Interactive Fiction Language – Final Report

Gema Almoguera, UNI GA2347

COMS W115 – Summer 2013

Contents

1. Introduction

1.1 Description of the language

The proposed project is an interactive fiction language to program text-based adventure games.

This is a type of text game that mixes writing and programming. The writer creates the code so that the player becomes the main character of the game. The player interacts with the elements of the game by using a command line to type two-word actions. These actions can be movements (go north, go south, etc.), interactions with the environment (take sword, play flute) or game commands (show inventory, look around). The story progresses as the player performs the right actions, uses the right items, and explores the correct rooms.

Throughout this document, the words "writer" and "programmer" are used indistinctively, since they are synonymous in our context: they represent the creator, writer, and programmer of the game.

The purpose of this project is to write a reduced version of one of the most used programming languages for interactive fiction, called TADS3 (Text Adventure Development System version 3), and so the syntax and semantics used look close to the original language.

1.2 How it should be used

Once the writer has a story, he or she will create the rooms, items, and actions in the story, defining how the game must behave depending on the commands typed by the user through a command line.

More concretely, the programmer will be able to create item and location objects. These objects can have properties (similar to attributes in Java) and actions (similar to methods in Java). In fact, the whole program is just a list of object declarations: a game declaration, one or more room declarations, and one or more item declarations.

The programmer will use these methods to attach actions to each object and specify the resulting behavior of such actions. For example, if there is an object in our game called 'flute', we can write a method to explain how the story progresses if the player types 'play flute'. Actions that are applied to objects are composed of a verb in the imperative mood and a noun (the item name).

The writer can use if/else, while, and for clauses within the methods, as well as checking whether other items are in the player's inventory. The main types used are strings, integers, and booleans, which have proven sufficient to write basic games.

Once the code is finished, the writer can compile the algorithm into C code. This C code can then be compiled, along with the interpreter provided (also written in C code), to create the full game.

1.3 Code examples

The programmer will start by defining a game object. The game object must include the name of the room where the player starts, as well as the introduction to the game:

```
gameMain: GameMainDef
 {
 initialRoom =yourRoom;
 intro =
     "Welcome to your new adventure! You are a new castle guard looking for the queen’s
     lost ring. If you find it, the queen will allow you to become her personal chef – Your
     childhood dream! Armed only with your sword and a lamp that’s running out of oil, you
     find yourself at the entrance of a cave. It’s getting dark and you can hear the wolves
     howling close. What are you going to do now?”;
  }
 }
```

This is an example of the definition of a room object, with an in-game name, description, and an exit as properties. As seen below, comments are allowed and their syntax is similar to Java’s.

```
bedroom: Room
   {
     roomName = “bedroom”;
      desc = "This is a luxury room that you found inside the cave, with a full bathroom
         and a nice view of the Caribbean.”;
     east = terrifyingCave; /* Where the player will go if they type ‘east’ from here. */
   }
```

Items follow a similar format, but can have methods that describe actions related to the item:

```
ring: Thing
   {
     name = “ring”;
     location = bedroom;        /*This is where the ring will be found. */
     desc = "It's a small silver ring with a a black flower";

        Fun wear ()
         {
           “You put on the silver ring and instantly feel like a king. ";
         }
    }
```

If the player meets certain conditions, the writer can finish the game by calling a function:

```
finishGameMessage (“Sorry! The giant squid ate the ring and the game is over.”);
```

This function will display the ending message, wait for the player to press a key, and finish the game.

Standard libraries

Even though the programmer is responsible for any programming that is relevant to his or her story, the programming language already provides some basic movement and action commands ready for the player:

- look around
- examine <name of item>
- show Inventory
- Basic direction movements (go north, south, east, west, northwest, northeast, southwest, or southeast and their abbreviations).
- Error messages when the actions are not recognized

These commands are explained in detail in the language tutorial.

The game will keep track of other elements like the current location of the player and their inventory contents.

2. Language Tutorial

2.1 For the programmer

a) *Hello, world!*

In order to get something displayed on the screen, the programmer only needs to write the introduction of the game. Here is an example:

```
myGame : GameMain
{
    initialRoom = myRoom;
    intro = “Hello, world!”;
}
```

Even though this would print out “Hello world!” , the game will then wait for the player to type commands. Since the previous program did not define how the game ends, it would never finish.

The story must have a beginning and end. An example of a complete program with the minimum components would include defining a game, a room, an item, and an action performed on that item, finishing the game. Here is an example:

```
myRoom : Room
{
    name = “My room”;
    desc = “This is my room. It’s locked, but it looks like somebody dropped a key.”;
}

key : Thing
{
  name = “key”;
  desc = “A small, golden key.”;
  location = myRoom;

  Fun use ()
    {
    finishGameMessage (“You open the door. It leads to a dungeon. Good job!”
    }

}

myGame: GameMain
{
    initialRoom = myRoom;
    intro = “You wake up in your room. There’s a small key on the floor.”;
}
```

In order to finish this game, the player would have to type “take key” to pick up the key and put it in their inventory. Once they have the key, they can type “use key”, which will finish the game.

b) More complex programs

As shown above, there are three types of objects. Each of them has some required fields:

- GameMain: Every game must have one object, and only one, of this type.
  This type has the two following required fields:
    - initialRoom: This object tells the game the starting location of the player. The Room object that this field refers to must have been previously defined.
    - intro: This is the introduction of the game and will be the first thing shown to the player.
- Room: Every game must have at least one object of this type, that is, the room where the player starts.
  This type has two required fields:
    - name: The name of the room.
    - desc: The description of the room.

  This type also has optional fields that represent exits from this room to different rooms. The field name must be a compass direction, and the destination must be an existing room. The programmer can define up to eight exits in a room, which must match one of the compass directions. In order to take that exit, the player must type "go north", "go south", etc.

  The format is as follows:

```
myRoom : Room
{
    name = "My room";
    desc = "This is my room.";
    north = bathroom;
    southeast = kitchen;
}
```

  In this example, objects called bathroom and kitchen must have been declared before myRoom. These exit fields are optional, so the writer must keep in mind that the player will not be able to go anywhere if no exits are defined. However, they will still be able to interact with items and use them.

- Thing: Every game must have at least one object of this type, that is, the item that triggers the end of the game when the correct action is performed on it. Players can place items their inventory to interact with them.
  This type has the following required fields:
    - name: The name of the item, which must match the object name.
    - desc: The description of the item.
    - location: Where the item can be found. Items are always first located in a room, not in the player's inventory.

  Optionally, the writer can create an action for the item. Actions are expressed by writing a function whose name is the name of the action. This name is usually a verb and will precede the item it is attached to. For example, if we are defining a 'play' action for an item called 'flute', the function should also be called 'play'. That way, when the player types 'play flute', the related function will be automatically executed.

The format is as follows:

```
Fun use ()
    {
    “You put on the ring and immediately feel like a king.”
    }
```

All function names must be unique throughout the program and written in lowercase letters.

The writer also has a tool at their disposal to see if a particular item is in the player’s inventory and take action based on the result. Here is a short example, where a player must have the lamp oil if they want to use the lamp:

```
oil : Thing
{
  name = “Oil”;
  desc = “Some lamp oil. This could come in handy!”;
  location = field;
}

lamp : Thing
{
  name = “lamp”;
  desc = “A simple but brand new lamp. Why don't you try using it?”;
  location = cave;

Fun use ()
  {
    if (IsinInventory (oil))

       finishGameMessage (“The cave lights up and you can rest here for the night.
       Good job!”)
   else
       finishGameMessage (“You don't have any oil! It looks like you got lost in the
       cave. Oh, no!”)
  }
}
```

The writer also has other common programming tools at their disposal, like for and while loops. Please check the language reference manual (section 2 of this document) for more details.

The program starts at the object defined with type GameMainDef. It first displays the introduction listed in that object and then sets the current location based on its InitialRoom attribute. Then, it displays the initial room’s description and waits for the player’s input, taking action depending on the player’s commands.

It is also possible to write comments by using a similar syntax to the C language: comments start with /* and finish with */.

2.2 For the player

All commands are composed of two words, one verb and one noun. The player can type the following general commands:

- `look around`, to see the description of the current room.
- `show inventory`, to see the current contents of their inventory.
- `take <name_of_item>`, to place that item in their inventory.
- `examine <name_of_item>`, to see the description of an item that is in the player's inventory.
- `go <compass_direction>`, where <compass_direction> can be the full word (north, south, etc.) or its abbreviation (n, s, etc.)

Also, the player might be able to type different commands depending on what items they have and the current situation of the game. Usually, the writer will give them hints throughout the game about what actions might be available.

If a player wants to leave the game before completing it, they can type 'exit now' to quit the game. The progress of the game will be lost, so the player will have to start over if they decide to quit the game early.

3. Language Reference Manual

3.1 Introduction

Interactive Fiction Language (IFL) is a programming language that allows writers to program text-based adventure games. The writer will have the ability to control locations, an inventory, and a set of vocabulary commands to present the story to the player. The player, in turn, will become the main character of the game and will type commands in order to progress through the game and advance the story.

The language tries to be as intuitive and simple as possible, yet still tries to give the programmer enough freedom to write complex stories.

3.2 Lexical conventions

There are five different types of tokens in IFL:

- Identifiers
- Keywords
- Constants
- Operators
- Punctuation

White spaces are ignored, except for the fact that they separate tokens. Indentation is not meaningful in IFL's syntax.

Comments are also ignored; their syntax is defined below.

3.2.1 Comments

The characters /* introduce a comment, and the text that follows them is ignored until */ is found. Comments do not nest and cannot be used within string literals.

