Racontr Language Reference Manual

Morgan Zee (mbz2112), Shirley Ye (sy2650), Saumya Agarwal (sa3656), Xinye Jiang (xj2253), Janelle Ponnor (jp4024)

February 24, 2020

Contents

1. Overview of Racontr

The Racontr programming language allows users to design and implement their own creative text adventure games. Racontr is fairly dynamic and can be used to develop a wide range of stories with customizable people, places, and things. The adventure that players can embark on will be in the hands of the programmer, who can either provide the user with predefined storylines that vary depending on what option the user selects or allow the player to decide how the story unfolds.

1.1 Goals and Motivations

Racontr is inspired by projects done by students in previous semesters, including GAWK (2014), a language used to build role-playing games, and GRIMM (2004), an interactive story-building language. In particular, we used the sample games from GRIMM as a key example of potential games that can be implemented in Racontr. We paid attention to their type declarations, assigning attributes to specific objects, and conditional statements. We also adapted elements from existing programming languages like Python, in terms of syntax and functionalities, and the interactive fiction programming language ZIL, specifically in terms of creating objects and using Boolean flags to enable specific manipulations of objects. We followed the basic structure of the Language Reference Manual of Coral (2018) and the C Reference Manual.

In terms of goals, we hope Racontr will 1.) allow users to easily define and customize people (characters), places (scenes), and things (items) to build detailed and creative scenarios, 2.) be easier to build text-adventure games than existing object-oriented languages, and 3.) incorporate slightly adapted, yet familiar syntax from Python to maximize simplicity and ease of use.

We have drawn on elements from the existing languages and interactive fiction experiences discussed above to develop Racontr, which we hope programmers and players alike will use to have fun and expand their creativity.

2. Lexical Conventions

There are five kinds of tokens: comments, identifiers, keywords, constants, operators. In general blanks, tabs, newlines, and comments as described below are ignored except as they serve to separate tokens. At least one of these characters is required to separate otherwise adjacent identifiers, constants, and certain operator-pairs. If the input stream has been parsed into tokens up to a given character, the next token is taken to include the longest string of characters which could possibly constitute a token.

2.1 Comments

The characters /* introduce a comment, which terminates with the characters */. They do not indicate a comment when occurring within a string literal. Comments do not nest. Once the /* introducing a comment is seen, all other characters are ignored until the ending */ is encountered.

2.2 Identifiers

An identifier, or name, is a sequence of letters, digits, and underscores (_). The first character cannot be a digit. Uppercase and lowercase letters are distinct. Name length is unlimited. The terms identifier and name are used interchangeably.

2.3 Keywords/Type Specifiers

The following identifiers are reserved for use as keywords, and may not be used otherwise:

```
return
if
elif
else
endif
for
while
endwhile
int
bool
string
extends
assert
scene
character
item
name
description
in
say
says
read
def
```

2.4 Literals/Constants

The three types of constants are integer, string, and boolean. Each constant has a type, determined by its form and value.

2.4.1 Integer constants

An integer constant is a sequence of digits.

2.4.2 Strings

A string is a sequence of characters surrounded by double quotes ‘‘ " ’’. In a string, the character ‘‘ " ’’ must be preceded by a ‘‘\’’.

2.4.3 Booleans

A boolean can have one of two values: true or false. It is used to perform logical operations, most commonly to determine whether some condition is true.

2.5 Operators

An operator specifies an operation to be performed. The operators ( ) and { } must occur in pairs, possibly separated by expressions. An operator can be one of the following:

{ } ( )
; , =
! = < <= & |
+ - *

3. Pre-defined Data types

Aside from int, string, boolean, and collection types such as list and array, there are five essential customized data types that allow the users to define the game: Scene, Item, Character, Mission, Ending. Related to the five essential data types, supporting property types help define the details; some of them should be customized by the users, while some of them are built in (mentioned in 6.4). There are also two optional data types that can enrich the game: Achievements and Story Background.

3.1 Scene

Scene is a data type that contains information about places a player can explore. The user would be expected to define a collection of scenes that characterize a virtual map of the game. The Scene contains sub-data types; some of them should be customized, while some of them should be selected from built-in property types.

3.1.1 Name

This contains a size-60 String of the scene's name. Scene's names are unique.

3.1.2 Address

This is a collection of the relationship between the current scenes and the adjacent scenes. Their relationship is defined by the Built-in Property types: NORTH, SOUTH, EAST, WEST, NE, SE, NW, SW. The adjacent scenes can be accessed through these addresses.

