Bonus Chapter 15

"I'll be back…"
-- You know who!

The *tEXT* Generation

By Andre' LaMothe

In this bonus chapter we will learn about the oldest games around known as "text adventures" or text based games. A long time ago (the 70's and early 80's) computers didn't have the graphics capabilities that they have today. Because of this, most games were descriptive in nature rather than graphic. The games would use text to convey the state of the game and conversely the player would enter in plain English sentences to command the character in the game. One of the most notable games created in the early 80's was a game called Zork by Infocom. It was incredibly successful because the language interpreter was very advanced and the game environment was very robust. Furthermore, the user could type in almost any sentence and the game would be able to figure out what the player was trying to say.

The material we will cover isn't hard, but it is different from what we have been covering. There will be a lot of new terms and many of them have meanings that aren't well defined. But, by the end of this chapter, you will be able to make your own text adventure! Here are the topics we'll cover:

- What is a text game
- How do text games work
- Getting input from the outside world
- Language analysis and parsing
- Lexical analysis
- Syntactical analysis
- Semantic analysis
- Putting all the pieces together
- Representing the universe
- Placing objects in the world
- Making things happen
- Moving around
- The inventory system
- Implementing sight, sound and smell
- Making it Real-Time
- Error handling
- Creeping around with Shadow Land

- The language of Shadow Land
- Building and playing Shadow Land
- Shadow's game loop
- Winning the game

What is a text game?

A text game is a video game without the video! Well at least without all the cool graphics. Text games are like interactive books that are written as you play. The user gets to use his/her own imagination to make up what he/she thinks the universe look likes that is being played in. You may have never seen a text game because you are a product of the "gui age". The interface to a text game is a simple as this:

What do you want to do? Eat the apple

Yum, that tasted good!

What do you want to do?

.

.

What you see here is a short dialog with the computer and player. In most text games, the computer prompts the player to tell it to do something. Then the computer will break the sentence down in to it components and see if the action is legal and proceed with it if possible.

The thing to remember is that there are no images or sounds. The only image is in the player's mind. And the only thing that creates this image is the descriptions given by the game. Therefore, the English language used to describe the game universe should be as "fluffy" and poetic as possible. As an example, say that you were designing a text game that had a bathroom in it. When the player ask to "see" what it looks like one possible description might be:

…you see a white bathroom with towels hanging on the racks…

This is fine, but it's boring. A better response would be:

…you are stunned by the size of the bathing room that surrounds you. To your West, you see a large shower enclosed in rose tinted glass. The entrance is paved with small polished stones. To the East, you see a large wash area with marble basins and silver faucets. From above you are bathed in sunlight from the three overhead skylights. Finally, at your feet is a pattern of black and white tiles placed with surgical precision….

As you can see the second version is much better. It creates an image in your mind and this is the key to text adventures. Even though the interface to a text game is usually nothing more than text, the game that is being played is limited only by the imagination of the designers and that of the player. Usually a text game will contain hundreds of pages of descriptive text within the universe database. This text is used as the foundation for the universe when the player asks something about it.

The technology needed for text games is based on compiler techniques coupled with very elegant data structures and algorithms. Remember, computers don't understand English and making them understand what nouns, verbs, adjectives, prepositional phrases, direct objects and so forth are is a great task to say the least(I wish I would have paid more attention when my teachers were diagramming sentences). Many people may think that creating a text based adventure is easy, but they are very wrong. Personally, I think that people who write text adventures probably know more about Computer Science, data structures, and Mathematics than the people who write arcade games.

Text game gurus definitely know a lot about compilers and interpreters -- which everyone knows is a very difficult subject. Even today text games are still going strong; however, they have been augmented with incredible graphics. The genre of RPG (Role Playing Games) is really the evolved state of text games. Many RPG's have text interfaces so the player can dialog and ask questions to the characters in the game. Today we aren't going to take a course in compiler design (which you should do to be a complete person), but we will learn some of the basic concepts and techniques used to create a text based adventure. You will also see a complete game at the end of the day called Shadow Land.

How do text games work?

That's a good question. And there are many answers, all of which are correct, but each of these possible answers will have some factors in common. First, the interface to the game will be a bi-directional text only communication channel. This means that the computer will only have text to say what it wants to say and similarly the player can only dictate his input with text likewise. Also, there are no joysticks, mice, flight sticks, or light pens. Secondly, the game will consist of some kind of "universe" whether it be the old west, an apartment building or a space station. But this universe will consist of geometry, descriptions and rules. The geometry is the actual geometry of the universe, the size and placement of the rooms, halls ways, ponds or whatever. The descriptions are the actual text that can be called on to describe what a location looks like, sounds like or smells like.

The rules of the game are the things that can and can't be done. For example, you may not be able to eat a rock, but you can eat a sandwich. Once the geometry,

descriptions and rules of the game are in place, then the data structures, algorithms and software to allow the player to interact with the environment need to be created. This will consist mainly of an input parser. The input parser is responsible for translating and making sense of the player's input. The player will communicate to the game using standard English words. These words may create complete sentences, small phrases or even single commands. The parser and all of it's components will break the sentence down into separate words, analyze the meaning of the sentence and then execute the appropriate functions to make whatever happen the player was asking for.

There is one catch here. The computer doesn't understand natural language and giving it a complete understanding of the English language is a Ph.D. thesis at the very least. As game programmers, we just want to give the player a subset of the English language and impose a few rules about how sentences are constructed. For example, the game you will see later today is called ***Shadow Land*** and has a very limited vocabulary. It can only understand the following words:

Table 15.1 - The vocabulary of Shadow Land

Word	**Used as**
LAMP	Noun
SANDWICH	Noun
KEYS	Noun
EAST	Noun
WEST	Noun
NORTH	Noun/Adjective
SOUTH	Noun/Adjective
FORWARD	Noun/Adjective
BACKWARD	Noun/Adjective
RIGHT	Noun/Adjective
LEFT	Noun/Adjective
MOVE	Verb
TURN	Verb
SMELL	Verb
LOOK	Verb
LISTEN	Verb
PUT	Verb
GET	Verb
EAT	Verb
INVENTORY	Verb
WHERE	Verb
EXIT	Verb
THE	Article
IN	Preposition
ON	Preposition

TO	Preposition
DOWN	Preposition

The vocabulary of Shadow Land is very small, but you would be surprised at how many legal sentences that can be constructed. The problem is making sense of these constructions using some general algorithm. This process is called ***syntactical analysis*** and is very tedious and complex. Entire books have been written on how to perform syntactical analysis such as the infamous "Dragon Book" by Aho, Sethi and Ullman. We don't need (want) to make things too complex so we will force rules on our vocabulary that creates a little "language". This language will be the one that the user must abide by when constructing sentences.

Once the user has typed something in and the game figures out what he/she is trying to say then the game will proceed to bring the request to fruition and the game will output the results. For example, if the player asks to "look" then the game would access the universe database along with the current position of the player. Together this data can be used to print out the general description of the room. Which may be static. If there are movable objects within the game then there is as a second phase of description. The game logic would test to see of there are any objects within the field of view of the virtual character and then print them out. However, when the final description prints out it must seem fluid and not choppy. For example, the game software should first test if there are any objects in the room and if so make note of it and slightly alter the last sentence of the static description to have a conjunction like "and" so that the objects when listed don't appear from nowhere.

As an example, a room may have this static description:

…You are surrounded by tall walls with Roman art hanging upon them.

And let's say that there are moveable objects in the room, such as a plant. Then the computer's response might be:

…You are surrounded by tall walls with Roman art hanging upon them.

There is a plant in the East corner of the room.

A better algorithm for printing out descriptions might take into consideration that there is a moveable object in the room and then print the description out like this:

…**You are surrounded by tall walls with Roman art hanging upon them and there is a plant in the East corner of the room.**

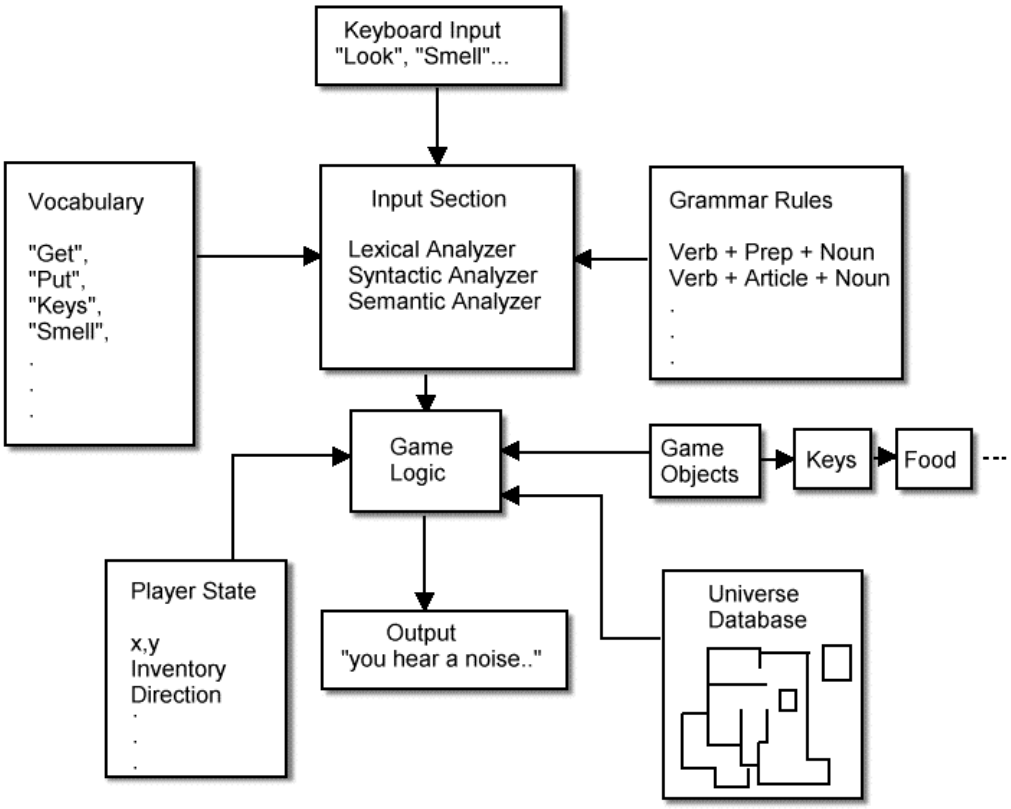
Although, the sentence is slightly artificial, rough, and the computer wouldn't know the difference between a plant and a ogre, at least the sentences are somewhat connected together. This is one of the "tricks" in making good text games. You must "work" the

output sentences to make them read as if they aren't being printed out from a static database.