3.2.2 Identifiers

Identifiers are a sequence of letters, numbers, and the character "_". Identifiers must start with a letter and cannot contain white spaces. Uppercase and lowercase characters are considered different.

3.2.3 Keywords

The following identifiers are used as keywords and cannot be redefined by the programmer:

if
else
for
while

true
false
name
desc
room
location
thing
initialRoom
gameMainDef
isInInventory
finishGameMessage

Also, they following actions are reserved as built-in commands for the player and cannot be given any other user by the programmer:

look around
examine <name of item>
show inventory
go east
go west
go north
go south
go southwest
go southeast
go northwest
go northeast
take <name of item>

3.2.4 Constants

There are different types of constants and literals, which also represent the different types available in IFL:

a) *Integer constants*

Integer constants are a sequence of base-10 digits (0-9). Floating point numbers are not supported, since they are not typically used in text-based adventure games.

b) *String literals*

A sequence of characters surrounded with double quotes is considered a string literal. A single character is simply considered a one-character string.

c) Boolean literals

Boolean literals are represented by the keywords `true` and `false`.

3.2.5 Operators

The following operators are supported:
`! + - * / =`
`== < <= >= != && ||`

Their precedence and associativity rules, as well as their syntax, are defined in section 5 of this manual.

3.2.6 Punctuation

-
- Statements within functions end with a semicolon.
- Blocks of statements and function bodies are enclosed in curly braces.
- Nesting of functions is not allowed.

3.3 Syntax notation

Throughout the manual, grammar productions and structures are noted in *italic* style and literal words, symbols, and characters in `typewriter` style. Optional elements are enclosed in brackets.

3.4 Meaning of identifiers

3.4.1 Types

3.4.1.1 Basic types

As mentioned in section 2.4, IFL supports three basic types: integers, strings, and booleans. Conversions between types are not allowed.

3.4.1.2 Derived types

IFL supports objects specific to text-adventure games and functions that check for actions performed on those objects. Their structure and syntax is explained below.

a) Objects

Programs in IFL are composed of object definitions. There are two main types of objects, rooms (type Room) and items (type Thing), and a special type that sets the initial location and description when the game starts (GameMainDef). A quick overview is shown below; please see the language tutorial section for details about required and optional fields of each object type.

- GameMainDef: This object type is used to define the ´welcome´ message to the player and to define the initial location. An object of this type must always be defined in the game, and it must be placed after the initial room has been defined.

  ```
  identifier: GameMainDef
  {
   initialRoom = identifier;
   intro = “Introduction goes here.”;
   }
  ```

  After the introduction is displayed, the player will be able to start typing commands.

- Thing: This object type defines items. Its attributes are the item name (as shown to the player), the different words that the player can use to refer to the item separated by spaces, the description (displayed when the player types “examine *item-identifier*”), and the location where the item can be found.

  ```
  identifier: Thing
   {
      name = “This is the item name as presented to the player.”
      desc = “This is the item description.”
      location = “This is the item description.”
      [function_definition]
   }
  ```

- Room: This object type is used to define each location of the game. Its attributes are the room name (as presented to the player), its description, and the different exits.

  ```
  identifier: Room
   {
     name = “This is the room name as presented to the player.”;
     desc = “This is the room description.”;
     [exit_name = room-identifier];
     [action_definition]
  }
  ```

  Exit names must be one of the compass directions: north, south, east, west, northwest, northeast, southeast, or southwest. The programmer must create a room in order to create an exit to that room: that is, if the programmer wishes to have an exit from room2 to room1, room1 must have been created before.

  As shown above, exits are optional. The writer must keep in mind that the player will be unable to move to other rooms if they arrive to a room with no exits.

  Function (which in our context is synonymous to action) structure and syntax is defined below.

b) Functions

Functions are used to check for actions performed on items. As shown above, these functions must be placed after the attributes of an object with type Thing have been defined. Their structure is as follows:

```
fun Action ()
     { statements }
```

Statements must be separated by semicolons.
Functions are mostly used to check what type of action has been performed on a room or item. An example of this is as follows:

```
fun drink (string action)
{
      “You drink the red potion and immediately feel refreshed.”;
}
```

When the player types an action (for example, ‘play flute’), the program will check if that action is listed in the Action function. If it is, the correspondent statements will be executed. If it is not listed, the program will automatically show a message explaining that nothing happens when the player performs the action typed.

3.4.2 Variable and Function Scope

Since all definitions take place within a function, all variables are considered local. Variables are not accessible from outside the object they were defined and cannot be used before they are defined.

Variable names must be unique within the function that contains them, but there can be different objects containing variables with the same name.

3.5 Expressions and operators

3.5.1 Precedence and associativity rules

All binary operators are left-associative, unless specified. The unary operator (assignment) is right-associative.

From highest to lowest precedence, the supported operators are the following:

Operator	Description
!	Unary logical negation
% *	Multiplication and division
+ -	Addition and subtraction
== < <= > >= !=	Relational operators
&& ||	Logical operators
=	Assignment

3.5.2 Unary operators

- *! boolean-expression*

This unary operator evaluates the boolean expression and returns the opposite value (true if the boolean expression is false, false if the boolean expression is true).

3.5.3 Binary arithmetic operators

- *int-expression / int-expression*

This operator returns the division of the two arguments, which must be integers.

- *int-expression * int-expression*

This operator returns the multiplication of the two arguments, which must be integers.

- *int-expression - int-expression*

This operator indicates subtraction. The two arguments must be integers.

- *int-expression + int-expression*

This operator indicates addition. The two arguments must be integers.

3.5.4 Relational operators

These operators evaluate the comparisons and return the result (true or false).

- *expression == expression*

This operator evaluates whether the two expressions are equal. The expressions can be integer, strings, or Booleans.

- *int-expression < int-expression*

The < operator evaluates if the first expression is less than the second one. The arguments must be integers.

- *int-expression <= int-expression*

The <= operator evaluates if the first expression is less or equal than the second one. The arguments must be integers.

- *int-expression > int-expression*

The > operator evaluates if the first expression is greater than the second one. The arguments must be integers.

- *int-expression >= int-expression*

The >= operator evaluates if the first expression is greater or equal than the second one. The arguments must be integers.

- *expression != expression*

The != operator evaluates if the two expressions are different. The expressions can be integer, strings, or booleans.

3.5.5 Logical operators

- *boolean-expression && Boolean-expression*

This operator evaluates if both boolean expressions are true.

- *boolean-expression || Boolean-expression*

This operator evaluates whether at least of the two expressions is true.

3.5.6 Assigment operator

- *identifier = expression*

This operator declares a variable with the same name as the identifier. Then, it evaluates the expression and assigns its result and type to the variable.

3.6 Statements

Statements are present within function bodies.

3.6.1 Expression statements

Expressions can be created using the operators listed in section 5, or they can also be function calls as explained in section 4.1.2.

3.6.2 Compound statements

Multiple statements are executed in the same order as they appear within a function. They must be separated with a semicolon.

3.6.3 Conditional statements

Conditional statements take one of the two following forms:

`If` (*boolean-expression) {statements} [*`else` *{statements}];*

If the boolean expression evaluates to true, the first set of statements is executed. Otherwise (and if the `else` statement is used), the second set of statements is executed.

3.6.4 For Statement

The `for` statement has the following syntax:

`For (`*loop-identifier* = *initial-expression, loop-identifier* = *final-expression, step-expression*`)`
  { *statements* }

In the first iteration of the loop, the initial expression is evaluated and its value is assigned to loop-identifier. Then, its value is compared to final-expression. If they are equal, the loop will stop. Otherwise, the statements enclosed in curly brackets will be executed. Then, step-expression will be applied and the comparison will take place again. This process will continue until loop-identifier reaches the final-expression value. As usual, the statements enclosed in curly brackets must finish with a semicolon.

3.6.5 While Statement

The while statement has the following syntax:

`While (`*boolean-expression*`)`
    { *statements* }

This statement will evaluate boolean-expression. If it is true, the statements in curly brackets will be executed. This process will be repeated as long as the boolean expression is true. Once the Boolean-expression is false, execution will continue with the statement following the ending bracket. As usual, the statements enclosed in curly brackets must finish with a semicolon.

3.7 Built-in Functions

a) `FinishGameMessage` (*string-literal*): The game will finish when the writer calls the FinishGameMessage function. This function will display the message sent as argument on the screen, ask the player to press a key, and finish execution.
b) `isInInventory` (*item-name*): The writer can check if a certain item is currently in the player's inventory when programming actions. Actions can only be performed in items

currently in the inventory, so the programmer can assume that the item containing the action currently being programmed is also in the inventory.

c) The basic commands to navigate the game are already programmed for the writer and can be used by the player during runtime, for example:

   - look around/examine <name of item>: The player will be able to see the current room's description, as well as the description for any items in their inventory.
   - show inventory: This command will display all items currently being carried by the player.
   - Error messages for actions not recognized.
   - take command: Using the take command followed by an item that the player is not currently carrying (but is present in in their current location) will place the item in their inventory.

d) The game will also keep track of the player's current location, inventory.

3.8 Program structure

Programs written in IFL are composed of the GameMainDef object definition and a series of object (rooms and items) declarations.

These programs do not have a beginning or end, other than using the GameMainDef object to declare the initial message and location of the game. Even though these programs have procedural sections, they are considered "declarative", that is, they are simply composed of definitions of objects. The reason for this type of structure is that the player is in control of the game at all times. After the starting room is presented, the player will decide where the program will continue by performing actions on the rooms and items.

In order to end the game, the writer can call the `FinishGameMessage` function.