3.1.3 Description

This contains text that describes the scenes.

3.1.4 Action

Users should define a list of actions that the character can make. Each action should be defined with a line of String. The action can result in a change of Scene, Character's status, missions' status, item's status, and/or achievements' status, depending on the users' definition.

3.2 Item

Item is a data type that contains information about an item in a scene that can be interacted by the player.

3.2.1 Name

This contains a size-60 String of the item's name. Items' names are not unique.

3.2.2 ID

This is an ID for the item, by default set to 0. This would differentiate items with the same name.

3.2.3 Effect

Users should define a list of effects of the item. Each effect should be defined with a line of String. The action can result in a change of Scene, Character's status, missions' status, item's status, and/or achievements' status, depending on the users' definition.

3.3 Character

This is the data type that contains information about player-controlled characters and other user-defined characters.

3.3.1 Name

This contains a size-60 String of the character's name. Characters' names are unique.

3.3.2 Category

Users can choose from "Enemy", "NPC", and "Player-Control". This is a String.

3.3.3 Health

This is an integer that indicates the health of the character. For NPC and Player-controlled characters, the health is by default 100, the upper limit is 200, and the lower limit is 0. For Enemy, the upper limit is infinite, and the lower limit is 0.

3.3.4 Item

This is a list of the items that the character's possessing.

3.3.5 Scene

This is the location of the current character.

3.4 Mission

This is the data type that contains the information of missions listed by the users for the players. This is optional and should be checked after every action is performed.

3.4.1 Name

This contains a size-60 String of the mission's name. This is unique.

3.4.2 Description

This contains text that describes the missions and how it can be completed.

3.432 Status

This contains a boolean value indicating whether the mission is completed or not.

3.5 Achievement

This is the data type that contains the information of achievements a player can make. This is similar to PlayStation's trophies and Steam's achievements. This is optional.

3.5.1 Name

This contains a size-60 String of the achievement's name. This is unique.

3.5.2 Description

This contains text that describes the achievements and how it can be completed.

3.5.3 Status

This contains a boolean value indicating whether the achievement is completed or not.

3.6 Story Background

This contains the user-defined story background of the game. This should be provided to the player before they start the game.

3.7 Ending

This contains the user-defined endings for the game. This could be related to achievements and is mandatory to implement.

3.7.1 Name

This contains a size-60 String of the ending's name. This is unique.

3.7.2 Description

This contains text that describes the ending.

3.7.3 Status

This contains a boolean value indicating whether the ending is reached. If reached, the description will be provided to the player.

4. Statements and Expressions

There are various types of statements and expressions that can be written in Racontr. These include conditional statements, declaration statements for defining variables and functions, and assignment statements. Racontr also makes use of binary operators to write useful expressions.

4.1 Conditional Statements

In Racontr, users can utilize various conditional statements, including if, elif, and else statements, for loop statements, and while loop statements. These statements align closely with the clear and concise syntax and functionality of the conditional statements provided in Python.

4.1.1 If, Elif, Else Statements

Racontr supports if, elif, and else statements. If statements begin with a conditional predicate or expression followed by a collection of statements enclosed in curly braces {}. The collection of statements of the conditional are indented and describe actions to if the predicate is met. If the conditional predicate evaluates to True, then the statements within the curly braces are evaluated and executed. If the conditional predicate evaluates to False, the program will continue to the next statement. The next statement could be an additional special condition that the user wants to define for the same variable tested in the if statement. The syntax will match the if statement, but will begin with the keyword elif. There is also the option to insert a final statement following the same syntax but starting with the keyword else. If neither the if and elif conditions evaluate to True, the program will execute the statements enclosed in the curly braces of the else condition.

A sample of if, elif, and else conditional statements in Racontr would appear as follows:

```
if condition:
```

```
    /*statements*/

elif condition:
    /*statements*/

else:
    /*statements*/
```

4.1.2 For Loop Statements

Racontr supports for loops as Python does. The for loops begin with the word For, followed by a counter variable such as “i” or “x”, initialized outside of the loop, and a specified range. The range is specified in terms of an iterable object within the parentheses that follow the word range. In a text-adventure game, the for loop may be used to iterate through a data structure, such as an array or a list, that contains a character’s possessions and then prints its contents.