Of course, there is more than parsing the text and trying to satisfy the request. The game must have some kind of data structure to represent the universe and the objects within the universe. This representation whatever it may be must also have some kind of valid geometrical coherence because as the player moves around, he/she will expect to find a key where they dropped it. This means that many times the universe will have to be modeled as a 2D/3D vector map or cell map so that the player can be moved around in a data representation that has some geometrical relation to the virtual space the player thinks he/she's in. Next the game has to have some kind of structure that contains the "state" of the player. This could mean his/her health, position, inventory, and position in the game universe.

Finally, other aspects of the game have to be implemented such as a goal and the enemies (if there are any). The goal may be as simple as finding an enchanted chalice and putting it somewhere. Or the goal may be as complex as solving some kind of puzzle with a questions and answers dialog between the player and some creature in the game. The creatures in the game will be implemented as data structures only; however, these data structures can move around the universe data structure, move objects, eat food and attack the player (possibly). All of these aspects of the text game must be implemented in a way so that the illusion of a real environment is upheld. That means that you should assume that the player can't see the game, but everything better make sense or else!

Figure 15.1 - The components of a text game.



Let's re-iterate all the components of a text game by studying Figure 15.1. Referring to the figure, we see that there is an input section that parses the players commands and then tries to execute these commands. The parser only understands a specific vocabulary and furthermore the sentences creates with this vocabulary are limited by the "language" designed by the game designer. Next there is a set of data structures holding the representation of the universe, the position of the objects, the description strings for sights, sounds and smells. Also there is the representation of the player and his/her inventory along with the representation of the enemies in the game. Finally, there is a million or so functions, rules, and little details that make it all work, let's cover a few of them!

Getting input from the outside world

Since a text based games uses the keyboard as it's sole input device, we should take some time to make this as easy as possible. The player will be typing in sentences to command the text game to do something. This seems simple enough, but the problem is how to read these sentences. The standard function *scanf()* won't work correctly because of the possibility of white-space and multiple arguments. We need a function that will get a line of input regardless of the characters that make up the line.

This input will stop if and only if the user hits the return key. At this point the sentence would be passed to the next piece of software in the parser chain. So the question is how to get a single line of text without the input line editor making decisions for us? The answer is to use single character input and build up a string until the carriage return is pressed. This can be done using the *getch()* function with a tests for backspace and carriage return. Here is a typical line input function.

Listing 15.1 - A single line input function with editing capability.

```
char *Get_Line(char *buffer)
{
// this function gets a single line of input and tolerates white space

int c,index=0;

// loop while user hasn't hit return
while((c=getch())!=13)
     {
// implement backspace
     if (c==8 && index>0)
        {

        buffer[--index] = ' ';
        printf("%c %c",8,8);

        } // end if backspace
     else
     if (c>=32 && c<=122)
        {
        buffer[index++] = c;
        printf("%c",c);

        } // end if in printable range

     } // end while

// terminate string
buffer[index] = 0;

// return pointer to buffer or NULL
if (strlen(buffer)==0)
   return(NULL);
else
return(buffer);

} // end Get_Line
```

The function takes as a parameter a pointer to a buffer where the inputted string will be placed after the function executes. During execution *Get_Line()* will allow the user to input a line of text and also to edit errors via the backspace key. As the characters are input they are echoed out to the screen, so the user can see what he/she is typing. When the user hits the return key the string is terminated with a *NULL* and function ends.

Once the user has input the string then the game is ready to parse it. The parsing process has many phases (which we will cover shortly), but before we can parse the sentence and see what it means, we must know what the language is and how it is constructed.

Language analysis and parsing

Before the user can type anything in we must define a "language" that he/she must stay within. This language consists of vocabulary and a set of rules (grammar). Together, these conventions create the overall language. Now, the whole idea of text game is to use the English language. This means that the vocabulary should be English words and the grammar should be legal English grammar. The first request is easy. The second request is much harder. Contrary to the belief of many of my English teachers, the English language is a terribly complex language system. It's full of contradictions, points of view, different ways of doing things and so on. In general, the English language is not robust and exact like computer languages. This means that as text game programmers we may have to only allow the user a very small subset of the possible grammatical constructions that could be made with a given vocabulary.

This may seem like a problem, but it's not. A player of a text game gets very comfortable very quickly with a more robust well defined subset of English and finds that the sentences are concise and to the point. With all that in mind the first thing that any game needs is a vocabulary. This is constructed in an on going manner as the game is written. The reason for this is that as the game is being written the designer may feel that another game object needs to be added such as a "zot". That would mean that "zot" would have to be added to the vocabulary if the player were ever to be able to refer to it. Secondly, the designer might decide that more prepositions are needed to make some sentences more natural. For example, to drop an object the player might type:

"**drop the keys**"

The meaning of this is very clear. The word "drop" is a verb, "the" is a worthless article and "keys" is a noun. However, it might be more natural for the player to type:

"drop down the keys"
or
"drop the keys down"

Regardless whether these two sentences are as "good", the fact of the matter is this is how people talk and that is all that is important. Therefore, the vocabulary will need words for all the objects in the game (nouns) along with a few good ***prepositions***. As a rule of thumb start with the prepositions of everything you can do to a mountain and that will usually suffice. Here is a list of common prepositions used in text games.

Table 15.2 - Some good prepositions

from
on
in
down
up
behind
into
before
at

The next class of words needed in our vocabulary are the ***action verbs***. These are the words that mean "do something". They are usually followed by a word or phrase that describes what the action is taking place on. For example, in a text game the action verb "move" is very useful in navigating the player about the universe. A set of possible sentences using "move" as an action verb might be.

1. move north
2. move to the north
3. move northward

Sentence number one is simply enough. It says to move to the north. The second sentence also says to move to the north, but it has a prepositional phrase "to the north". Finally, sentence number three uses the adjective "northward" to mean the same thing. Technically, it could be argued that each of the sentences could mean different things, but to us as game programmers they all mean the same thing. And that is to take a step in the direction of north. Therefore, we see that the use of articles such as "the" and prepositions are absolutely needed to make the sentences have a little variety even if they all mean the same thing.

Back to the subject of action verbs. The game should have a large list of action verbs (many of which can be used by themselves with implied objects). For example, say that the vocabulary for your game had the action verb "smell" in it. You might say:

"smell the sandwich"

Which is clear. However, the command:

smell

Isn't so clear. Yes, it's clear what the player wants to do, but what he/she wants to smell is ambiguous. This is where the concept of context comes into play. Since there is no direct or indirect object given to "smell", the game will assume the player means the exterior environment or the last object acted upon. For example, if the player just

picked up a rock. And then requested the game engine to "smell". Then the game might reply "the rock?" and then the player would say "yes" and the proper description string would be printed. On the other hand, if the player hadn't recently picked anything up then the single command "smell" might illicit a general description of the smell within the room that the player is standing in.

After you have decided on all the action verbs, nouns, prepositions, and articles that can be used in the language then the rules for the language itself must be generated. These rules describe the possible sentence constructions that are legal and this information is used by the syntactical analyzer along with the ***semantic analyzer*** to compute the meaning of the sentence along with its validity. As an example, let's generate a vocabulary along with the rules (grammar) that governs it.

NOTE: By the way, if I say something is a verb, noun, adjective, preposition or article then just accept it because I am going to use the word in that manner even if it isn't strictly correct to think of it as such. Remember the language we make up for games is NOT English. It is only based on English and a verb in one language may be and adjective in the other!

Table 15.3 - A sample vocabulary

Word	Used as	Type name
rock	Noun	OBJECT
food	Noun	OBJECT
table	Noun	OBJECT
key	Noun	OBJECT
get	verb	VERB
put	verb	VERB
the	article	ARTICLE
on	preposition	PREP
onto	preposition	PREP

Where the "type name" is used to group similar word types, so they can be worked with more efficiently.

Now that we have the vocabulary let's construct the language with it. This means creating the rules for legal sentences. Not all of these sentences may make sense, but they will all be legal. These rules are usually referred to as "the syntax of the language" or the ***productions***. I prefer to call them ***productions***.

NOTE: I will refer to the language we are constructing as "Glish".

Table 15.4 - The productions of "Glish"

OBJECT- > "rock" | "food" | "table" | "key"

VERB- > "get" | "put"

ARTICLE- > "the" | NULL

PREP- > "on" | "onto" | NULL

SENTENCE- > VERB + ARTICLE + OBJECT + PREP + ARTICLE + OBJECT

Where the " | " means logical OR, " + " means concatenate and NULL means NULL string.

Now if you try to build sentences using the production for "SENTENCE" then there are quite a few sentences that can be constructed. However, some of them may not make any semantic sense. For example, the sentence:

"**put the rock onto the table**"

Makes perfect sense, but the sentence:

"**get the rock onto the table**"

Is unclear, it could mean place the rock on the table? This is where one of two things must be done. Either more productions must be incorporated into the language to separate specific verbs and their constructions or the game code must test to see if sentences make "sense". To make sure you see how the productions are used let's do a few examples of legal sentences.