4. Project Plan

4.1 Process

The project was divided in three main steps, according to the structure of the files and the programming languages used:

1) Creation of the project proposal, followed by the scanner, parser, IFL abstract syntax tree and pretty-printer;
2) Creation of the language reference manual followed by the C-code interpreter (also programmed in C);
3) Programming of the C abstract syntax tree, pretty-printer, and AST-to-AST converter (all of them programmed in OCaml), as well as creation of the final report.

Even though the natural flow of the language would be switching steps 2) and 3), it became necessary to know what type of C code the interpreter needed from the AST-to-AST program. For this reason, all the features of the interpreter were created with sample C programs. After knowing exactly what code was necessary, the C abstract syntax tree, pretty printer and AST-to-AST code were created keeping those exact needs in mind.

4.2 Style guide

For C, common rules typically used for programming languages have been followed for clarity (indentation, spaces, comments, .etc). When programming the OCaml files, the general guidelines provided by Inria have been followed. These general guidelines can be found in the following URL:

http://caml.inria.fr/resources/doc/guides/guidelines.en.html

Since the IFL syntax is similar to C structs or Java classes, using the same conventions for indentation, naming of variables, spaces, comments, etc. is recommended when writing and formatting text. If formatted in a clear and readable way, it will be intuitive and easy for a programmer to figure out what the IFL code is doing, even if they have never written an IFL program before.

4.3 Project timeline

The table below shows the planned project timeline, which contains the main milestones:

Date	Task
6/7/2013	Project proposal submitted
6/20/2013	Scanner, parser, AST, and pretty printer completed, all ambiguities resolved
6/28/2013	Language reference manual submitted
7/6/2013	"Hello, World" support for C interpreter using sample programs
7/19/2013	Support for all features of C interpreter except functions and variables with testing
7/28/2013	Support for all features of C interpreter with testing
8/5/2013	AST for C code, pretty printer, and AST-to-AST converter function
8/11/2013	Integration of OCaml and C code with testing
8/16/2013	Final report submitted

4.4 Software Development Environment

The Ocaml code was developed using Inria's compiler for Windows and CygWin. For developing C code, MinGW (a minimalist GNU for Windows) was used, which includes a port of the GNU Compiler Collection (GCC). All code was written on Notepad 2, and all commands were entered through the command line on a Windows Vista (64-bit) machine.

4.5 Project log

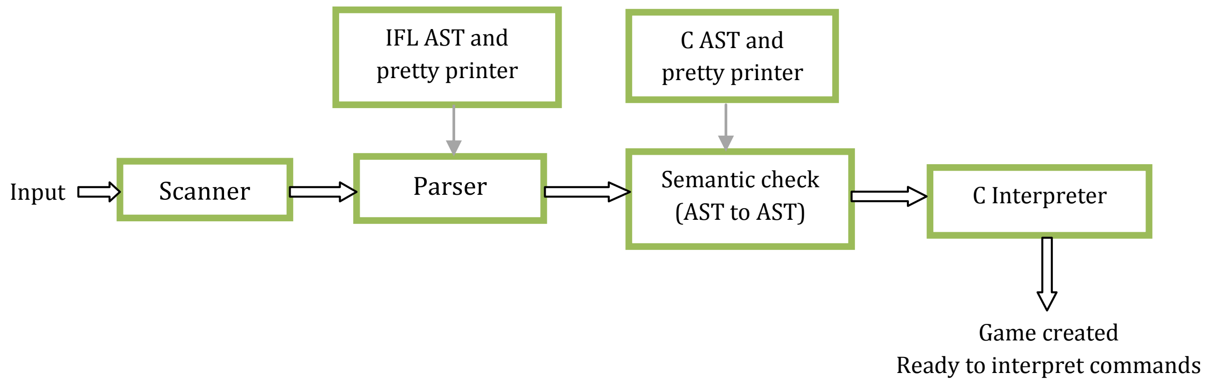
Below are the project dates on the different milestones achieved:

Date	Task
6/3/2013	Research of current interactive fiction programming languages completed
6/7/2013	Project proposal submitted
6/10/2013	Initial grammar specified
6/13/2013	Scanner created
6/20/2013	Parser and AST completed, all ambiguities resolved
6/25/2013	Pretty printer created for AST
6/28/2013	Language reference manual submitted
7/4/2013	Research for functions needed on C interpreter completed
7/6/2013	"Hello, World" support for C interpreter using sample programs
7/14/2013	Support for different types of objects in C interpreter with testing
7/19/2013	Support for exits and inventory in C interpreter with testing
7/24/2013	Support for functions within objects in C interpreter with testing
7/28/2013	Support for If, While, For, and variables within functions in C interpreter with testing
8/2/2013	Support for built-in functions in C interpreter with testing
8/9/2013	AST for C code created
8/11/2013	Pretty printer for C AST created
8/15/2013	AST-to-AST converter created
8/16/2013	Final report submitted

5. Architectural design

5.1 Components of the translator

The main components of IFL are shown in the following diagram:


IFL AST and pretty printer
C AST and pretty printer
Input
Scanner
Parser
Semantic check (AST to AST)
C Interpreter
Game created
Ready to interpret commands

5.2 Interfaces between the components

As shown above, IFL has several components:

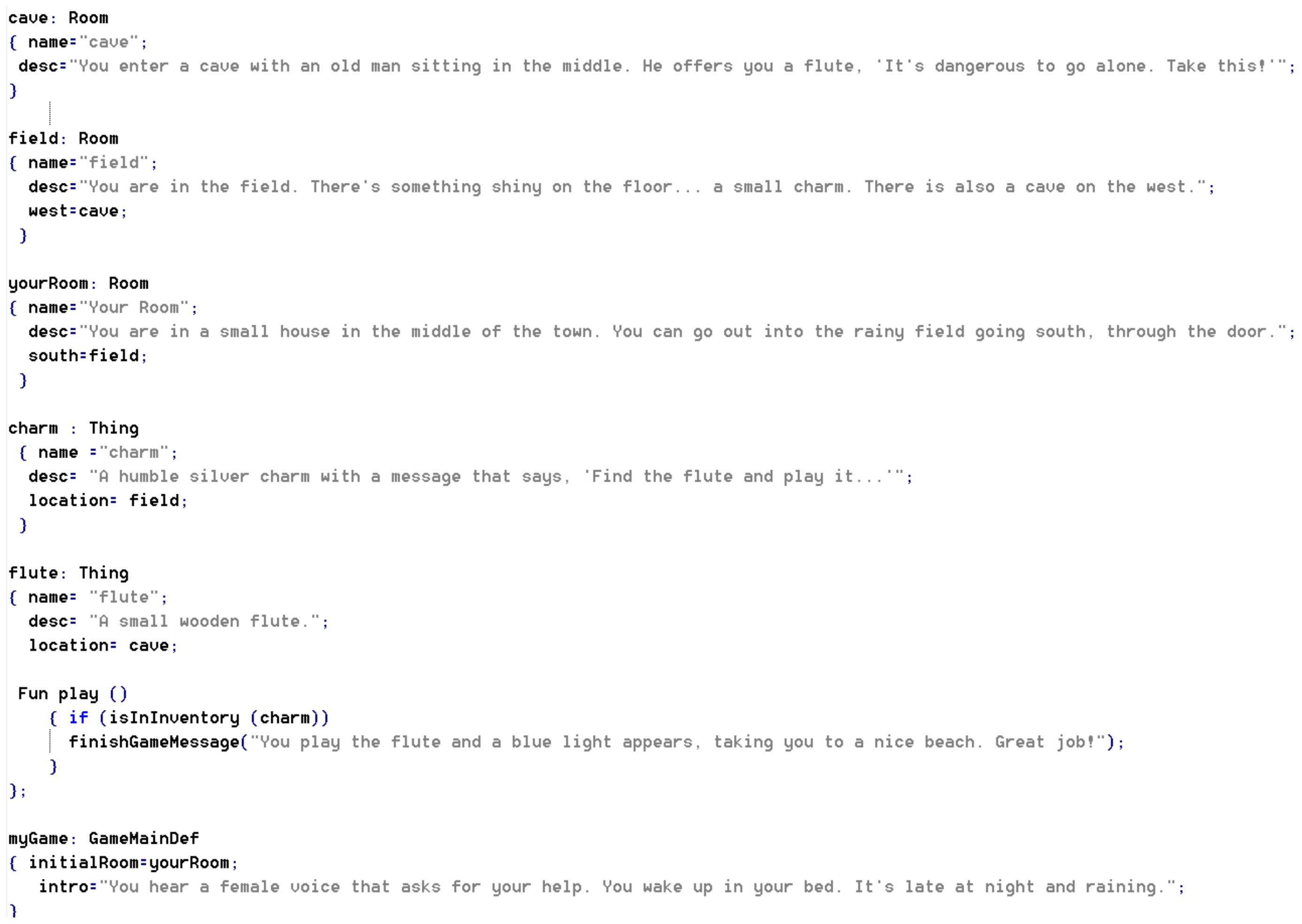
- <u>Scanner</u>: First, the programmer must write a file with extension .ifl that contains the items, rooms, and actions they want to create in the game. This code that is treated by the scanner, which generates tokens. The scanner recognizes tokens specified in scanner.mll. In order to generate scanner.ml, ocamllex is used.
- <u>Parser</u>: The parser takes the tokens as an input and generates an abstract syntax tree. The code is stored in parser.mly, and the AST types defined in ast.ml. Ocamlyacc is used to generate the abstract syntax tree.
- <u>Semantic check</u>: After the abstract syntax tree is generated, the code is converted to C code using ASTtoAST.ml. The C types are stored in CAst.ml, along with its pretty printer.
- <u>Interpreter</u>: Once the C code is generated containing the items, rooms, and actions that compose the game (that is, equivalent code to the original .ifl file but written in C code), the interpreter is ready to add these components to its environment and create the game. The interpreter will also read the commands typed by the player and react based on the world created by the IFL programmer. These features are provided and handled by IFL so that the programmer can focus on designing the components of their story. The interpreter code can be found in interpreter.c, which also uses two header files: types.h and headers.h.