A sample of a for loop statement in Racontr would appear as follows:

```
character_possessions = [“magic wand”, “golden key”, “invisible cloak”, “baby dragon”]

for x in character_possessions:
    print(x)
```

The example above shows a code snippet from a hypothetical text-adventure game that gives characters a list of possessions, which the player can access and view. The for loop in Racontr can iterate over data structures such as lists.

```
for i in range(1, len(arr)):
    print(i)
```

The above sample code shows the use of the range keyword, starting at 1 and looping for the length of the array, which can be defined before the loop. This specific range is specified within the parenthesis after the word range.

```
for x in range(5):
    print(x)
else:
    print(“You win!”)
```

Similar to Python, the word range starts the loop at 0 and increments by 1 by default, but can be specified otherwise as shown in the previous code samples. In the case of the above example, x starts at 0 unless specified differently outside the loop and is incremented until x reaches 5. There is also the option of adding an else statement as shown, which will be executed when the for loop is complete.

4.1.3 While Loop Statements

Racontr also supports while loop statements, which start with the word while, a conditional predicate, and a collection of statements. As long as the condition evaluates to True, the statements within the loop are continuously evaluated. The program continues beyond the loop when the condition is False.

A sample of a while loop statement in Racontr would appear as follows:

```
while condition:
     /*statements*/
```

Considering a specific instance of use in a text-adventure games Racontr might produce, a game designer may choose to give players a limited number of lives. This allows players to carry out specific actions as long as they have lives available.

```
while has_lives:
     /*statements*/
```

Another example would be allowing a user to carry out specific actions as long as they are in a specific location shown as follows:

```
while in Library:
     /*statements*/
```

Racontr uses the conditional keywords while in and endwhile to indicate when the user is in the main method. This allows for extra clarity and organized code. This may appear as follows:

```
while in Location
     /*statements*/
endwhile
```

4.2 Declaration Statements

4.2.1 Variable Declaration and Assignment Statements

Racontr allows users to define variables using three keywords made up of the string data type. These keywords include character, scene, and item. Users can create characters by using the keyword character followed by the name of the character. The characters can interact with one another, move between scenes, and possess various items. In a similar way, users can use the keyword scene followed by a location and the keyword item followed by a thing to create these variables as well.

A sample of declaring character, scene, and item variables to be used throughout a text-adventure game would appear as follows:

```
character name
scene location
item thing
```

Users can take these declarations further by assigning specific attributes or details to the people, places, or things they construct. These attributes or assignment statements are enclosed in curly braces and exist whenever the object of type character, scene, or item is called. The assignment statements include the variable name, followed by an equals sign operator, and an expression such as a string or a list. The sample code below shows a series of assignment statements that are used to customize a scene. It is also worth noting the Global variables, objects that can exist in multiple scenes, and Local variables, objects that only exist in the specified scene, can also be declared as shown below.

```
scene location{
     /*attributes/details about the place*/
}

character name{
     /*attributes/characteristics*/
}

item thing{
     /*attributes/details about the thing*/
}
```

4.2.2 Function Calls and Declaration Statements

Functions are declared with the keyword def, followed by an identifier, parenthesis, and a colon. The contents of the function can be a series of statements, which will be carried out if the function is called. Arguments can be passed into the function within the parenthesis.

A sample of declaring a function in Racontr (similar to Python syntax) would appear as follows:

```
def identifier():
     /*statements*/
```

Functions will likely be called within the "main" method, within the keywords while in and endwhile as shown below. Functions are called with the identifier followed by parenthesis.

```
while in location
     /*statements*/
     identifier()
endwhile
```

4.3 Expressions

The main expressions Racontr uses are identifiers (similar to variables), strings, and constants (integers, booleans). Racontr expressions are evaluated from left to right and follow the standard precedence of operators, starting with statements in parenthesis, exponents, multiplication, division, addition, and subtraction. While not all of these operators exist in Racontr, it is still useful to cover order of operations.

4.3.1 Binary Operators

Racontr supports arithmetic operators: Plus (+), Minus (-), Times (*). These operators appear between expressions.

```
expr + expr
expr - expr
expr * expr
```

It supports comparison and equality operators: Equals (=), Less than (<), and Less than equals (<=). These statements evaluate to True if the comparison is True and False otherwise.

```
expr == expr
expr < expr
expr <= expr
```

It supports logical Boolean operators: and, or, not.

```
expr and expr
expr or expr
not expr
```

We intentionally eliminated redundant operators (such as Greater than (>) and Greater than equals (>=) since we already have Less than and Less than equals operators) to reduce complexity.