Table 15.5 - Some sample sentences and their clarity of meaning ranked 1-10 (with 10 meaning; very clear).

Sentence	Clarity	Legality
"put rock"	4	yes
"put the rock"	4	yes
"get the key"	8	yes

"put down the food"	10	no (There is no production for placing a preposition after the verb and "down" isn't in the vocabulary)
"put the food on rock"	7	yes
"put the rock down onto table"	7	no(there are two prepositions concatenated after each other following the object, there is no production for this form)

As you can see from the table, if a production doesn't exist for a particular sentence form then the sentence is illegal regardless if it makes sense. This is because the computer will not know that it is a legal sentence unless there is an implementation of the production or syntactic rule that governs the construction of the desired sentence. To put it another way, you must program in every single sentence type as a logical construction of the elements in the vocabulary and you must be able to test each sentence to see if it follows the production rules. Of course I was joking about the "clearly" part!

Once the sentence has been broken down and the meaning is starting to become clear then comes the semantic checking. This phase tests if a valid sentence makes sense. For example, the sentence:

"get the key onto the table"

Is a legal construction, but it is unclear what the user wants and should probably be flagged as unclear and the user should be requested to say it in another way, possible:

"put the key on table"

The question that should be burning in your mind is how do I make the program do what the sentence says to do after I have figured out what the user wanted to say? The answer is with ***action functions***. Action functions are called based on the action verb of the sentence. These action functions can act as syntax checkers, semantic checkers or a combination of both or neither (it's up to you). But, one thing the action function should do, is make something happen. This is accomplished by having a separate function for each action function. The action functions themselves are responsible for figuring out what object(s) the action verb is supposed to be applied to then perform the

requested action. For example, the action function for the verb "get" might step through the following set of operations.

- Extract the "object"(noun) from the sentence.

- Using "vision" to see if the object that is being requested to be picked up is within the reach of the player

- If the object is within reach then it is retrieved. At this point, the universe database along with the player's inventory are updated to reflect this action.

The first step (extracting the object) is accomplished by "consuming" words that don't change the meaning of the sentence. For instance the prepositions and articles in our language Glish don't change the meaning much of any particular sentence. Take a look below:

"**put the key**"

The article "the" can be taken out of the sentence without changing the meaning. This consumption of irrelevant words is transparent to the player, but allows him/her to write sentences that make more sense from their point of view. Even though our language parser might like:

"**put key on table**"

A human player may feel more comfortable with:

"**put the key onto the table**"

In any case, the action function will take care of this logic. Once the action function has figured out what needs to be done then doing it is easy. In the case of the original example of getting something; the object would be taken out of the universe database and inserted into the player's inventory. Where the player's inventory might just be an array of characters or structures that list the objects the player is holding. For example, in the game Shadow Land, the player's inventory and everything about him/her is contained within the single structure shown below:

Listing 15.2 - The structure that holds the player in Shadow Land.

```
// this structure holds everything pertaining to the player
typedef struct player_typ
        {
        char name[16];   // name of player
        int x,y;         // position of player
        int direction;   // direction of player, east,west north,south
```

```
    char inventory[8]; // objects player is holding (like pockets)
    int num_objects;   // number of objects player is holding

} player, *player_ptr;
```

The inventory in this case is an array of characters that represent the objects the player is holding. For example, 's' stands for sandwich. This data structure is just an example, you may want to do it differently than this.

> TIP: Text games are usually not played in real time, hence they can be slow internally since the game will have years relatively speaking to think as the player types in a single sentence!

In essence all a text game has to do is break a sentence down, figure out the action requested, call the appropriate action function(s), and perform the action. Performing the action is accomplished by updating data structures, changing coordinates, and printing out text. The majority of the problem in a text game is the translation of the input command string into something the computer can deal with, such as numbers (as you have probably surmised). This process is called ***tokenization*** or ***lexical analysis*** and is the next topic of discussion.

Lexical analysis

When the player types in a sentence using the vocabulary we supply him/her with then the sentence must be converted from strings into tokens (integers), so that syntactic and semantic phases of analysis can proceed. The reason lexical analysis is necessary is due to the fact that working with strings is much more difficult, time consuming and memory intensive than working with tokens (which are usually integers). Hence, one of the functions of the lexical analyzer is to convert the input sentence from string form to token form. There are three parts to this translation. The first part of lexical analysis is simply separating the "words" in a sentence and extracting them. By words, it is meant strings of characters separated by "white space". White space is usually considered to be the space character (ASCII 32) and the horizontal tab (ASCII 9). For example, the sentence:

**"This	is	a test."**

Has four words (tokens), they are:

1. "This"
2. "is"
3. "a"
4. "test"

We also see that periods will have to be taken into consideration since they separate sentences. If the user is only going to be inputting a single line at a time, he/she may or may not put a period at the end of each input sentence. Alas, the period can be thought of as white space. On the other hand, if it is legal for the player to type multiple commands separated by a period or other phrase separator like the colon ":" or semi-color ";" then extra logic may be needed since the sentences should be parsed separately. For example, if the period is assumed to be white space in a single sentence construction as in:

"This is a test."

Then the meaning to the parser is the same as:

"This is a test"

However, when two sentences are placed next to each other with the period being interpreted as white space then the following problem can occur. This sentence:

"This is a test. Get the book."

Will be interpreted as:

"This is a test get the book"

Which has no meaning.

The moral of the story is be careful what you elect to call "white space" and what you don't. Moving on, we need to figure out a way to extract the "words" in a sentence that are separated by white space. The C library actually has a string function to do this called *strtok(),* but it doesn't work properly on some compilers and we will summarily dismiss it since we like to re-invent the wheel.

TIP: Game programmers like to re-write everything. Even the operating system!

What we need is function that will separate the words out for us. This is actually a fairly easy function to write as long as you take your time and take all the cases into consideration that can occur such as *NULL* terminators, white space and the return character. Anyway, here is one such implementation of a token extraction function.

Listing 15.3 - A function to extract tokens from an input sentence (excerpted from Shadow Land)

```
int Get_Token(char *input,char *output,int *current_pos)
{

int index,   // loop index and working index
    start,   // points to start of token
    end;     // points to end of token

// set current positions
index=start=end=*current_pos;

// eat white space
while(isspace(input[index]) || ispunct(input[index]))
     {
     index++;
     } // end while

// test if end of string found
if (input[index]==NULL)
   {
   // emit nothing

   strcpy(output,"");
   return(0);

   } // end if no more tokens

// at this point, we must have a token of some kind, so find the end of
it
start = index; // mark front of it
end   = index;

// find end of Token
while(!isspace(input[end]) && !ispunct(input[end]) && input[end]!=NULL)
     {
     end++;
     } // end while

// build up output string
for (index=start; index<end; index++)
    {
    output[index-start] = toupper(input[index]);
    } // end copy string

// place terminator
output[index-start] = 0;

// update current string position
*current_pos  = end;

return(end);
} // end Get_Token
```

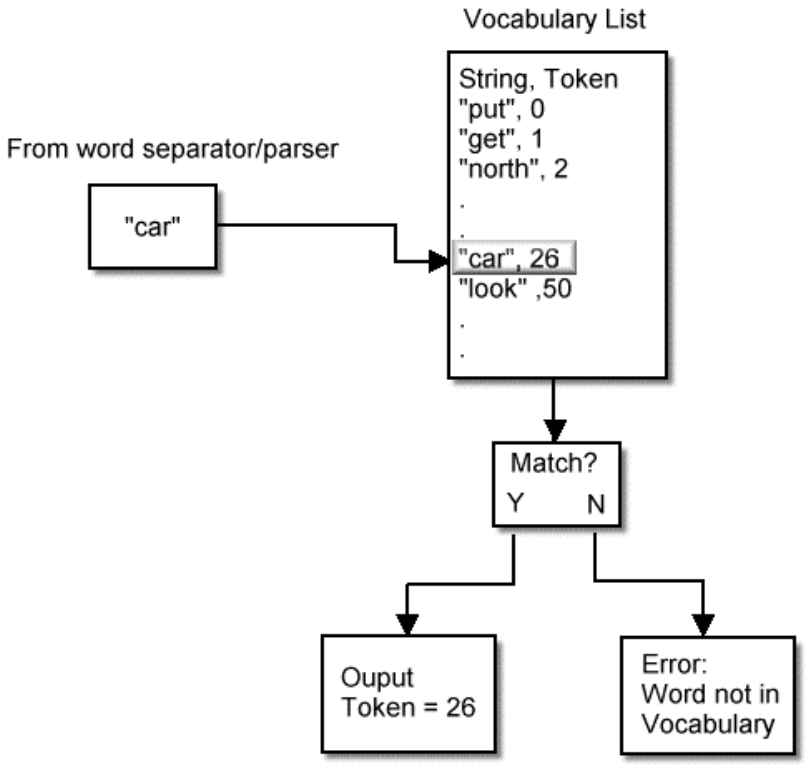
The function takes three inputs. An input string to be parsed, an output string that holds the token extracted and the current position in the string that is being processed. For example, let's see how a string can be parsed using the function. First we need to declare a couple of variables:

```
int position=0;  // used as index from call to call
                 // to keep track of current string
                 // position

char output[16]; // output string

// begin program
Get_Token("This is a test",output,&position);
```

After this call the variable *output* would have a "This" in it and *position* would be equal to 4. Let's do one more call…

```
Get_Token("This is a test",output,&position);
```

Figure 15.2 - A word being tested to see if it's in the vocabulary of the game.