6. Test plan

6.1 Representative programs

Instead of showing a couple of small programs, I am including one complete program that shows the main and most representative components of IFL. This program includes several items, rooms, exits, an action for the items, and calls to the functions already provided by the language (a call to `isInInventory` to check if an item is in the inventory before using another one, and a call to `finishGameMessage`).

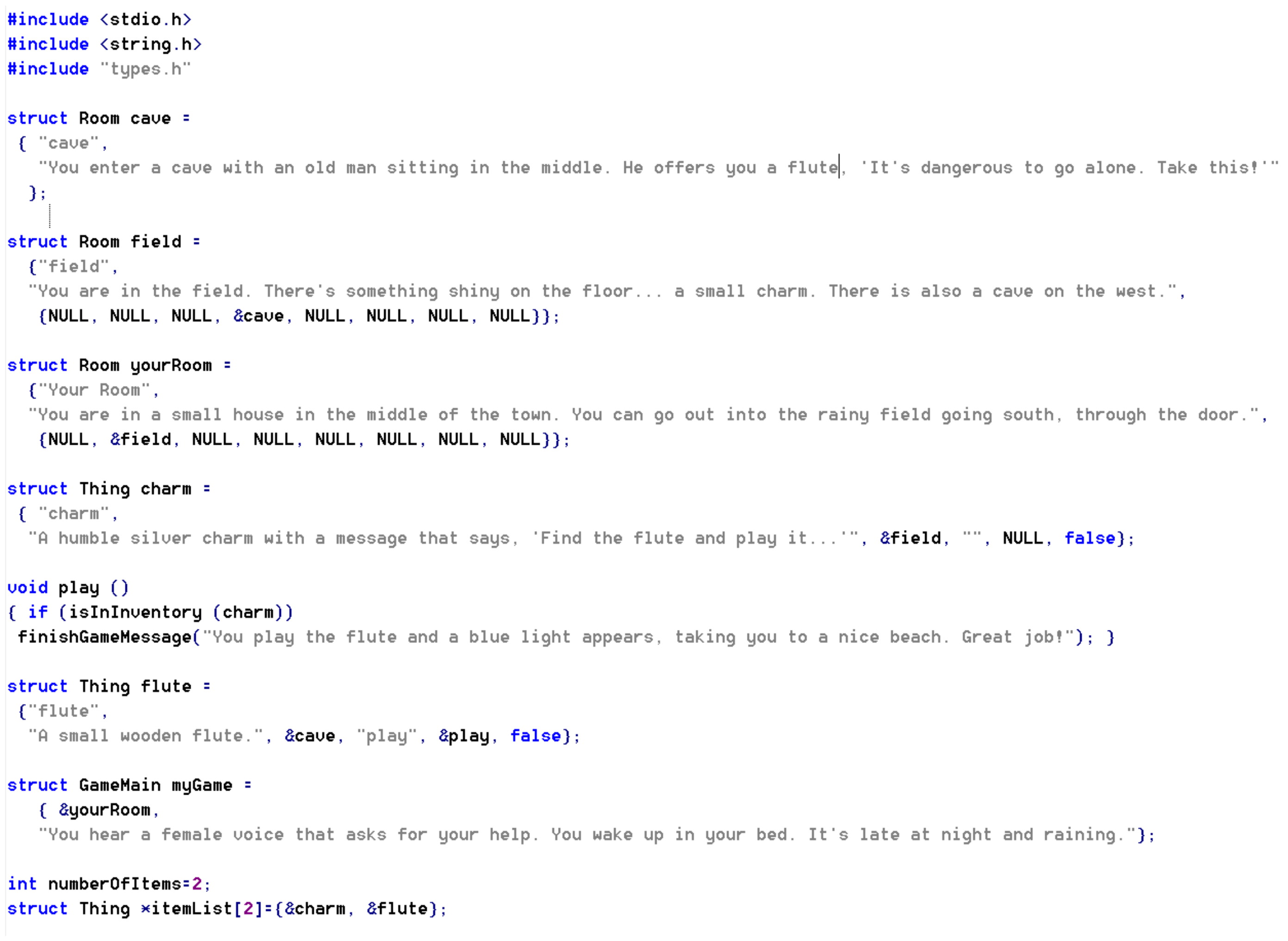
First, we see the IFL code for this program:



```
cave: Room
{ name="cave";
 desc="You enter a cave with an old man sitting in the middle. He offers you a flute, 'It's dangerous to go alone. Take this!'";
}

field: Room
{ name="field";
  desc="You are in the field. There's something shiny on the floor... a small charm. There is also a cave on the west.";
  west=cave;
 }

yourRoom: Room
{ name="Your Room";
  desc="You are in a small house in the middle of the town. You can go out into the rainy field going south, through the door.";
  south=field;
 }

charm : Thing
 { name ="charm";
  desc= "A humble silver charm with a message that says, 'Find the flute and play it...'";
  location= field;
 }

flute: Thing
{ name= "flute";
  desc= "A small wooden flute.";
  location= cave;

 Fun play ()
    { if (isInInventory (charm))
      finishGameMessage("You play the flute and a blue light appears, taking you to a nice beach. Great job!");
    }
};

myGame: GameMainDef
{ initialRoom=yourRoom;
   intro="You hear a female voice that asks for your help. You wake up in your bed. It's late at night and raining.";
}
```

This code is pretty intuitive: several rooms and items are created, with an action (play) attached to an item (flute). In this particular game, it is only possible to play the flute and finish the game if the charm has been found before.

The equivalent C code is shown below (please note that this was one of the sample programs used to test the interpreter):



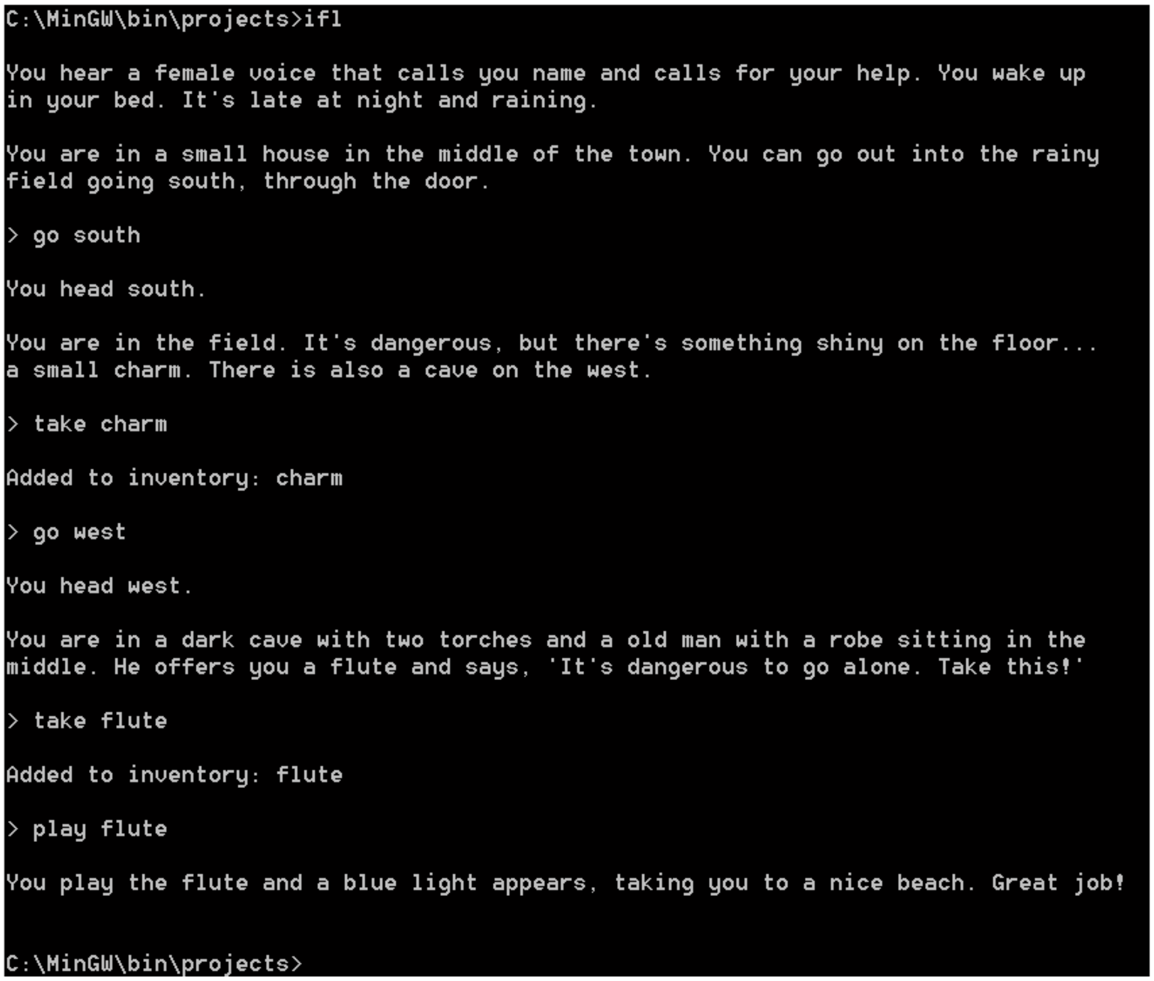
```
#include <stdio.h>
#include <string.h>
#include "types.h"

struct Room cave =
 { "cave",
   "You enter a cave with an old man sitting in the middle. He offers you a flute, 'It's dangerous to go alone. Take this!'"
  };

struct Room field =
  {"field",
  "You are in the field. There's something shiny on the floor... a small charm. There is also a cave on the west.",
   {NULL, NULL, NULL, &cave, NULL, NULL, NULL, NULL}};

struct Room yourRoom =
  {"Your Room",
  "You are in a small house in the middle of the town. You can go out into the rainy field going south, through the door.",
   {NULL, &field, NULL, NULL, NULL, NULL, NULL, NULL}};

struct Thing charm =
 { "charm",
  "A humble silver charm with a message that says, 'Find the flute and play it...'", &field, "", NULL, false};

void play ()
{ if (isInInventory (charm))
 finishGameMessage("You play the flute and a blue light appears, taking you to a nice beach. Great job!"); }

struct Thing flute =
 {"flute",
  "A small wooden flute.", &cave, "play", &play, false};

struct GameMain myGame =
   { &yourRoom,
   "You hear a female voice that asks for your help. You wake up in your bed. It's late at night and raining."};

int numberOfItems=2;
struct Thing *itemList[2]={&charm, &flute};
```

The fields used to initialize rooms are the name, description, and any exits to other rooms. If there are any exits, the object is initialized with a list of pointers for each compass direction. These point to the room each individual exit is linked to, or NULL if there is not an exit in that direction.

