VENTURE

COMS 4115 - Language Reference Manual

Zach Adler (zpa2001), Ben Carlin (bc2620), Naina Sahrawat (ns3001), James Sands (js4597)

Section 1. Lexical Elements

1.1 Identifiers

An identifier in VENTURE is a sequence of letters, uppercase A-Z, and lowercase a-z, a integer 0-9, and/or the underscore. The identifiers are case sensitive and are used to create the names for variable and functions. For example, the identifiers, "ifTiffanyExists" and "IfTiffanyExists" are two different identifiers are the same. Identifiers always begin with a letter, uppercase or lowercase. Identifiers cannot equal any keyword, as defined in the Keyword section.

1.2 Keywords

These keywords are reserved and are to be used by the language. These keywords can not be used as regular identifiers and are case sensitive:

- int, String, boolean, void
- events, room, item, start, npc, end
- if, else, while, return
- true, false
- fn, currentRoom, use

1.3 Literals

Literals are immutable and have a primitive data type corresponding to their value. The data type that a literal can be of is a string, and int or a boolean. Trying to assign a value to an incorrect data type will raise an error and you can not cast a value as a different type.

Examples: `"hello" , 69, true`

1.3.1 String Literals

A String literal is a sequence of characters that are enclosed by single quotes and/or escape characters enclosed in double quotes.

Examples: `"\n", "\t" "string"`

1.3.2 Integer Literals

An integer literals are a sequence of one or more integers, 0-9. Floating point numbers won't be recognized by the language and will raise an error.

Examples: `48, 17, 2748`

1.3.3 Boolean Literals

A boolean literal is a value of true or false. Assigning any other value of any other data type to a boolean variable will raise an error.

Examples: `true, false`

1.4 Operators

Operators are tokens that are used to denote an operation to be done on an element or combination of elements. Simple mathematical operations are common operators, such as addition, subtraction, multiplication and division. More on operators are explained in Section 3.

1.5 Delimiters

Delimiters help the compiler interpret tokens and separate tokens from each other.

1.5.1 Parentheses

Parentheses will be used to enclose arguments for a function, which is exemplified in section 5, Functions. Parentheses will also be used to make the compiler understand to evaluate a certain order of a program first.

1.5.2 Commas

Commas will be used to separate arguments of a function.

1.5.3 Brackets

Brackets will be used to declare arrays, used to access indices of an array and assignment of indices of an array.

1.5.4 Semicolon

The semicolon will designate the end of a statement.

1.5.5 Curly Braces

Curly braces will be used designate the statements that will be evaluated in a function. Examples are provided in the function section.

1.5.6 Periods

Periods will be used to access fields of objects in the standard library, such as a room or player.

1.5.7 Whitespace

Whitespace will be used to separate tokens for the compiler and has no special meaning. Examples of whitespace is a space, tab, newline and carriage return with the newline.

Section 2: Data Types

2.1 Primitive Data Types

2.1.1 int

This token int is case sensitive and will be used to store 32-bit signed integer, from -2,147,483,648 to 2,147,483,647.

2.1.2 Boolean

This token is case sensitive and will be used to store values of `true` and `false`.

2.1.3 String

This token will be used to store sequences of characters and will be where text values will be saved.

2.1.4 void

This token will only be used for return types in function definitions. An error will be raised if a developer tries to assign void to an identifier.

I.e.: `void iD = "tiffany";`

2.2 Non-Primitive Data Types

2.2.1 Arrays

Arrays are lists that contain primitive and non-primitive data types; however, an array must be of one the same data type;

2.2.2 Declaring Arrays

To declare an array, one must start with the data type the array will contain and then denote the identifier for the array and lastly, in brackets declare the amount of indices the array will have. This name will be case sensitive

I.e.: `String newArr [10];`

2.2.3 Accessing/Assigning Elements in an Array

An array's elements can be accessed by addressing the array by its correct identifier and, in brackets, supplying what index you wish to access. Indexes of arrays will begin at 0 and the last element of an element will one numerical value less than the number used to declare the array.

I.e.: `newArr[2];  ##The last element in this array will be index 1.##`

2.2.4 Length of an Array

The length of an array can be returned by using the name of the array followed by a period and the field "len"

I.e.: `int length = newArr.len;`

2.3 room

A room is an object in a program that is a location a player can go to to find items and NPCs. A room must be defined in the same file as the it will be used and is defined by typing "room" followed by curly brackets. Inside the brackets, fields for the room such as its name, its description, a boolean value of whether it has been visited. Other fields can be added as well. Initialization of these fields require the assignment of these fields to specific values.

I.e:

```
room{
      ## some fields ## = ## some value##
      }
```

2.3.1 Setting Adjacencies

To set a room adjacent to another room, the operators <> will be used. Being Adjacent means a user can go from room A to room B. I.e: roomA <> roomB. Multiple Adjacencies can be set by including the operator <> between multiple rooms. For example, roomA <> roomB <> roomC. This can be read as "roomA is adjacent to roomB. And roomB is adjacent to roomC." RoomA is not adjacent to room C. However; roomA<>roomC would make roomA and roomC adjacent.