Now, the variable *output* would have "is" in it and position would now be updated to 7. If *Get_Token()* is repeatedly called in this fashion then each "word" in the sentence will be extracted and be placed in the buffer *output* (of course each previous string in *output* will be overwritten by the next token word). Consequently we now have a method of obtaining the "words" that make up a sentence. The next task is to convert these strings into integer tokens so that they can be worked with more easily. This is done with a function that has a table of strings along with the token for each of the

vocabulary words. A search through the table is done for each word and when and if the string is found in the table then it is converted to an integer token, see Figure 15.2.

During this phase of lexical analysis is where the vocabulary checking is done. If a word is not in the vocabulary then it is not in the language and hence is illegal. As an example, the game Shadow Land has the following data structure to hold each word in the vocabulary:

Listing 15.4 - The data structure for a token in Shadow Land

```
// this is the structure for a single token
typedef struct token_typ
        {
        char symbol[16];   // the string that represents the token
        int value;         // the integer value of the token
        } token, *token_ptr;
```

This structure has a place for both the string and a value to be associated with it. The string is the actual vocabulary word while the value is arbitrary. However, the values for tokens should be mutually exclusive unless you wish to make words synonyms. Using the above structure and some defines, you can create a vocabulary table in very compact form that can be used as a reference in the lexical analysis to convert words to tokens and to check their validity. Here is the vocabulary table used in Shadow Land.

Listing 15.5 - The static initialization of the vocabulary table for Shadow Land.

```
// this is the entire "language" of the language in Shadow Land.
token language[MAX_TOKENS] = {

 {"LAMP",       OBJECT_LAMP     },
 {"SANDWICH",  OBJECT_SANDWICH },
 {"KEYS",       OBJECT_KEYS     },
 {"EAST",       DIR_1_EAST      },
 {"WEST",       DIR_1_WEST      },
 {"NORTH",      DIR_1_NORTH     },
 {"SOUTH",      DIR_1_SOUTH     },
 {"FORWARD",    DIR_2_FORWARD   },
 {"BACKWARD",  DIR_2_BACKWARD  },
 {"RIGHT",     DIR_2_RIGHT     },
 {"LEFT",       DIR_2_LEFT      },
 {"MOVE",     ACTION_MOVE     },
 {"TURN",     ACTION_TURN     },
 {"SMELL",   ACTION_SMELL    },
 {"LOOK",     ACTION_LOOK     },
 {"LISTEN",   ACTION_LISTEN   },
 {"PUT",        ACTION_PUT      },
 {"GET",        ACTION_GET      },
 {"EAT",        ACTION_EAT      },
 {"INVENTORY", ACTION_INVENTORY},
 {"WHERE",     ACTION_WHERE    },
 {"EXIT",         ACTION_EXIT     },
 {"THE",          ART_THE         },
 {"IN",             PREP_IN         },
```

```
 {"ON",           PREP_ON         },
 {"TO",         PREP_TO         },
 {"DOWN",       PREP_DOWN       },
};
```

As you see there are character strings followed by defined symbols (the value of which are irrelevant as long as they are different). You will notice the defined symbols have familiar prefixes such as PREP (preposition), ART (article) and ACTION (verb).

> NOTE: I have been using example code excerpts from Shadow Land. If you haven't seen the game or the code for it don't worry. We are only trying to grasp concepts at this point and using specific implementation examples will not degrade the lesson since the techniques are so standard.

Once a table like this exists in some such form then the token strings can be compared to the elements in the table and the character strings representing the tokens can be converted to integers. At that point, the input sentence will be a string of numbers, which can be processed in a more convenient fashion. A sample ***tokenizer*** is listed below. It is also from the game Shadow Land, but if you have seen one you have seen them all. Don't worry about the defined constants, just concentrate on the overall operation of the function.

Listing 15.6 - A function that converts token strings of an input sentence into integer tokens.

```
int Extract_Tokens(char *string)
{
// this function breaks the input string down into tokens and fills up
// the global sentence array with the tokens so that it can be processed

int curr_pos=0,      // current position in string
    curr_token=0,    // current token number
    found,           // used to flag if the token is valid in language
    index;           // loop index

char output[16];
// reset number of tokens and clear the sentence out
num_tokens=0;

for (index=0; index<8; index++)
    sentence[index]=0;

// extract all the words in the sentence (tokens)
while(Get_Token(string,output,&curr_pos))
     {
     // test to see if this is a valid token
     for (index=0,found=0; index<NUM_TOKENS; index++)
         {
         // do we have a match?
         if (strcmp(output,language[index].symbol)==0)
```

```
            {
            // set found flag
            found=1;

            // enter token into sentence
            sentence[curr_token++] = language[index].value;
            break;
            } // end if

         } // end for index

         // test if token was part of language (grammar)
         if (!found)
            {
            printf("\n%s, I don't know what \"%s\" means.",you.name
                                                          ,output);

            // failure
            return(0);

            } // end if not found

         // else
         num_tokens++;
      } // end while
} // end Extract_Tokens
```

The function operates with an input string passed to it containing the user's commands. The function then proceeds to break the sentence down into separate "words" using the *Get_Token()* function. Each word is then scanned for in the vocabulary table and when a match is made the word is converted into a token and inserted into a token sentence which is basically a version of the input sentence in token form (integers instead of strings). If the "word" is not found in the vocabulary table then there is a problem and the code will emit an error. I have highlighted this section in the function above, so that you may see how easy it is.

Let's take a brief detour for a moment to cover two topics. The first is error handling. This is always an important part of any program and I can't emphasis how important it is especially in a text based game where a bad input can "trickle" down into the bowels of the game engine and logic and really mess things up (you PERL scripters should know something about this). Therefore, it can't hurt to have too much error checking in a text game. Even if you are 99% sure an input to a function should be of the correct form, always test that one last case to make sure.

Secondly, you will notice that I use very primitive data structures. Sure a linked list or binary tree might be a more elegant solution, but is it really worth it for a dozen or so vocabulary words? The answer is no. My rule of thumb is the data structure should fit the problem and arrays fit a lot of small problems. When the problems get large then it's time to bring in the big guns such as linked lists, B-trees, graphs, and so forth. But for small problems, get the code working with simple data structures or else you will

be forever trying to figure out why you keep getting *NULL* pointer errors, GP-faults and other bothersome features!

All right, now that we finally know how to convert the sentence into tokens now it's time for the syntactic and semantic analysis phases, let's cover the syntactic phase first.

Syntactical analysis

Hold on tight because here is where things start getting a bit cloudy. Strictly speaking syntactic analysis is defined in compiler texts is the phase where the input token stream is converted into grammatical phrases so that these phrases can be processed. Since the languages we have been considering for implementation are fairly simple and have simple vocabularies. This highly general and elusive definition needs to be pruned down to mean something in our context. As far as we are concerned, syntactic analysis will mean "making sense out of the sentence". This means applying the verbs, determining the objects, extracting the prepositions, articles, and so on.

The syntactic analysis phase of our games will occur in parallel with the action(s) processing of the sentence. This is totally acceptable and many compilers and interpreters are designed in this way. The code is generated or interpreted "on the fly". We have already seen what the syntactic phase of analysis look like. It is accomplished by calling functions that are responsible for all the action verbs in the language. Then these action function further processes the remainder of the input sentence which is already in token form and then try to do whatever is supposed to be done. For example, a syntactic parser for a text game might begin by figuring out which action verb began the sentence and then calling an appropriate action function to deal with the rest of the sentence.

Of course, the rest of the sentence could be processed and tested for validity before the call to the action function, but this isn't necessary. I prefer to place the burden of further processing on the action functions. My philosophy is this: each action function is like an object that operates on a single kind of sentence. This single sentence is one that starts with a specific word. We shouldn't hold a single function responsible for checking the syntax of all the possible sentences before making the call to the appropriate action function.

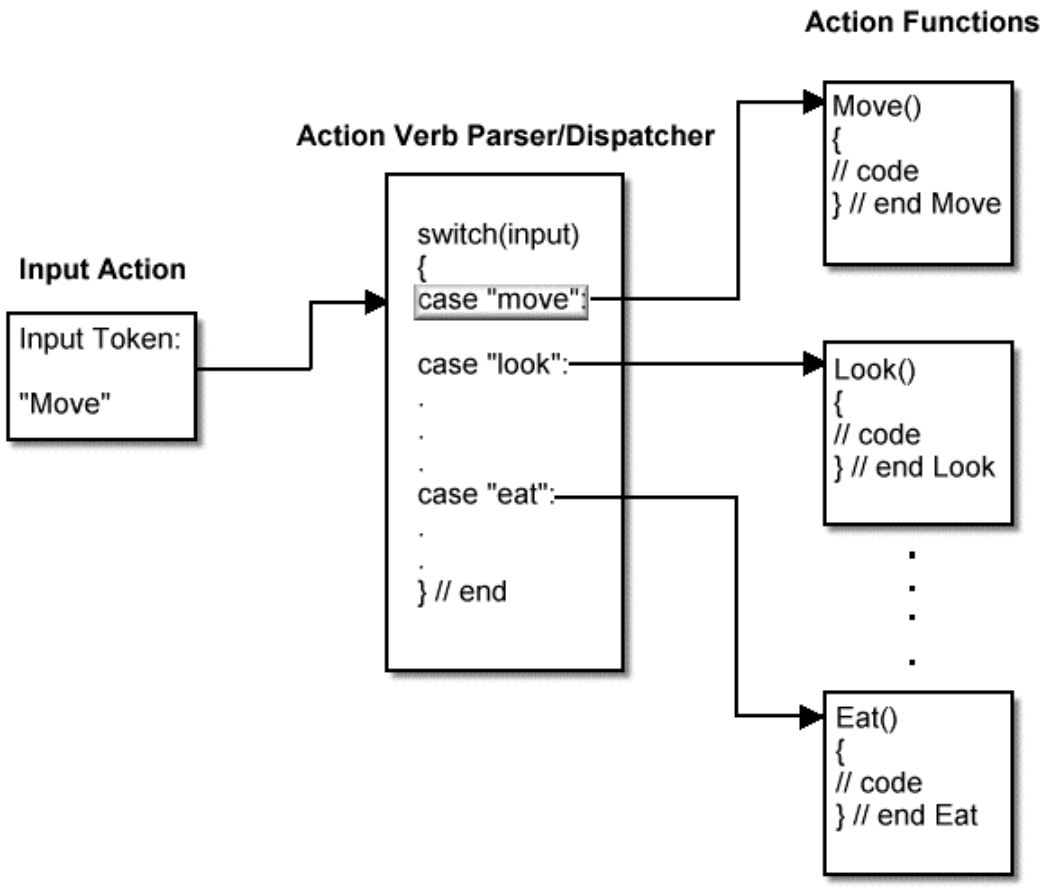
As an example, below is the a function used in Shadow Land to dispatch each sentence to the proper action function. The single purpose of this function is to look at the first word in the sentence (which is a token of course) and then vector to the correct action function. At that point, it's the action function's job to check the rest of the syntax of the sentence. Here is the action dispatcher.