The fields used to initialize Thing objects are the name, description, pointer to the room where the item is located, a string with the name of its action, a pointer to the action function, and "false" to indicate that the item is not in the inventory. As seen in the original IFL code, some of these fields are not created by the player; they should be generated by the code converter in order to give the interpreter all the information it needs.

Also, as seen at the bottom of the last listing and for the interpreter's convenience, the number of items is counted and an array with pointers to each of those items is created. This is very useful for the interpreter in order to find out if an item exists and to locate it in memory.

Once compiled along with the C interpreter and executed, the game can be played. Below is a screen capture of the game generated by the interpreted based on the above code, and the commands entered to progress and complete the game.



```
C:\MinGW\bin\projects>ifl

You hear a female voice that calls you name and calls for your help. You wake up
in your bed. It's late at night and raining.

You are in a small house in the middle of the town. You can go out into the rainy
field going south, through the door.

> go south

You head south.

You are in the field. It's dangerous, but there's something shiny on the floor...
a small charm. There is also a cave on the west.

> take charm

Added to inventory: charm

> go west

You head west.

You are in a dark cave with two torches and a old man with a robe sitting in the
middle. He offers you a flute and says, 'It's dangerous to go alone. Take this!'

> take flute

Added to inventory: flute

> play flute

You play the flute and a blue light appears, taking you to a nice beach. Great job!


C:\MinGW\bin\projects>
```

6.2. Test suites used

Below is a list of the test suites prepared. The file names listed are the C file that the interpreter will use and its equivalent IFL file.

For clarity, the header C files have not been listed in the table, even though they would change depending on the available objects and functions.

Filenames	Test type	Interpreter status
01 helloworld.c 01 helloworld.ifl	Basic program that displays the game introduction on the screen. Since the initial location of the player must be set, one room is created.	At this point, the introduction can be displayed but it's not possible to accept player commands yet.
02 Two Rooms and Exit.c 02 Two Rooms and Exit.ifl	Basic program with two rooms and an exit that connects them.	* The program accepts compass directions from the player. * 'look around' command is available. * Basic error messages are added when a player tries to use a command that is not available.
03 Item.c 03 Item.ifl	Basic program with one room and one item.	* The 'take <name of item>' command becomes available. * Error messages are added when a player wants to take an item that is not available. * Inventory is added. * 'examine <name of item>'command becomes available. * Error messages are added when a player wants to examine an item that is not in their inventory. * 'show inventory' command becomes available. * Behavior of empty inventory tested; message is shown to the player in this situation.
04 basic_game_complete.c 04 basic_game_complete.ifl	Program with all the basic elements: A starting room, one item, and one action that will finish the game when used. This test was used when functions were added.	* Interpreter is able to handle all features listed above plus handling of functions. * Error messages are displayed when a player tries to use an action that is not available for a certain item. * FinishGameMessage() is added and can be used by the programmer to finish the game.
05 game_and_inventory.c 05 game_and_inventory.ifl	Program will all the basic elements mentioned above, but including a slightly more complex function that checks whether an item is in the inventory (on top of the item being currently used) and takes action depending on the result.	* Program is able to handle more complex functions. * isInInventory() is added and can be used by the programmer.

These particular tests were chosen because they cover all the critical parts of the game, as seen above.

In addition, the interpreter was tested using player commands. The commands tested depend on the feature added in each step and simply follow the table shown above. For example, during step 3 on the table (having the basic program and one item as the test case), commands were tested based on the features listed in the right column:

- 'take <name of item>' would be tested for the existing item.
- 'take <name of item>' would be tested for a non-existing item, verifying the error message.
- 'show inventory' would be tested with an empty inventory.
- 'show inventory' would be tested after having picked one item (later on, this would be tested with several items).
- 'examine <name of item>' would be tested for a non-existing item, verifying the error message.
- 'examine <name of item>' would be tested for an existing item that is currently in the inventory.
- 'examine <name of item>' would be tested for an existing item that is currently not in the inventory.

Similar tests would be performed for each of the steps and functions mentioned above.

7. Lessons Learned

The most important lesson learned from the project is how powerful OCaml is, and how simple it can be to write a complete scanner, parser and abstract syntax tree.

A few months ago, I worked on a personal project that required the same type of work. It was not for a programming language; my goal was to extract information for art pieces (name, description, filename, tags, etc.) from a static HTML website and transfer that information to a SQL database in order to create a dynamic website.

Using Java, I had the program scan through every character in the HTML files to find the right HTML tokens and extract the fields I needed. Knowing there were many different HTML templates used and over 100 files total, this was not an easy task. The amount of time and effort invested on that project would have been reduced dramatically if I had been aware of OCaml's features.

Regression testing was very useful. Starting with a simple "Hello, World" program and testing features in iterations helped me verify that the program continued to be functional, while ensuring that no errors had been introduced by adding a change or a new feature. This process also made version control very easy, since the files could be archived after each iteration.

Having a close-to-final parser, scanner, and AST for my language, as requested by the professor, was key in order to have a solid language reference manual close to the beginning of the project. While small changes were necessary to streamline the interpreter portion of the project, these were just minimal syntactical changes.

Even though I am used to having to learn new programming languages and produce results fast, dealing with Ocaml errors was the main challenge I found. Some of these errors took days to resolve and delayed the general progress of the project significantly. The learning curve for some OCaml features was much steeper than I initially expected, so I was only able to get a quick working knowledge of them. I would like to deepen my understanding of these features in the future.

Before this class, my knowledge of a compiler was very vague. While being very challenging at times because of having to balance work and school during the short summer term, I have found that creating my own language has been a fun, interesting, and rewarding learning experience. Because of my personal interest in games, I plan on continuing to work on text-based gaming programming languages projects.

8. Appendix: Interactive Fiction Language Code

Scanner (scanner.mll)

```
{ open Parser }

rule token = parse
  [' ' '\t' '\r' '\n'] { token lexbuf } (* Whitespace *)
| "/*"              { comment lexbuf }     (* Comments *)
| '('               { LPAREN }
| ')'               { RPAREN }
| '{'               { LBRACE }
| '}'               { RBRACE }
| ';'               { SEMI }
| ':'               { COLON }
| ','               { COMMA }
| '.'               { PERIOD }
| '+'               { PLUS }
| '-'               { MINUS }
| '*'               { TIMES }
| '/'               { DIVIDE }
| '='               { ASSIGN }
| ":="              { DECLARE }
| '!'               { NOT }
| "=="              { EQ }
| "!="              { NEQ }
| '<'               { LT }
| "<="              { LEQ }
| '>'               { GT }
| ">="              { GEQ }
| "if"              { IF }
| "else"            { ELSE }
| "for"             { FOR }
| "while"           { WHILE }
| "bool"            { BOOL }
| "fun"             { FUN }
| "true"            { BOOLEANLITERAL(true) }
| "false"           { BOOLEANLITERAL(false) }
| "int"             { INT }
| "bool"            { BOOL }
| "string"          { STRING }
| "int"             { ROOM }
| "bool"            { THING }
| "string"          { GAMEMAINDEF }
| "&&"              { AND }
| "||"              { OR }
| '\"'[^'"']*'\"' as lxm
            { STRINGLITERAL(String.sub lxm 1 ((String.length lxm) - 2)) }
| ['0'-'9']+ as lxm { LITERAL(int_of_string lxm) }
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm { ID(lxm) }
| eof { EOF }
| _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }

and comment = parse
  "*/" { token lexbuf }
| _    { comment lexbuf }
```

