementation of the data structure used to represent passages, since a map doesn't allow multiple values with the same key. What you need to do is change the data structure used to represent the passages from a map to an **array** in which the individual elements are **AdvPassage** objects containing a direction, the name of the destination room, and an optional key value. The code that moves from one room to another based on the player's input must search through the array to find the first option that applies.

The first step in adopting this new design is to take a look at the **AdvPassage** class, which you've been able to ignore up to now. This class, which is fully implemented in the starter folder, is an extremely simple one that looks like any of the simple class examples from the reader. The factory method takes the three components and keeps track of them in the closure. The getter methods simply return these values.

The changes you need to make for Milestone #7 are in the **AdvRoom** and **AdvGame** classes. You will, for example, have to change your implementation of **readRoom** so that it stores the data for the passages in an array rather than a map. You also need to change the way that the **getNextRoom** method works, since this method now has to take account of what the player is carrying. That requirement, however, creates a bit of a problem. The **AdvRoom** class doesn't know what the player is carrying, since that information is stored available only inside the **AdvGame** class.

There are several strategies you might use to solve this problem. One possibility is to pass the player's inventory—along with the map from object names to **AdvObject** structures—as additional arguments to the **getNextRoom** method. On the whole, however, it is probably simpler to reassign the task of figuring out the which room you reach after moving in a particular direction into the **AdvGame** class, which already has the necessary information. Doing so requires making the structure containing the array of passages available to clients of **AdvRoom**, which is easily done by adding a **getPassages** method to the class.

Once you have implemented this change, you should be able to explore the entire Adventure game, picking up objects and using them as keys to get through previously closed passages. You still, however, would be prevented from escaping through a passage unless you're holding the necessary key. For that, you need to implement the final milestone.

Milestone #8: Implement forced motion

When the player tries to go through a locked passage without the necessary key, the game has to indicate that the motion is prohibited. One possible strategy would be to design a whole new data structure to represent messages of this type. A simpler way, however, is to make a small extension to the structure that is already in place.

When Willie Crowther faced this problem in his original Adventure game, he chose the simple approach. He simply created new rooms whose descriptions contained the messages he wanted to deliver. When the player entered one of those rooms, the code that you've been running all along would print out the necessary message, just like any other room description. The only problem is that you don't actually want the player to end up in that room, but rather to be moved automatically to some other room. To implement this idea, Crowther came up with the idea of using a special motion verb called **`"FORCED"`** to specify ***forced motion.***

Whenever the player ever enters a room in which one of the connections is associated with the motion verb **`FORCED`** (and the player is carrying any object that the **`FORCED`** verb requires to unlock the passage), your program should display the long description of that room and then immediately move the player to the specified destination without waiting for the player to enter a command. This feature makes it possible to display a message to the player and continue on from there.

This facility is illustrated by the room named **`"MissingKeys"`**, which has the following definition:

```
<room name="MissingKeys">
  The grate is locked and you don't have any keys.
  <passage dir="FORCED" room="OutsideGrate" />
</room>
```

The effect of this definition is to ensure that whenever the player enters this room, the room will automatically be set to **`"OutsideGrate"`**.

It is possible for a single room to use both the locked passage and forced motion options. The **`CrowtherRooms.txt`** file, for example, contains the following entry for the room just north of the curtain in the building:

```
<room name="Curtain1">
  <passage dir="FORCED" room="Curtain2" key="NUGGET" />
  <passage dir="FORCED" room="MissingTreasures" />
</room>
```

The effect of this set of motion rules is to force the player to the room named **`Curtain2`** if the player is carrying the nugget and to the room named **`MissingTreasures`** otherwise. When you are testing your code for locked and forced passages, you might want to pay particular attention to the last eight rooms in the **`index.html`** file. These rooms implement the shimmering curtain that marks the end of the game.

Section 4
Administrative Rules

Project teams

As on the Breakout and Enigma assignments, you are encouraged to work on this assignment in teams of two, although you are free to work individually as well. Each person in a two-person team will receive the same grade, although individual late-day penalties will be assessed as outlined below.

Grading

Given the timing of the quarter, your assignment will be evaluated by your section leader without an interactive grading session.

Due dates and late days

As noted on the first page of this handout, the final version of the assignment is due on Friday, December 7th. You may use late days on this assignment, except that the days are now *calendar* days rather than *class* days (which makes sense given that class isn't meeting). If you submit the assignment by 5:00P.M. on Friday the 8th, you use up one day late, and so forth. All Adventure assignments, however, must be turned in by 5:00P.M. on Sunday, December 9th, so that your section leaders will be able to grade it (and so you can study for the December 10th final exam).

On the Adventure assignment, late-day accounts are calculated on an individual basis. Thus, if you have a free late day but your partner does not, you would not be penalized if the assignment came in on Saturday, but your partner would.