Listing 15.7 - The function used in Shadow Land to call the action functions based on the action verb used in the input sentence.

```
void Verb_Parser(void)
{
// this function breaks down the sentence and based on the verb calls
the
// appropriate "method" or function to apply that verb
// note: syntactic analysis could be done here, but I decided to place
it
// in the action verb functions, so that you can see the way the errors
are
// detected for each verb (even though there is a lot of redundancy)

// what is the verb?

switch(sentence[FIRST_WORD])
      {
      case ACTION_MOVE:
           {
           // call the appropriate function
           Verb_MOVE();
           } break;

      case ACTION_TURN:
           {
           // call the appropriate function
           Verb_TURN();
           } break;

      case ACTION_SMELL:
           {
           // call the appropriate function

           Verb_SMELL();
           } break;

      case ACTION_LOOK:
           {
           // call the appropriate function
           Verb_LOOK();
           } break;

      case ACTION_LISTEN:
           {
           // call the appropriate function
           Verb_LISTEN();
           } break;

      case ACTION_PUT:
           {
           // call the appropriate function
           Verb_PUT();
           } break;

      case ACTION_GET:
           {
```

```
            // call the appropriate function
            Verb_GET();
            } break;

      case ACTION_EAT:
            {
            // call the appropriate function
            Verb_EAT();

            } break;

      case ACTION_WHERE:
            {
            // call the appropriate function
            Verb_WHERE();
            } break;

      case ACTION_INVENTORY:
            {
            // call the appropriate function
            Verb_INVENTORY();
            } break;

      case ACTION_EXIT:
            {
            // call the appropriate function
            Verb_EXIT();
            } break;

      default:
              {
              printf("\n%s, you must start a sentence with an action
verb!",
                     you.name);
              return;
              } break;

      } // end switch

} // end Verb_Parser
```

Figure 15.3 - An action verb being dispatched to the proper action function.



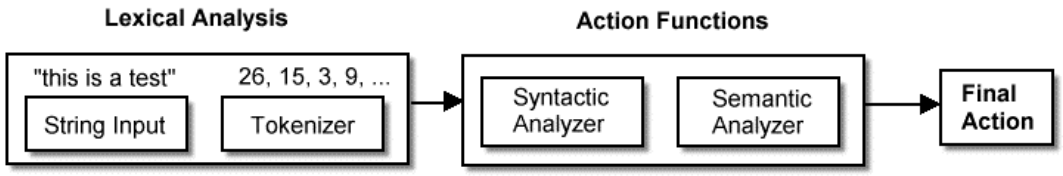
The function is beautifully simply. It looks at the first word in the sentence (which by definition of the language better be a verb) and then calls the corresponding action function. For a graphically representation of this take a look at Figure 15.3. The next part of parsing analysis is the semantic analysis phase.

Semantic analysis

Semantic analysis is where most of the error checking takes place. However, since we are doing the syntactic analysis "on the fly" along with making things happen in the game, this responsibility is merged into the action function during syntactic analysis. Syntactic analysis is supposed to break the sentence down into its meaning and then semantic analysis is used to determine if this meaning makes sense (getting confused yet!). As you can see, this business of syntactic and semantic analysis seems to be almost circular and therefore, again we will cut to the quick and make a hard fast rule. In a text based game, the syntactic and semantic analysis are done simultaneously. In other words, **as the sentences meaning is being computed the sentence is tested to make sense.**

I hope you understand this very subtle concept. If it helps it's kind of like snow skiing. It easy once you can do it, but until then you have no idea how to do it.

Figure 15.4 - The sections of the input parser.

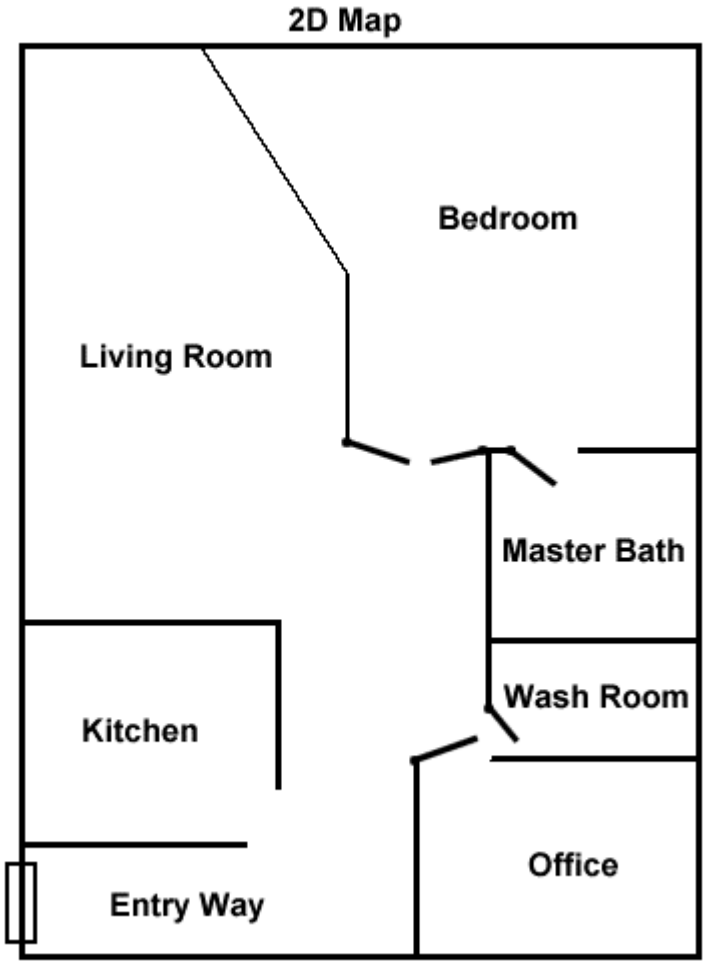


Putting all the pieces together

We have all the components of the complete parser and text interpreter engine. Let's see how all the pieces go together to form the backbone of a text game, see Figure 15.4. Referring to the figure, we see that the input stream is broken into words by the front end of the lexical analyzer, then the words strings are converted into integer tokens. The token stream is then feed into the front end of the syntactic parser which "sends" the sentence of tokens to the proper action function. The action function then tries to apply the action requested by the player to the object(s) in the sentence. As the action is being applied, syntax checking, and semantic integrity are also being tested simultaneously by the action function.

The important concept to grasp is that all the input parsing, syntax checking, and so forth are all for the single reason of trying to figure out what the player wants to do. Once this is determined then the coding is straightforward. The only caveat is that the actions performed are acted upon internal data structures. And the only output is text. Remember, the player only "views" the game universe by way of strings of static or synthesized text that the game outputs as each "move" is made in the game. Hopefully, you have a grasp of how the text input phase of a text game works. If I were to leave the subject now, you should be able to make a text game, but instead let's cover some techniques to control and manipulate other aspects of a text game. However, be warned these techniques are illustrative only and I'm sure there are better, different and more clever ways of doing them.

Figure 15.5 - A top down "blue print" view of an environment.



Representing the universe

The universe in a text game can be represented in many ways. Whatever representation is used, it should have some kind of geometrical relationship to the virtual environment the player is running around in. For example, if you created a full 3-D space text adventure then you might have a 3-D model in a database. Even though the player would never see the world visually, all collision detection and motion would be done within the 3-D model. Since this book is primarily about 2-D games we'll make our representations more flat. For a moment I want you to think of a one story house. Imagine the roof is removed and you are looking down at the house from above (your in a hover craft), see Figure 15.5. You could see the rooms of the house, the objects and the people moving around. This is the kind of representation we want to use for our text games.

How do we implement a data structure of such a model? It can be done in two ways. We could use a vector based model based on lines and polygons. There would be a database of this geometry and the game code would use it to move the game objects around. This technique is fine, but a bit hard to visualize since you would be working in pure "vector space". A better approach would be to have a 2-D matrix of cells that represented the game universe.

Admitingly, you will never "see" the world, but having an actual 2-D floor map that can be accessed very simply will make the game code very simple to write. For example, the game Shadow Land uses a 2-D matrix of characters to represent the game universe (which is my old apartment). Take a look below to see the actual data structure used in the game.