Parser (parser.mly)

```
%{ open Ast %}
/* Punctuation */
%token SEMI COLON LPAREN RPAREN LBRACE RBRACE COMMA QUOTE
/* Operators */
%token AND OR NOT PLUS MINUS TIMES DIVIDE ASSIGN DECLARE
/* Comparators */
%token EQ NEQ LT LEQ GT GEQ
/* Keywords */
%token IF ELSE FOR WHILE INT BOOL STRING ROOM THING GAMEMAINDEF FUN PERIOD

%token <int> LITERAL
%token <bool> BOOLEANLITERAL
%token <string> STRINGLITERAL
%token <string> ID
%token EOF

%nonassoc NOELSE
%nonassoc ELSE
%right ASSIGN DECLARE
%left AND OR
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS
%right NOT
%left TIMES DIVIDE

%start program
%type <Ast.program> program

%%

program:
  /* nothing */ { [] }
  | program odecl { List.rev $1 }

odecl:
 ID COLON obtype LBRACE att_list fdecl_list RBRACE
     { { object_name = $1;
     object_type = $3;
     attrib = List.rev $5;
     fundecl = $6 } }

att_list:
    /* nothing */    { [] }
   | att_list att { $2 :: $1 }

att:
   ID ASSIGN expr SEMI {
          { attname = $1;
            attvalue = $3;
          } }

fdecl_list:
   /* nothing */ { [] }
   | fdecl_list fdecl { $2 :: $1 }

 fdecl:
  FUN ID LPAREN RPAREN LBRACE vdecl_list stmt_list RBRACE
     { { fname = $2;
     locals = List.rev $6;
     body = List.rev $7} }


primitive:
    BOOL { Bool }
  | INT { Int }
  | STRING { String }

obtype:
    ROOM { Room }
  | THING { Thing }
  | GAMEMAINDEF { GameMainDef }

 vdecl_list:
    /* nothing */    { [] }
  | vdecl_list vdecl { $2 :: $1 }

vdecl:
   primitive ID DECLARE expr SEMI {
   {
     vtype = $1;
     vname = $2;
     vvalue = $4;
    }}

stmt_list:
    /* nothing */  { [] }
  | stmt_list stmt { $2 :: $1 }

stmt:
   expr SEMI { Expr($1) }
  | LBRACE stmt_list RBRACE { Block(List.rev $2) }
  | IF LPAREN expr RPAREN stmt %prec NOELSE { If($3, $5, Block([])) }
  | IF LPAREN expr RPAREN stmt ELSE stmt    { If($3, $5, $7) }
  | FOR LPAREN expr_opt SEMI expr_opt SEMI expr_opt RPAREN stmt
     { For($3, $5, $7, $9) }
  | WHILE LPAREN expr RPAREN stmt { While($3, $5) }

expr_opt:
    /* nothing */ { Noexpr }
  | expr          { $1 }

expr:
    LITERAL          { Literal($1) }
  | STRINGLITERAL    { StringLiteral($1)}
  | BOOLEANLITERAL   { BoolLiteral($1) }
  | ID               { Id($1) }
  | NOT expr         { Unaryop(Not, $2) }
  | expr PLUS   expr { Binop($1, Add,   $3) }
  | expr MINUS  expr { Binop($1, Sub,   $3) }
  | expr TIMES  expr { Binop($1, Mult,  $3) }
  | expr DIVIDE expr { Binop($1, Div,   $3) }
  | expr EQ     expr { Binop($1, Equal, $3) }
  | expr NEQ    expr { Binop($1, Neq,   $3) }
  | expr LT     expr { Binop($1, Less,  $3) }
  | expr LEQ    expr { Binop($1, Leq,   $3) }
  | expr GT     expr { Binop($1, Greater,  $3) }
  | expr GEQ    expr { Binop($1, Geq,   $3) }
  | expr AND    expr { Binop($1, And, $3) }
  | expr OR     expr { Binop($1, Or, $3) }
  | ID ASSIGN expr   { Assign($1, $3) }
  | ID PERIOD ID     { Call($1, $3) }
  | ID LPAREN actuals_opt RPAREN { Call($1, $3) }
  | LPAREN expr RPAREN { $2 }

actuals_opt:
    /* nothing */ { [] }
  | actuals_list  { List.rev $1 }

actuals_list:
    expr                    { [$1] }
  | actuals_list COMMA expr { $3 :: $1 }
```

IFL Abstract Syntax Tree and pretty printer (ast.ml)

```
type op = Add | Sub | Mult | Div | Equal | Neq | Less | Leq | Greater | Geq
        | And | Or | Not


type primitive = Int | Bool | String
type obtype = Room | Thing | GameMainDef

type expr =
    Literal of int
  | BoolLiteral of bool
  | StringLiteral of string
  | Id of string
  | Unaryop of op * expr
  | Binop of expr * op * expr
  | Assign of string * expr
  | Call of string * expr list
  | Noexpr

 type stmt =
    Block of stmt list
  | Expr of expr
  | If of expr * stmt * stmt
  | For of expr * expr * expr * stmt
  | While of expr * stmt

 type vdecl = {
   vtype : primitive;
   vname : string;
   vvalue : expr;
    }
 type fdecl = {
    fname : string;
    locals : vdecl list;
    body : stmt list;
   }

 type att = {
  attname: string;
  attvalue : string;
  }

 type odecl = {
  object_name : string;
  object_type : obtype;
  attrib : att list;
  fundecl : fdecl list;
  }

type program = odecl list

let rec string_of_type t = match t with
    Bool -> "bool"
  | Int -> "int"
  | String -> "string"

let rec string_of_obtype t = match t with
  | Room -> "Room"
  | Thing -> "Thing"
  | GameMainDef -> "GameMainDef"

let rec string_of_expr = function
    Literal(l) -> string_of_int l
  | BoolLiteral(false) -> "false"
  | BoolLiteral(true) -> "true"
  | StringLiteral(s) -> "\"" ^ String.escaped s ^ "\""
  | Id(s) -> s
  | Binop(e1, o, e2) -> "(" ^
      string_of_expr e1 ^ " " ^
      (match o with
    Add -> "+" | Sub -> "-" | Mult -> "*" | Div -> "/"
      | Equal -> "==" | Neq -> "!="
      | Less -> "<" | Leq -> "<=" | Greater -> ">" | Geq -> ">="
      | Or -> "||" | And -> "&&"
      | Not -> "!"
      | _ -> raise(Failure("illegal binop"))
      ) ^ " " ^
      string_of_expr e2 ^ ")"
  | Unaryop(o, e) -> "(" ^
      (match o with
        Not -> "!"
      | _ -> raise (Failure("illegal unary operator")))
      ^ string_of_expr e ^ ")"
  | Assign(v, e) -> v ^ " = " ^ string_of_expr e
  | Call(f, el) ->
      f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^ ")"
  | Noexpr -> ""
  let rec string_of_stmt = function
    Block(stmts) ->
      "{\n" ^ String.concat "" (List.map string_of_stmt stmts) ^ "}\n"
  | Expr(expr) -> string_of_expr expr ^ ";\n";
  | If(e, s, Block([])) -> "if (" ^ string_of_expr e ^ ")\n" ^ string_of_stmt
s
  | If(e, s1, s2) ->  "if (" ^ string_of_expr e ^ ")\n" ^
      string_of_stmt s1 ^ "else\n" ^ string_of_stmt s2
  | For(e1, e2, e3, s) ->
      "for (" ^ string_of_expr e1  ^ " ; " ^ string_of_expr e2 ^ " ; " ^
      string_of_expr e3  ^ ") " ^ string_of_stmt s
  | While(e, s) -> "while (" ^ string_of_expr e ^ ") " ^ string_of_stmt s



 let string_of_vdecl vdecl =
  string_of_type vdecl.vtype ^ " " ^ vdecl.vname ^ " = " ^ string_of_expr
vdecl.vvalue ^ ";\n"

let string_of_fdecl fdecl =
    " FUN " ^ fdecl.fname ^ "() \n { \n" ^
  String.concat "" (List.map string_of_vdecl fdecl.locals) ^
  String.concat "" (List.map string_of_stmt fdecl.body) ^
  "}\n"


let string_of_assignm att = att.attname ^ " = " ^ att.attvalue ^ ";\n"

let string_of_odecl odecl =
  odecl.object_name ^ " : " ^ string_of_obtype odecl.object_type ^ "\n{\n" ^
   String.concat "" (List.map string_of_assignm odecl.attrib) ^ "\n\n" ^
   String.concat "" (List.map string_of_fdecl odecl.fundecl) ^ "}\n"

let string_of_program (objs) =
  String.concat "\n" (List.map string_of_odecl objs)
```