2.3.2 Movement Amongst Rooms

To move to adjacent rooms, use the '>>' function. I.e.: >>roomC. If the room isn't adjacent, an error will be raised.

2.3.3 Start Location

A Start location is necessary when a developer uses a room in their program. Start notation is the keyword start followed by an assignment operator. I.e.: start = roomA

2.4 Defining NPC's and Items

Similar to Rooms, NPC's and Items can be defined by the keyword npc or item followed by curly brackets with fields and initial values which are assigned as noted below.

Examples:

```
Npc {
      String name = "tiffany";
      ##…##
}
```

```
Item {
      String name = "shank";
      ##...##
}
```

Section 3: Operators & Expressions

3.1 Operators

Operator	Description
.	attribute access
* / %	multiplication, integer division, modulo
+ -	Addition, subtraction
< <= > =>	less than, less than or equal to, greater than, greater than or equal to
== !=	equal, not equal
&	Logical AND
|	Logical OR
!	Logical NOT
=	assignment

3.2 Operator Precedence

Operator precedence is according to the order in which they appear in the above table; the attribute access has highest precedence and the assignment has lowest.

3.3 Expressions

Expressions consist of one or more operands and zero or more operators.

3.4 Expression Precedence

Expressions are evaluated in a way such that the innermost expressions are evaluated first, as indicated by parentheses. Each expression is evaluated left to right, accounting for operator precedence.

Section 4: Statements

4.1 if-else Statements

The if-else statement is used to execute a block of code if a specified condition is met, and if the "if" condition is not met, another block of code after the "else" is executed. The general form of an if-else statement is as follows:

```
if (condition) {
      expression;
} else {
      expression1;
}
```

In the case where there is no alternative block of code you'd like to execute with the "else", that block of code can be left empty, but if-else statements require both an "if" and an "else":

```
if (condition) {
      expression;
} else {
}
```

4.2 while Statements

The while statement is used to execute a block of code repeatedly (in a loop) until the specified condition is no longer met. The condition is checked immediately before entering each iteration of the loop. The general structure of a while loop is as follows:

```
while (condition) {
      expression;
}
```

Section 5: Functions

5.1 Function Definitions

Function definitions consist of an initial keyword "fun," a return type, a function identifier, a set of parameters and their types, and then a block of code to execute when that function is called with the specified parameters. An example of an addition function definition is as follows:

```
fun int sum( int a , int b ){
return a + b; }
```

5.2 Calling Functions

A function can be called by its identifier followed by its parameters in parentheses, such as in the following example:

```
sum(1, 2);
```

Section 6: Program Structure and Scope

6.1 Program Structure

VENTURE programs will be one source file. VENTURE programs consist of:

1. Import libraries
2. Room declarations
3. Adjacency Declarations
4. NPC Declarations
5. Item Declarations
6. Start Declarations
7. Global Variable Declarations
8. Function Declarations
9. Event conditions and routines

6.2 Event Handling System

VENTURE programs will be event driven. The user will define 'events' which will consist of a condition and a routine that will be carried out if the condition evaluates to 'true'. A VENTURE program will run in a loop which evaluates each event after each instance of user input.
The keyword 'events' is used to create a new scope in which all of the events are defined at the end of a VENTURE program:

```
events {

      Tiffany.room == currentRoom {
            print("Tiffany says: Hello traveler.");
      }

      Tiffany.alive == true {
            end_game = true ;
      }
}
```

The events loop will terminate when the `end_game` global variable no longer evaluates to true.

6.3 Libraries

Libraries are imported using the keyword 'use' at the beginning of the VENTURE program.

```
use stdlib;
```

6.4 Scope

Global declarations are made outside of the block of a function definition or if statement or while statement. Globals can be referenced in the program following their declaration. Declarations made within blocks of an if statement, a while statement, or a function definition are only available for reference within that block. Brackets can be used for nested scoping.

```
string hello = "hello";
if condition {
      int value = 5;
      {
            int value2 = 10;
      }
      ## value2 is out of scope here ##
}
## both value1 and value2 are out of scope here##
## hello can be referenced throughout the program ##
```

Section 7: Built In Functions

The `print` function can be used to print integers, booleans, or strings.

```
print(arg);
```

The `println` prints the arg, be it an integer, boolean, or string. After the argument is printed, a newline character is printed.

```
println(arg);
```

The `len` function takes as an argument an array, and returns an integer value of its length.

```
len(arg);
```

The `user_input` function accepts text input from i/o with each '\n'. This function returns a string.

```
user_input();
```

Section 8: Context Free Grammar

expr -> literal
id
id actuals
expr Add expr
expr Sub expr
expr Mult expr
expr Div expr
expr And expr
expr Or expr
expr Equal exor
expr Neq expr
expr Less expr
expr Greater expr
expr Geq expr

type -> int
-> string
-> boolean
-> string
-> void

vdcel -> type id expr
type id asssign
type id