Listing 15.8 - The data structure used to represent the game universe of Shadow Land.

```
// this array holds the geometry of the universe

// l - living room
// b - bedroom
// k - kitchen
// w - washroom
// h - hall way
// r - restroom
// e - entry way
// o - office

//                  ^
//                NORTH
// < WEST           EAST >
//               SOUTH
//                  v

char *universe_geometry[NUM_ROWS]=
{"******************************",
 "*lllllllll*bbbbbbbbbbbbbbbbbbb*",
 "*lllllllllll*bbbbbbbbbbbbbbbbbb*",
 "*lllllllllllll*bbbbbbbbbbbbbbbbb*",
 "*lllllllllllll*bbbbbbbbbbbbbbbbb*",
 "*lllllllllllll*bbbbbbbbbbbbbbbbb*",
 "*lllllllllllll*bbbbbbbbbbbbbbbbb*",
 "*lllllllllllll*bbbbbbbbbbbbbbbbb*",
 "*lllllllllllll*bbbbbbbbbbbbbbbbb*",
 "*lllllllllllll*bbbbbbbbbbbbbbbbb*",
 "*lllllllllllll*bbbbbbbbbbbbbbbbb*",
 "*lllllllllllll*bbbb*rrr**********",
 "*lllllllllllhhhhhh*rrrrrrrrrrr*",
 "*lllllllllllhhhhhh*rrrrrrrrrrr*",
 "*lllllllllhhh*****rrrrrrrrrrr*",
 "*********hhhh*rrrrrrrrrrrrrrrr*",
 "*kkkkkkk*hhhh*rrrrrrrrrrrrrrrr*",
 "*kkkkkkk*hhhh*rrrrrrrrrrrrrrrr*",
 "*kkkkkkk*hhhh*rrrrrrrrrrrrrrrr*",
```

```
"*kkkkkkkhhhhh*******************",
"*kkkkkkkhhhhhhhhhhhwwwwwwwwwwww*",
"*kkkkkkkhhhhhhhhhhhwwwwwwwwwwww*",
"*kkkkkkk*hhhhhhhhhhwwwwwwwwwwww*",
"*kkkkkkk*hhhh*ooooo*************",
"*kkkkkkk*hhhh*oooooooooooooooo*",
"*kkkkkkk*hhhh*oooooooooooooooo*",
"*kkkkkk*hhhhh*oooooooooooooooo*",
"*******hhhhhh*oooooooooooooooo*",
"*eeeeeeeeeee*oooooooooooooooo*",
"*eeeeeeeeeee*oooooooooooooooo*",
"*eeeeeeeeeee*oooooooooooooooo*",
"*********************************",};
```

It might be a bit hard to see the rooms due to the print font, but if you blur your eyes a little you will be able to see all the rooms. You will notice that walls are defined by the '*' character and different rooms are defined by placing a single character within the confines of the room such as 'o' for office. Placing characters in the each room that describe the room type is helpful in determining where the player is at any time. If the player is placed at any (x,y) position in the map then the code that prints out "where" the player is only has to index into the array and look at the character the player is standing on. Then using this information the appropriate text strings can be displayed.

Placing objects in the world

Objects in the game world can be done using two main techniques. The first technique is to have an array or linked list of objects. Where each element defines the object, it's position and any other relevant properties. The only problem with this technique is that when a player is in a room and asks to "look" then the object list must be accessed and visibility must be determined of each object. Also, collision detection, "putting" and "getting" are more complex when the objects are stored in this way. An easier method that is related to the way the universe is defined is to have another 2-D matrix of characters with the same geometry as the universe map. However, instead of walls and floor tiles within it, the objects are placed in it. Then you can imagine that both data structures are synthetically overlaid by the game code. For example, here is the object placement map for Shadow Land.

Listing 15.9 - The object placement database for Shadow Land.

```
// this array holds the objects within the universe

// l - lamp
// s - sandwich
// k - keys
//            ^
//          NORTH
// < WEST           EAST >
///         SOUTH
//           v
```

```
char *universe_objects[NUM_ROWS]=
{"                                ",
 " l                           k ",
 "                                ",
 "                                ",
 "                                ",
 "                                ",
 "                                ",
 "                                ",
 "                                ",
 " l                              ",
 "                                ",
 "                                ",
 "                                ",
 "                                ",
 "                                ",
 "                                ",
 "                                ",
 "                                ",
 "                                ",
 "                                ",
 "  s                             ",
 "                                ",
 "                                ",
 "                                ",
 "                                ",
 "           s                    ",
 "                           l    ",
 "                                ",
 "                                ",};
```

As you can see the object map is very sparse; and it should be since it contains only the objects in the universe. However, determining if the player is near an object or even on top of one is very simple since there is a 1-1 relationship between the geometry of the universe, universe of objects and the position of the player. The only drawback to using the "cell" technique to represent a game universe for a text game is that the cell world has only a finite number of cells and thus positions within the universe, but this is really a small inconvenience and the player will never know!

Making things happen

We previously learned the making things happen merely means to print out some text or change some values in data structures such as the position of the player. A text game is really just a "live" database that is accessed and modified as the game is played. Frankly, an arcade game is also, but there is a more complex output device being used (the screen) and arcade games must be in real time. Text games wait for the user to input a text string before doing anything. When this "waiting" is taking place the code is stuck in the line input function and the game universe is "motionless" during this period.

Moving around

Motion in a text game is accomplished by changing the position of the player's character in the game. This is usually as simple as modifying a couple variables. For example, if the player typed:

Player's input:

"move north"

Computer's output:

"You take a few steps…"

The game code might decrement the Y position of the player's character (North is in the negative Y direction) and then that would be it. Of course the player's position would be tested to see if the player stepped on something, hit a wall, or fell off a cliff, but the actual motion is accomplished with only a couple variable changes. This simplicity stems from the fact that the player is plopped down in the 2-D map and is just a square that can either move East, West, North, or South.

The inventory system

The inventory system is a text game is a list that contains a description and tally of all the objects that the player is holding on himself/herself. This list can be an array, a linked list or whatever. If the objects are complex then the list might be a list of structures. In Shadow Land the inventory is so simple that nothing more than an array of characters was used to hold the inventory. The player can have a sandwich 's', a set of keys 'k' and a lamp 'l'. If the player is holding one or more of these objects then in a character array called *inventory[]* the characters representing these objects are stored. Then when the player asks to see what he is holding, then a simple traversal of this list and a few output strings are all that is needed.

But how does the player get the objects? Well, he/she picks them up from the game universe. For example, if a player was in a room a "saw" the keys then he/she might request the game to "get" the keys. The game would then "pick up" the keys by removing the 'k' from the object universe, replacing the spot the keys were in with a blank and then inserting a 'k' into the inventory list of the player. Of course, making the computer "see" what's in a room and ascertain if the player is within reach is a bit complex, but you get the idea.

Implementing sight, sound, and smell

The implementation of the human senses in a text game is the most challenging of all since the player can't really, see, hear or smell. This means that apart from the algorithmic considerations the output descriptions must be full of adjectives, descriptives, qualifiers, and so forth to create mental images. Implementing sounds and smells are the easiest since they are not focused senses. By focused, I mean that if a specific room has a general smell then the player should smell it at any location of the room. Similarly, the same goes for sounds. If the room has music playing then the music will be heard in any part of the room. Vision is the most complex because it is more focused than sound and smell and is much harder to implement. Let's take a look at all three senses and see how to implement them.

Sound

There are two types of sounds in a text game; ***ambient*** and ***dynamic***. Ambient sounds are the sounds that are always in a room and dynamic sounds are sounds that can enter and leave a room. First let's talk about the ambient sounds. To implement ambient sounds there should be a data structure that contains a set of descriptive strings for each room. Then when the player asks to "listen" then these strings are printed out by testing the room the player is standing within and using this information to select the correct set of strings. For example, if the player types in listen in a machine shop then this might be the monologue he/she is presented with:

What do you want to do? Listen

…You hear the sounds of large machines all around you. The sounds are so strong and piercing you feel them in your teeth. However, beyond all the sounds of the large machines, somewhere in the background you here a peculiar hum, but you're not sure what it is…

The data structure containing these static strings is up to you, but I suggest either an array of strings with a field that describes which room the string is four our an array of structures with a structure for each room. For example, here is the ambient sound string structure data structure used for Shadow Land along with the sounds for the game.

Listing 15.10 - The static data structure used to contain the informational strings in Shadow Land along with the ambient sounds in the game.

```
// this is the structure used to hold a single string that is used to
// describe something in the game like a smell, sight, sound...

typedef struct info_string_typ
        {
        char type;        // the type of info string i.e. what does it describe
        char string[100]; // the actual description string
```

```
        } info_string, *info_string_ptr;

// these info strings hold the smells in each room
info_string smells[]={

{'l',"You smell the sweet odor of Jasmine with an undertone of potpourri. "},
{'b',"The sweet smell of perfume dances within your nostrils...Realities possibly. "},
{'k',"You take a deep breath and your senses are tantalized with the smell of"},
{'k',"tender breasts of chicken marinating in a garlic sauce. Also, there is "},
{'k',"a sweet berry smell emanating from the oven.                          "},
{'w',"You are almost overwhelmed by the smell of bathing fragrance as you"},
{'w',"inhale.                                                              "},
{'h',"You smell nothing to make note of. "},
{'r',"Your nose is filled with steam and the smell of baby oil... "},
{'e',"You smell pine possible from the air coming through a small orifice near"},
{'e',"the front door.                                                      "},
{'o',"You are greeted with the familiar odor of burning electronics. As you inhale"},
{'o',"a second time, you can almost taste the rustic smell of aging books.        "},
{'X',""}, // terminate
};
```

As you can see there are strings for each room in the game and the end of the strings is delineated with a 'X' character. This is just one way to do things but it works for me!

Dynamic sounds are more complex to implement than static ones since they can move around the environment. Shadow Land has no dynamic objects, but I will explain how to implement dynamic sound. Each object that can move around the universe has a sound attached to it which is just a string that describes the sound the object makes. Hence, when a player asks to listen to the sounds in a room, first the static sound is printed out then it is determined what objects are in the room and their "sounds" are printed out also after the static portion of the text. Of course, you should try and make the sentences "connect" together with some kind of conjunction so the sounds don't look like a bullet list!