C Abstract Syntax Tree and pretty printer (CAst.ml)

```
exception Error of string

type file = declaration list

and declaration =
    Decl of specifier list * declarator * initializr
  | Function of specifier list * declarator * statement
  | String of string

and declarator =
    Name of string
  | Array of declarator * expression
  | FuncDecl of declarator * parameter list
  | Pointer of declarator
  | NoDeclarator
```

```
and parameter =
    specifier list * declarator

and initializr =
    NoInitializer
  | ExprInitializer of expression
  | ArrayInitializer of expression list

and specifier =
    Type of string
  | Struct of string option * declaration list option

and statement =
    Nop
  | Expr of expression
  | Block of declaration list * statement list
  | If of expression * statement * statement
  | While of expression * statement
  | For of expression * expression * expression * statement
  | Quote of string
  | WhileSt of expression * operator * expression
  | WhileEnd of expression * operator * expression


and operator =
    Arrow | Dot |
    Deref | Addrof |
    Mult | Div |
    Plus | Minus |
    LT | LE | GT | GE |
    Equals | Noteq |
    LAnd |
    LOr |
    Assign |
    Comma


and expression =
    NoExpr
  | Literal of string
  | Id of string
  | BinOp of expression * operator * expression
  | PrefixOp of operator * expression
  | Conditional of expression * expression * expression
  | Call of expression * expression list
  | Index of expression * expression                  (* foo[bar] *)
  | Cast of specifier list * declarator * expression
  | CompoundLiteral of expression list

(* ********************************************************************** *)

let precedence_level = function
    Arrow | Dot -> 10
  | Deref | Addrof -> 9
  | Mult | Div -> 8
  | Plus | Minus -> 7
  | LT | LE | GT | GE -> 6
  | Equals | Noteq -> 5
  | LAnd -> 4
  | LOr -> 3
  | Assign -> 1
  | Comma -> 0

let is_right_associative = function
    Deref | Addrof | Assign -> true
  | _ -> false

let precedence_of_conditional = 2

let highest_precedence = 11

let string_of_operator = function
    Arrow -> "->"
  | Dot -> "."
  | Deref -> "*"
  | Addrof -> "&"
  | Mult -> " * "
  | Div -> " / "
  | Plus -> " + "
  | Minus -> " - "
  | LT -> " < "
  | LE -> " <="
  | GT -> " > "
  | GE -> " >= "
  | Equals -> " == "
  | Noteq -> " != "
  | LAnd -> " && "
  | LOr -> " || "
  | Assign -> " = "
  | Comma -> ", "

let indentation spaces =
  String.make (spaces / 8) '\t' ^
  String.make (spaces mod 8) ' '

let rec string_of_expression e =
  let (string, _) = expr e in string

and expr = function
      NoExpr -> "", highest_precedence
    | Literal(s) -> s, highest_precedence
    | Id(s) -> s, highest_precedence
  | BinOp(e1, op, e2) ->
      let (l, lp) = expr e1
      and (r, rp) = expr e2
      and p = precedence_level op
      and is_right = is_right_associative op in
      (if lp > p || (not is_right && lp = p) then l else "(" ^ l ^ ")") ^
      string_of_operator op ^
      (if rp > p || (is_right && rp = p) then r else "(" ^ r ^ ")"),
      p
    | PrefixOp(op, e) ->
      let (s, ep) = expr e
      and p = precedence_level op in
      string_of_operator op ^
      (if ep >= p then s else "(" ^ s ^ ")"), p
  | Conditional(e1, e2, e3) ->
      let (s1, p1) = expr e1
      and (s2, p2) = expr e2
      and (s3, p3) = expr e3 in
      (if p1 > precedence_of_conditional then s1 else "(" ^ s1 ^ ")") ^
      " ? " ^
      (if p2 > precedence_of_conditional then s2 else "(" ^ s2 ^ ")") ^
      " : " ^
      (if p3 > precedence_of_conditional then s3 else "(" ^ s3 ^ ")"),
      precedence_of_conditional
  | Call(e, el) ->
      let s1, _ = expr e
      in s1 ^ "(" ^ (* FIXME: precedence *)
      String.concat ", " (List.map (fun e -> fst (expr e)) el) ^ ")",
      highest_precedence
  | Index(e1, e2) ->
     let s1, p1 = expr e1
     and s2, _ = expr e2 in
    (if p1 = highest_precedence then s1 else "(" ^ s1 ^ ")") ^ "[" ^ s2 ^ "]",
      highest_precedence
  | Cast(sl, d, e) -> "((" ^ string_of_specifier_list sl ^
      string_of_declarator d ^ ") " ^ string_of_expression e ^ ")",
      highest_precedence
  | CompoundLiteral(el) ->
      "{\n    " ^
      String.concat ",\n    " (List.map string_of_expression el) ^
      "\n    }", highest_precedence

  and string_of_specifier specifier =
  let aggregate_body s d =
      (match s with
    None -> ""
      | Some(s) -> " " ^ s) ^
      (match d with
    None -> ""
      | Some(d) -> " {\n  " ^ (String.concat "\n  "
                     (List.map string_of_declaration d)) ^ "\n}")
  in
  match specifier with
    Type(s) -> s
  | Struct(s, d) ->
      "struct" ^ aggregate_body s d

  and string_of_specifier_list sl =
  List.fold_left (fun s sp -> s ^ string_of_specifier sp ^ " ") "" sl

  and string_of_declarator declarator =

  let rec prefix = function
      Name(s) -> s
    | Array(d, _) -> prefix d
    | FuncDecl(d, _) -> prefix d
    | Pointer(d) -> "(*" ^ prefix d
    | NoDeclarator -> ""
  in
```

```
 let rec suffix = function
     Name(_) -> ""
   | Array(d, e) ->
   "[" ^ string_of_expression e ^ "]" ^ suffix d
   | FuncDecl(d, pl) ->
   suffix d ^
   "(" ^ String.concat ", " (List.map string_of_parameter pl) ^ ")"
   | Pointer(d) -> ")" ^ suffix d
   | NoDeclarator -> ""

 in
 prefix declarator ^ suffix declarator

and string_of_parameter (sl, decl) =
 (string_of_specifier_list sl) ^ (string_of_declarator decl)

and string_of_statement ind = function
   Nop -> (indentation ind) ^ ";"
 | Expr(e) -> (indentation ind) ^ (string_of_expression e) ^ ";"
 | Block(dl, sl) -> indentation ind ^ "{\n" ^
     List.fold_left (fun s d -> s ^ indentation (ind + 2) ^
   string_of_declaration d ^ "\n") "" dl ^
     List.fold_left (fun s st -> s ^
   string_of_statement (ind + 2) st ^ "\n") "" sl ^
     indentation ind ^ "}"
 | If(e, Block(dl, sl), Nop) ->
     (indentation ind) ^ "if (" ^ (string_of_expression e) ^ ") {\n" ^
     List.fold_left (fun s d -> s ^ indentation (ind + 2) ^
   string_of_declaration d ^ "\n") "" dl ^
     List.fold_left (fun s st -> s ^
   string_of_statement (ind + 2) st ^ "\n") "" sl ^
     indentation ind ^ "}"
 | If(e, Block(dl1, sl1), Block(dl2, sl2)) ->
     (indentation ind) ^ "if (" ^ (string_of_expression e) ^ ") {\n" ^
     List.fold_left (fun s d -> s ^ indentation (ind + 2) ^
   string_of_declaration d ^ "\n") "" dl1 ^
     List.fold_left (fun s st -> s ^
   string_of_statement (ind + 2) st ^ "\n") "" sl1 ^
     indentation ind ^ "} else {" ^
     List.fold_left (fun s d -> s ^ indentation (ind + 2) ^
   string_of_declaration d ^ "\n") "" dl2 ^
     List.fold_left (fun s st -> s ^
   string_of_statement (ind + 2) st ^ "\n") "" sl2 ^
     indentation ind ^ "}"
 | If(e, Block(dl, sl), s2) ->
     (indentation ind) ^ "if (" ^ (string_of_expression e) ^ ") {\n" ^
     List.fold_left (fun s d -> s ^ indentation (ind + 2) ^
   string_of_declaration d ^ "\n") "" dl ^
     List.fold_left (fun s st -> s ^
   string_of_statement (ind + 2) st ^ "\n") "" sl ^
     indentation ind ^ "} else\n" ^
     string_of_statement (ind + 2) s2
 | If(e, s1, Block(dl, sl)) ->
     (indentation ind) ^ "if (" ^ (string_of_expression e) ^ ")\n" ^
     (string_of_statement (ind + 2) s1) ^ "\n" ^
     indentation ind ^ "else {\n" ^
```

```
    List.fold_left (fun s d -> s ^ indentation (ind + 2) ^
  string_of_declaration d ^ "\n") "" dl ^
    List.fold_left (fun s st -> s ^
  string_of_statement (ind + 2) st ^ "\n") "" sl ^
    indentation ind ^ "}"
| If(e, s1, s2) ->
    (indentation ind) ^ "if (" ^ (string_of_expression e) ^ ")\n" ^
    (string_of_statement (ind + 2) s1) ^
    (match s2 with
  Nop -> ""
    | _ -> (indentation ind) ^ "else\n" ^
    (string_of_statement (ind + 2) s2)
    )
| While(e, Block(dl, sl)) ->
    indentation ind ^ "while (" ^ string_of_expression e ^ ") {\n" ^
    List.fold_left (fun s d -> s ^ indentation (ind + 2) ^
  string_of_declaration d ^ "\n") "" dl ^
    List.fold_left (fun s st -> s ^
  string_of_statement (ind + 2) st ^ "\n") "" sl ^
    indentation ind ^ "}"
| While(e, s) ->
    indentation ind ^ "while (" ^ string_of_expression e ^ ")\n" ^
    string_of_statement (ind + 2) s
| For(e1, e2, e3, Block(dl, sl)) ->
    indentation ind ^ "for (" ^
    string_of_expression e1 ^ " ; " ^
    string_of_expression e2 ^ " ; " ^
    string_of_expression e3 ^ ") {\n" ^
    List.fold_left (fun s d -> s ^ indentation (ind + 2) ^
  string_of_declaration d ^ "\n") "" dl ^
    List.fold_left (fun s st -> s ^
  string_of_statement (ind + 2) st ^ "\n") "" sl ^
    indentation ind ^ "}"
| For(e1, e2, e3, s) ->
    indentation ind ^ "for (" ^
    string_of_expression e1 ^ " ; " ^
    string_of_expression e2 ^ " ; " ^
    string_of_expression e3 ^ ")\n" ^
    (string_of_statement (ind + 2) s)
| Quote(s) -> s
| WhileSt(opr1, op, opr2) ->
  let (l, lp) = expr opr1
      and (r, rp) = expr opr2
  in
      "while (" ^ l ^ " " ^ string_of_operator op ^ " " ^
    r ^ ")\n{"
| WhileEnd(opr1, op, opr2) ->
  let (l, lp) = expr opr1
      and (r, rp) = expr opr2
  in
      "}//while (" ^ l ^ " " ^ string_of_operator op ^ " " ^
    r ^ ")\n"

and string_of_declaration = function
  Decl(sl, decl, init) ->
    (string_of_specifier_list sl) ^ (string_of_declarator decl) ^
    (match init with
    NoInitializer -> ";"
      | ExprInitializer(e) -> " = " ^ (string_of_expression e) ^ ";"
      | ArrayInitializer(el) ->
      " = { " ^ String.concat ", " (List.map string_of_expression el) ^ " };"
      )
  | Function(sl, decl, body) ->
      (string_of_specifier_list sl) ^ (string_of_declarator decl) ^ " " ^
      (string_of_statement 0 body)
  | String(s) -> s
(** Inline blocks with no declarations *)

let rec flatten = function
  | Block([], sl) -> Block([], flatten_list sl)
  | If(e,s1,s2) -> If(e, flatten s1, flatten s2)
  | While(e,s) -> While(e, flatten s)
  | st -> st

and flatten_list = function
    [] -> []
  | Block([], sl)::tl -> (flatten_list sl) @ (flatten_list tl)
  | hd::tl -> (flatten hd)::(flatten_list tl)
```

