Interactive Fiction Language Reference Manual

Gema Almoguera, UNI GA2347

COMS W115 – Summer 2013

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.

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.

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.

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.

2.3 Keywords

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

if
else
for
while
say
true
false
vocab_words
name
desc
room
location
Thing
InitialRoom
GameMainDef

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

look
examine
inventory
east
west
north
south
southwest
southeast
northwest
Northeast
take
drop

2.4 Constants

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

2.4.1. 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.

2.4.2. 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.

In order to represent double quotes, new line or a slash within a string, the following escape characters may be used:

Character	Escape Character
New line	\n
Double quotes	\”
Slash	\\

2.4.3. Boolean literals

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

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.

2.6 Punctuation

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

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.

4. Meaning of identifiers

4.1 Types

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. Please notice that the first letter of object type names must be capitalized.

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.

- **Objects**

  Programs in IFL are mostly 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: 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
     showIntro ()
    {
    "Introduction goes here."
    }
    ```

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

  - 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]
       [identifier = attribute_value]
       [function_definition]
    }
    ```

    Exit names must be one of the compass directions: north, south, east, west, northwest, northeast, southeast, or southwest. When an exit is defined from room 1 to room 2, it is not defined in both directions: an exit from room 2 back to room 1 is not automatically defined. This allows for the programmer to create one-way exits, for example, doors that lock behind the player after they have entered a room. If the writer wishes to create an exit that works in both directions, they must define an exit from room 1 to room 2 and a different exit from room 2 to room 1.

    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.

    Optionally, other local attributes can be defined. These can be other room properties that the writer can use along with items: for example, the room can have a room_is_dark: true attribute that can be changed to false when a lamp is lit.

Function structure and syntax is defined below, since it is common to both rooms and items.

- 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 "look *item-identifier*"), and the location where the item can be found.

  identifier: Thing
  {
  vocab_words = '*name1 [name2… nameN]*'
  name = "*This is the item name as presented to the player.*"
  desc = "*This is the item description.*"
  location = "*This is the item description.*"
  [*identifier* = *attribute_value*]
  [*function_definition*]
  }

  Just like with rooms, new attribute names can be defined. These can be other item properties that the writer can use. For example, a bboolean "lit" property can be defined for a lamp.

- Functions

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

  ```
  Action (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:

  ```
  Action (action)
  {
    if (action == "take")
      then say "You pick up the red potion and put it in your pouch.";
  }
  ```

  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.

4.2 Variable and Function Scope

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

The Actions function for a certain object is made public and can be called from any object in the game, but only the object containing said action has access to its local variables.

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

Since, as mentioned above, functions can be called from other objects, the syntax used to call a function is the name of the object followed by a period and then followed by the name of the function, as follows:

Name_of_Object.Action (*action-identifier*)

5. Expressions and operators

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

5.2 Unary operators

5.2.1 *! 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).

5.3 Binary arithmetic operators

5.2.1 *int-expression / int-expression*

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

5.2.2 *int-expression * int-expression*

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

5.2.3 *int-expression - int-expression*

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

5.2.4 *int-expression + int-expression*

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

5.3 **Relational operators**

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

5.3.1 *expression == expression*

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

5.3.2 *int-expression < int-expression*

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

5.3.3 *int-expression <= int-expression*

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

5.3.4 *int-expression > int-expression*

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

5.3.5 *int-expression >= int-expression*

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

5.3.6 *expression != expression*

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

5.4 Logical operators

5.4.1 *boolean-expression && Boolean-expression*

This operator evaluates if both boolean expressions are true.

5.4.2 *boolean-expression || Boolean-expression*

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

5.5 Assigment operator

5.5.1 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.

6. Statements

Statements are present within function bodies.

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.

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.

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.

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.

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.

7. Built-in Functions

- `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.

- `Say` (*string-literal*): This function will display the string sent as an argument.

- 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/examine: The player will be able to see the current room's description, as well as the description for any items in the current room or in their inventory.
    - 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.
    - Drop command: Using the drop command followed by an item name that the player is carrying will remove the item for the player's inventory and leave it in the current room.

- The game will also keep track of the player's current location, inventory, and which items and locations have been already discovered.

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.