Smell

The sense of smell is programmed in the exact same way sounds are except that the text strings should describe the smells in the room instead of the sounds. Moreover, dynamic smells are implemented in the same way as described above as an "attachment" to the dynamic objects. For example, an ogre might have a bad smell attached to it that is described along with the static smell of the room. This would be determined by testing if the ogre is in the same room as the player. Which is easy since we can look at the position of the ogre and of the player, index into the universe map and see if they are on the same cell type. Before moving on, there is one artistic aspect of describing smells we should touch on. If something smells bad then try to make it sound as if it smells bad in a good way, get my drift?

Sight

Implementing vision in a text game is an interesting problem. We want to make a virtual character "see" in a virtual world that isn't even visible to the player! Well, being the software sorcerers that we are this is no problem at all if the correct data structures and approach are used. First, the easiest data structure to implement sight

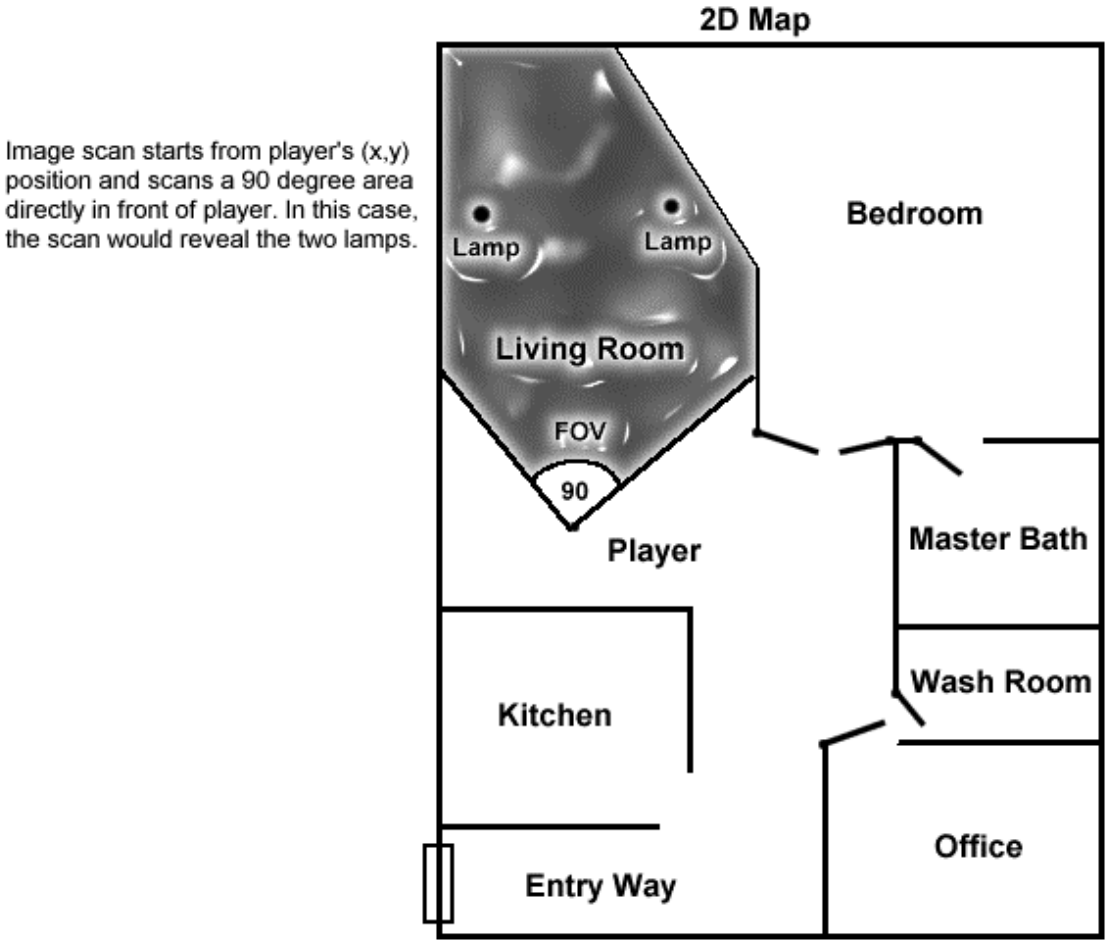
with is the map worlds since it is a 2-D version of the world the player is walking around in. The algorithm is easy also, all we have to do is understand how we (ourselves) see and then implement a version of our sight within the text game that is based on the data structures in the game.

To begin with, there is static vision and dynamic object base vision. The static vision is taken care of as we have seen and is very general. If a player asks to "look" at a room then the first part of the description would be a static one that is always the same. Then the second part of the description might focus on what's within the player's virtual field of view or VFOV (I like acronyms). This is the hard part. We must somehow scan in front of the player and detect if objects are within this scan space. Unlike the smell and sound test, we just can't check if the dynamic object is within the room since it might be behind the player. We must instead see if it within room in conjunction with being within the VFOV.

To test if an object is within the VFOV of view of the player we need to know five things:

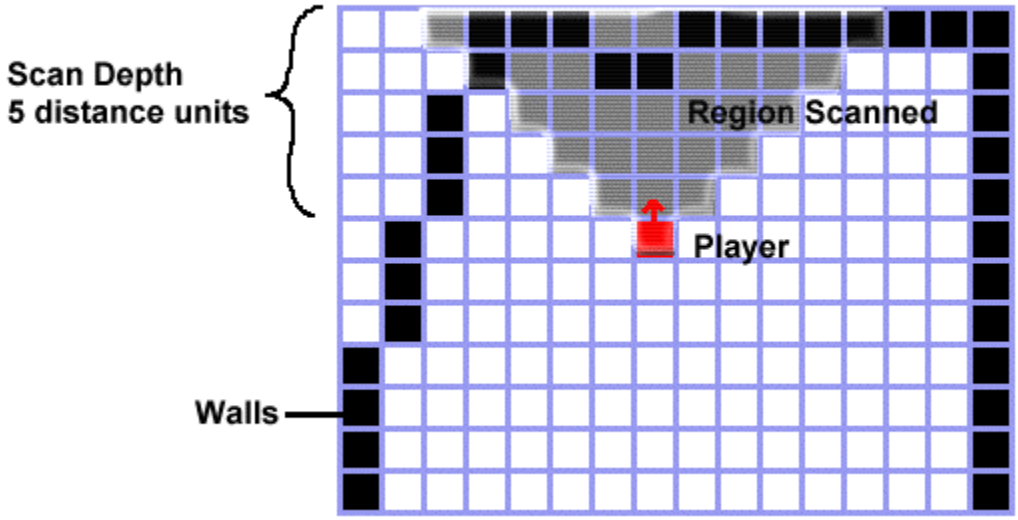
1. The position of the player.
2. The direction of the player.
3. The positions of all the dynamic objects.
4. The depth of the scan (distance).
5. The view angle of the scan.

Figure 15.6 - The virtual vision system in action.



Number 1,2, and 3 are easy. We can find that information by looking at the data structures for the player and all the dynamic objects. The only question arises when we consider numbers 4, and 5. First, the depth means how far should the player be able to see? This is relative, but as a rule of thumb the player should be able to see at least the length of the largest room. The second factor "view angle" is really the field of view and simply means the angle at which the scan will take place, see Figure 15.6. Now, sine the player can only be facing four directions (North, South, East and West) the scan become very simple. All that is needed are two "for" loops to accomplish the scan.

Figure 15.7 - A close up of the player's vision scan.



The scan will have a shape of an upside down pyramid and the view and or field of view will be 90 degrees. This is close enough to a normal humans field of view and is realistic for a game. Hence, the scan will emanate out from the players position and test each cell in the scan, see Figure 15.7. In general, one *for* loop will control the X axis deflection of the scan and one *for* loop will control the Y axis deflection. The scan will continue until a specified depth is reached (which is the distance) and then the vision scan will be complete. Now, as the vision scan is running, tests are made to see if any of the blocks within vision contain a dynamic object. If this is true then each object is "tagged" and placed in a list. Then when the vision strings print out, the list is referred to and the specific visual strings for each dynamic object are printed out along with the general static view.

Shadow Land uses this technique to "see" in room. As an example, here is the vision code that takes care of vision in the Northern direction. Pay close attention to the structure of the *for* loops and the tests. Don't worry if some of the variables or defines are not visible, just try and understand the overall operation of the code fragment.

Listing 15.11- A code fragment from Shadow Lands that produces a vision scan in the Northern direction.

```
case NORTH:
{
// scan like this
//  .....
//   ...
//    P
for (y=you.y,scan_level=0; y>=(you.y-depth); y--,scan_level++)
    {
    for (x=you.x-scan_level; x<=you.x+scan_level; x++)
```

```
        {
        // x,y is test point, make sure it is within the universe
        // boundaries and within the same room
        if (x>=1 && x<NUM_COLUMNS-1 &&
           y>=1 && x<NUM_ROWS-1 &&
           universe_geometry[y][x]==universe_geometry[you.y][you.x])
           {
           // test to see if square has an object in it
           if (universe_objects[y][x]!=' ')
              {
              // insert the object into object list
              stuff[*num_objects].thing = universe_objects[y][x];
              stuff[*num_objects].x     = x;
              stuff[*num_objects].y     = y;
              // increment the number of objects
             (*num_objects)++;

              } // end if an object was found
            } // end if in boundaries
          } // end for x
      } // end for y

// return number of objects found
return(*num_objects);

} break;
```

The function works by scanning the area in front of the player and testing each block in the scan to see if it is within the universe. If the block is within the universe then the objects data structure is referred to see if an object is sitting on the block. If so, then the object is inserted into a list and the code proceeds until the scan is done. You should pay close attention to the bounding code that determines if the scanned block is within the universe. This is absolutely needed since as the scan takes places a given distance, the code doesn't have any way of determining of it has gone out of bounds of the array or in the negative direction. This is why a "filter" is needed to condition and test each test block position that is generated by the double *for* loops. Finally, when you print out the dynamic objects after the static visual make sure they read in a fluid manner!