IFL AST to C AST (asttoast.ml)

```
(* This file is not complete *)

open Ast
open Cast

type symbol_table = {
  parent : symbol_table option;
  variables : Cast.declaration.Decl list;
  functions: Cast.declaration.Function list;
}

type environment = {
  scope : symbol_table;
}

type t = Int | String | Bool

(* Find variables and functions, figure out if they exist *)

let rec find_variable (scope : symbol_table) name =
  try
    List.find (fun decl -> decl.name = name) scope.variables
  with Not_found ->
    match scope.parent with
    Some(parent) -> find_variable parent name
    | _ -> raise (Failure("This variable has not been defined."))

let variable_exists scope name =
    List.exists (fun decl -> decl.name = name) scope.variables
```

```
let rec find_function (scope : symbol_table) name =
  try
    List.find (fun f -> f.name = name) scope.functions
  with Not_found ->
    match scope.parent with
    Some(parent) -> find_function parent name
    | _ -> raise (Failure("This function has not been defined."))

let func_exists scope name =
    List.exists (fun f -> f.name = name) scope.functions
```

Interpreter header file – types.h

```
#ifndef _TYPESH_
#define _TYPESH_
/* Type definitions */
typedef enum { false, true } bool;

struct Exitlist
  {
   struct Room *north;
   struct Room *south;
   struct Room *east;
   struct Room *west;
   struct Room *northeast;
   struct Room *northwest;
   struct Room *southeast;
   struct Room *southwest;
  };

struct Room
 {
   char name[30];
   char desc[300];
   struct Exitlist exits;
 };
 struct Thing
{
   char name[30];
   char desc[300];
   struct Room *location;
   char action[300];
   void (*actionAddress)();
   bool inInventory;
  };
 struct GameMain
{
  struct Room *initialRoom;
  char intro[300];
 };

 #endif
  /* End of type definitions */
```

Interpreter header file – headers.h

```
#include "types.h"
#ifndef _HEADERSH_
#define _HEADERSH_

struct Room cave;
struct Room field;
struct Room yourRoom;
struct Thing charm;
void play();
struct Thing flute;
struct GameMain myGame;
int numberOfItems;
struct Thing *itemList[2];
#endif
```

Interpreter (Interpreter.c)

```
#include <stdio.h>
#include <string.h>
#include "types.h"
#include "headers.h"

/* Global variables */
char com1[20];
char com2[20];

struct Thing *inventory[10]={NULL};
struct Room *currentLocation=NULL;

bool gameFinished =false;

/* End of global variables */

   /*Start of included commands: inventory, look, take, examine */

showInventory ()
 {
   int i=0;
   if (inventory[0] == NULL)
    printf("\nYour inventory is empty.\n");
   else
   {
      printf("\nYou are carrying:\n");
      while ((inventory[i]!=NULL) && (i<10))
        {
          printf("%s\n", inventory[i]->name);
          i++;
        }
    }
 }
```

```
bool isInInventory (struct Thing item)
/*Returns whether the item passed as an argument is in the inventory or not.
*/
{
  if (item.inInventory)
    return true;
  else
    return false;
 }

lookAtRoom ()
 /* This procedure simply shows the description of the current room. */
  {
     printf("%s\n",currentLocation->desc);
  }

addToInventory (struct Thing *item)
 /* Adds  the desired item to the inventory. */
  {
    bool added=false;
    int i=0;
    while ((added==false) && (i<10))
    {
       if (inventory[i]==NULL)
          {
            inventory[i] = item;
            added = true;
            item->inInventory= true;
            printf ("\nAdded to inventory: %s\n", item->name);
          }
        else i++;
    }
  }

 struct Thing *addressOfItem ()
 /* Receives a command (name of an item), finds the address
 of the item that corresponds to that name and returns the address. */
 {
    struct Thing *address =NULL;
     int i=0;
     while (address ==NULL && i<numberOfItems)
     {
       if (strcmp(com2, itemList[i]->name) == 0)
           address=itemList[i];
       else
          i++;
     }
   return address;
 }

takeItem ()
 /* Verifies whether the item is already in the inventory. If not, calls
addToInventory. */
  {
   struct Thing *item = addressOfItem ();

   if ((item==NULL) || (item->location != currentLocation))
```

```
          printf ("\nI don't think that's something you can take.\n");
    else
      {
        if (item->inInventory)
            printf ("You are already carrying it.");
            else
           addToInventory (item);
       }
     }

 examineItem ()
   /* Verifies whether the item is already in the inventory.  If so,  shows
its description.*/
   {
    struct Thing *item = addressOfItem ();

      if (item != NULL && item->inInventory)
        printf ("%s\n", item->desc);
      else
        printf ("You can't examine something unless it's an item in your
inventory.");
     }


finishGameMessage (char finalMessage[20])
   /* When the game has been completed, this procedure will display
   the text sent as an argument to the player and finish the game. */
   {
     printf ("\n%s\n\n", finalMessage);
     gameFinished=true;
   }


compassDirection ()
 {
     if ((strcmp(com2, "n") == 0) || (strcmp(com2, "north") == 0))
         {
             if (currentLocation->exits.north != NULL)
             { printf ("\nYou head north.\n\n");
               currentLocation = currentLocation->exits.north;
               lookAtRoom();
             }
             else printf ("\nYou can't go that way.\n");
         }
     else

     if ((strcmp(com2, "s") == 0) || (strcmp(com2, "south") == 0))
         {
             if (currentLocation->exits.south != NULL)
             { printf ("\nYou head south.\n\n");
               currentLocation = currentLocation->exits.south;
               lookAtRoom();
             }
             else printf ("\nYou can't go that way.\n");
         }

     else
```

```
if ((strcmp(com2, "e") == 0) || (strcmp(com2, "east") == 0))
    {
        if (currentLocation->exits.east != NULL)
        { printf ("\nYou head east.\n\n");
          currentLocation = currentLocation->exits.east;
          lookAtRoom();
        }
        else printf ("\nYou can't go that way.\n");
    }
else
if ((strcmp(com2, "w") == 0) || (strcmp(com2, "west") == 0))
    {
        if (currentLocation->exits.west != NULL)
        { printf ("\nYou head west.\n\n");
          currentLocation = currentLocation->exits.west;
          lookAtRoom();
        }
        else printf ("\nYou can't go that way.\n");
    }
else
if ((strcmp(com2, "ne") == 0) || (strcmp(com2, "northeast") == 0))
    {
        if (currentLocation->exits.northeast != NULL)
        { printf ("\nYou head northeast.\n\n");
          currentLocation = currentLocation->exits.northeast;
          lookAtRoom();
        }
        else printf ("\nYou can't go that way.\n");
    }
else
if ((strcmp(com2, "nw") == 0) || (strcmp(com2, "northwest") == 0))
    {
        if (currentLocation->exits.northwest != NULL)
        { printf ("\nYou head northwest.\n\n");
          currentLocation = currentLocation->exits.northwest;
          lookAtRoom();
        }
        else printf ("\nYou can't go that way.\n");
    }
else
if ((strcmp(com2, "se") == 0) || (strcmp(com2, "southeast") == 0))
    {
        if (currentLocation->exits.southeast != NULL)
        { printf ("\nYou head southeast.\n\n");
          currentLocation = currentLocation->exits.southeast;
          lookAtRoom();
        }
        else printf ("\nYou can't go that way.\n");
    }
else
if ((strcmp(com2, "sw") == 0) || (strcmp(com2, "southwest") == 0))
    {
        if (currentLocation->exits.southwest != NULL)
        { printf ("\nYou head southwest.\n\n");
          currentLocation = currentLocation->exits.southwest;
          lookAtRoom();
        }
```

```
            else printf ("\nYou can't go that way.\n");
        }
    else printf ("\nYou can't go that way.\n");
    }

evalCommands ()
  /* If the user type a known two-word command (take or examine)
   this procedure evaluates it and outputs the result to the user. */
  {
    if (strcmp(com1, "take") == 0)
    takeItem ();
    else
      if ((strcmp(com1, "show") == 0) && (strcmp(com2, "inventory") == 0))
      showInventory ();
      else
         if (strcmp(com1, "examine") == 0)
          examineItem ();
         else
          if ((strcmp(com1, "look") == 0) && (strcmp(com2, "around") == 0))
             lookAtRoom ();
          else
             if (strcmp(com1, "go") == 0)
                compassDirection();
             else
               if ((strcmp(com1, "exit") == 0) && (strcmp(com2, "now") == 0))
                  gameFinished=true;

               else
                     { struct Thing *item = addressOfItem ();
                      if ((item != NULL) && (item->inInventory)
                         && (strcmp(com1, item->action) == 0))
                            item->actionAddress();
                      else printf ("You can't do that!\n");
                     }

  }

 readCommands()
  /* Reads the commands from the console and evaluates them. */
  {
    printf("\n> ");
    scanf("%s",com1);
    scanf("%s",com2);
    {evalCommands();}
  }

 main ()
 {
    printf("\n%s\n\n",myGame.intro);
    currentLocation = myGame.initialRoom;
    lookAtRoom();
    while (gameFinished==false)
    {
        readCommands ();
    }
    /* getch(); */
 }
```