Making it Real-Time

The text game technology we have been considering thus far is not real time. This is because the game logic waits for the user to input the text string, the string is processed and the game logic executed and the cycle starts over. The problem is that during the text input phase the game logic is in a wait state. This has two effects on the game. First, time stops until the player inputs a string and presses return. Secondly, dynamic objects in the game can't move around, in essence the game universe can't evolve while the player is "thinking". This may or may not be desired. There are some text games that stop and wait and other run while waiting for text input.

For example, in a real-time text game, the player may sit at the command prompt for ten minutes, but as he/she does this he/she is being surrounded by orcs! To make a game real-time is very easy, all you need to do is slightly change the input function so that it resembles a keyboard handler that only sends the string to the input parser when the return key is kit. So, the game event loop might look like this.

```
while(!done)
     {
     Get_Next_Input_Character();

     if (user has typed in a whole string)
        {
        Parser_Logic();
        } // end if an input was entered

     // whatever happened above, let's move all the

     // objects in the game universe
     Move_Objects();
     } // end main loop
```

The real-time aspect of this structure is that the input character is not waited for, if there is one fine; otherwise, game logic continues in real-time.

NOTE:The game Shadow Land doesn't have any "living" objects in the universe, so a real-time implementation wasn't needed.

Error handling

A text game must have error handling just as arcades do. However, most of the error handling is done in the text parsing. There are lexical tests, syntax tests, and semantic tests that all must be performed. Lexical tests are easy since they have to do with string comparisons and vocabulary checks. The syntax and semantic tests are more complex since the game logic must start using language productions and rules to check if the sentence is valid. This is complex and there are usually many special cases that have to be considered. Similarly, when writing a compiler or interpreter the code starts off clean, but has a lot of kludges in it by the completion of it. Text games are very similar, you may find yourself doing some lexical testing in the string input function and some syntax checking in the lexical analyzer. However, better to be safe than sorry! So, the more filters, traps and tests that the input parser has the better!

Creeping around with Shadow Land

Well I know that you are an expert text game writer at this point, but I wanted to show you an example of such a game. The name of the game is called ***Shadow Land*** and is a fully operational text game that allows you full interaction with the environment. The idea behind the game is simple, you must find your keys and drop them in the office. The environment you will be playing in is my old apartment in Silicon Valley, CA. I have completely modeled my apartment for you along with the sights, sounds, and smells that were usually in it.

The idea behind the game is that you are an invisible shadow that can move freely around without being detected by me or anyone else in the apartment. The vocabulary and grammar of the game is very simple as is the actions you can perform, but if you understand Shadow Land then you shouldn't have a problem making something like Zork with the proper planning, data structures and algorithms. Moreover, if you are interested in creating RPGs at all then this is something you need to understand!

The language of Shadow Land

In table 15.1, we already saw the vocabulary of Shadow Land, but now let's take a look at the productions or syntax rules. But instead of stating them in a rigorous manner let's try and list them in a bit more of a relaxed way. Let's begin by listing all the word in the vocabulary again.

- The objects in the game

  LAMP,SANDWICH,KEYS

- Nautical directions

  EAST,WEST,NORTH,SOUTH

- Relative directions

  FORWARD,BACKWARD,RIGHT,LEFT

- The "actions" or verbs of the language

  MOVE,SMELL,LOOK,LISTEN,PUT,GET,EAT,INVENTORY,WHERE, EXIT

- The articles or connectives of the language

THE

- The prepositions of the language

  IN,ON,TO,DOWN

Let's begin with the legal form of the action verbs. Some of the verbs need no object to mean something. These are "smell", "listen", "inventory", "where", and "exit". If any of these action verbs are typed by themselves they will work. Furthermore, if prepositional phrases, articles, or objects are placed after the verbs they will have the effect of causing warnings. The outcome of entering any of these verbs is below:

"smell" - This will describe the smell of the room you are in.

"listen" - This will describe the sounds of the room you are in.

"inventory" - This will tell you what you are carrying.

"where" - This will describe your location in the house along with the direction you are facing.

"exit" - This will end the game.

The next set of action verbs can be further qualified by either objects, adjective or complete prepositional phrases. These verbs are "move", "put", "get","look" and "eat". Here are the legal forms using some of the earlier production rule syntax.

"move" + (relative direction) | "move" + "to" + relative direction |
"move" + "to the" + relative direction

Where the parenthesis mean that the word(s) are optional and "|" means logical OR.

Using the above production the following sentence would be legal:

"move to the right"

This would have the effect of having the player parry right (sidestep). Another possibility would be:

"move"

This would have the effect making the player walk in the direction he/she was currently facing. An illegal sentence would be:

"move to the east"

This is illegal since "east" is a nautical direction instead of a relative direction. The next interesting action verb is "look", here is its rule:

"look" + (nautical direction) | "look" + "to" + nautical direction |
"look" + "to the" + nautical direction

Using this verb the player can "see" objects in the room. For example, if the player just typed in "look" without a direction then the game would print out the static visual only. To see objects in a room you MUST use "look" combined with a nautical direction. For example, to see the Northern part of the room, you would type.

"look to the north"

or

"look north"

or

"look to north"

All of the above forms all equivalent and the result of them will be the objects within the players view being described.

The player will start the game off facing North, so there needs to be a way to turn him/her. This is done with the "turn" action which is similar to the "look" verb as far as the productions rules go.

"turn" + (nautical direction) | "turn" + "to" + nautical direction |
"turn" + "to the" + nautical direction

Hence, to turn East you can type:

"turn to east"

The next two action verbs relating to object manipulation are "put" and "get". They are used to put down and pick up objects. The only valid objects in the game are the keys, lamp and sandwich. Here are the production rules for each action verb.

"put" + object | "put" + "down" + "object" | "put" + "down the" + object

"get" + object | "get" + "the" + "object"

You use the "put" and "get" to move objects around the environment. For example, say that you saw a set of keys in front of you, then you might say this:

"**get the keys**"

> WARNING: When "getting" objects you must be near them, so even though you can see an object, you may not be able to grasp is until you move close enough.

Then later you may wish to drop the keys. This would be accomplished with something like this:

"**put down the keys**"

The final most important action verb is "eat" and I'm sure you know what it does. It will make you eat whatever you tell it to as long as you have the object in your possession. The rule for "eat" is:

"eat" + object | "eat" + "the" + "object"

So, if you wanted to eat the lamp, you would type:

"**eat the lamp**"

After which you will have satisfied your iron requirements for the day!

At first you will find the grammar and limited vocabulary tedious, but after a few moments of playing the game, it will become very natural to you. You are of course free to add to the vocabulary and grammar rules. One final detail, the input parser is case insensitive, so you can use upper and lower case characters at will.

Building and playing Shadow Land

Shadow Land is the only game so far that is almost totally portable. This is due to the fact that the game is text only. The only two things that might change this statement is that Shadow Land uses *kbhit()* and the ANSI color text driver. Other than that, the code is straight C/C++ without any graphics or machine dependent calls. To build the program use the source module called SHADOW.CPP, compile it as a CONSOLE application and link it with the standard C/C++ libraries and that's all you need to do. I have created an executable for you already called SHADOW.EXE if you don't want to do this.

You already know just about everything you need to know to play. However, here are a few tips. The game will begin by asking you your name. Then it will ask you what you want to do. At this point in the game you are standing in the entryway to my apartment. To your left is a kitchen and to the north is a hallway. Move around in the environment with the "move" command and be sure to listen and smell everything. Remember "move" by itself will always move you in the direction you are currently facing and "look" needs to be qualified by a nautical direction if you wish to see the objects in the room.

Shadow's game loop

The game loop for Shadow Land is extremely simple since it's not in real-time. Here it is:

Listing 15.12 - The game loop of Shadow Land

```
void main(void)
{

// call up intro
Introduction();

printf("\n\nWelcome to the world of  S H A D O W  L A N D...\n\n\n");

// obtain users name to make game more personal
printf("\nWhat is your first name?");
scanf("%s",you.name);

// main event loop,note: it is NOT real-time
while(!global_exit)
     {
     // put up an input notice to user

     printf("\n\nWhat do you want to do?");

     // get the line of text
     Get_Line(global_input);

     printf("\n");
     // break the text down into tokens and build up a sentence
     Extract_Tokens(global_input);

     // parse the verb and execute the command
     Verb_Parser();

     } // end main event loop

printf("\n\nExiting the universe of S H A D O W  L A N D...see you later
%s.\n",you.name);

// restore screen color
printf("%c%c37;40m",27,91);

} // end main
```

NOTE: Be sure to look at the use of the ANSI screen driver which is activated by sending out ESC followed by "[" and the appropriate commands. For VC++ CONSOLE applications it doesn't really work, but if you compile with a compiler that creates old DOS applications then the ANSI stuff will work as long as you load the ANSI driver which is usually in your DOS directory and named ANSI.EXE). The ANSI driver allows colored text and other special effects.

The *main()* begins by printing an introduction screen and then asking the player for his/her name. At this point the game falls into the main event loop which is static. The loop will wait for the user to type in a string which is returned from *Get_Line()*. Then the string is tokenized by *Extract_Tokens()* and finally parsed and acted upon the verb parser named *Verb_Parser()*. That's all there is to it. This cycle will occur every time the player enters a line of text until he/she types "exit".

Winning the game

I think I already told you how to do this? But if I didn't then good luck figuring it out!

Summary

This chapter has definitely been a burn for both of us. You have taken a crash course in compiler design along with learning the details of implementing a text game. You learned about universe representations, how to implement the senses and how to make the descriptions in a text game "fluffy" and fun. Finally, you got to see where I used to live!