5. Classes

5.1 Class Definitions

Racontr will contain built in classes like Scene and Character. However, users can also define their own classes. The class type will be followed by the class name. The below example code sets up an instance of a Scene class named butler. The body of the class will be within the two curly braces.

```
Scene butler{}
```

Within each class, users can add variables and data structures that can be attributed to that specific class. For example, for the above example Scene butler, this can entail a name field and nearest exits.

```
Scene butler{
     name = "Butler Library"
     EXITS = SOUTH TO lawn, NORTH TO 113th Street, EAST TO
Lerner
}
```

Outside of this class definition, when the user writes code that involves a class defined beforehand, all contents defined in the class are available to them. For instance, after the class instance above, the below code is valid since it references values specific to the butler scene class.

```
while user in butler:
     exit butler to lawn
endwhile
```

5.2 Inheritance

Racontr also supports inheritance between classes by allowing one class to inherit from a superclass. To make class2 inherit from class1, the syntax would be *class2 extends class1* when defining class2. This would allow situations involving class2 to have access to the same instance variables and methods as class1 as well as additional values that the user can define. For instance, if the user were to create a Scene butler_room202 that would clearly inherit from the Scene butler class, the syntax would be the following.

```
Scene butler_room202 extends butler{
     name=”Butler Library: Room 202”
}
```

Then, the user can continue to write code that references both values defined in the superclass Scene butler as well as values defined only in the subclass Scene butler_room202.

6. Standard Library

6.1 List

Racontr has a built-in list data structure with dynamic length. Lists in Racontr can hold objects of any arbitrary type and behave identically to Python lists, and support the following operations:

Method	Behavior
list[x]	Returns the xth element
list.append[x]	Adds element x to the end of the list
list.remove[x]	Removes element x from the list
list.count[x]	Returns the length of the list

6.2 Strings

Class Strings in Racontr supports indexing and a variety of useful methods for handling text-based data. Any operation on an existing string will return a new string.

Method	Behavior
str[x]	Returns the xth character of the string
str1 + str2	Concatenate two strings

6.3 Properties

Properties make up the object definitions of scenes, characters, and things in Racontr. This class has three main functions that allow users to handle properties of an object.

Method	Behavior
getp[x]	Returns information on Property x of an object
putp[x]	Add Property x to an object
setp[x]	Change Property x of an object to a newly defined one

6.4 Built-in Property types

NORTH, SOUTH, EAST, WEST, NE, SE, NW, SW: These are the direction properties, generally used only in room definitions. For the various types of direction properties, see section 2.2. Note that the cardinal direction properties are not abbreviated, but that the non-cardinal ones are abbreviated. There is no direction property called NORTHEAST, for example.

UP, DOWN: These are just like the eight direction properties.

IN, OUT: These are just like the eight direction properties. If the player just types IN or OUT, this property will handle the movement. Generally, it's a good idea to give the OUT property to any room with only one exit.

ACTION: Defines the action routine associated with the object. In the case of an object, the action routine is called when the object is the PRSO or the PRSI of the player's input. In the case of a room, the routine is called with M-BEG and M- END once each turn, with M-ENTER whenever the room is entered, and with M- LOOK whenever the describers need to describe the room.

LOC: Once again, technically not a property, but it looks just like one when you're creating an object. Simply, this property contains the name of the object which contains this object (in the case of a room, this is the object ROOMS).

SIZE: Contains a number which is the size/weight of the object. Generally, it is only meaningful for a takeable object. If a takeable object has no size property, the game usually gives it a default size of 5. The size of an object affects the number of object that a player can carry, how much of a container it takes up, and so on.

CAPACITY: Contains a number which is the capacity of the object. Generally, it is only meaningful for a container. If a container has no size property, the game usually gives it a default capacity of 5. The capacity of a container affects the number of objects which can be placed inside it.

VALUE: This property is used in many games that have scoring. The property contains a number; in the case of rooms, it is the number of points the player gets for entering the room for the first time; in the case of objects, it is the number of points the player gets for picking up the object for the first time. GLOBAL: Generally found only in room definitions, this property contains a list of objects which are local-globals referenceable in that room.

TEXT: This property contains a string which is used when the player tries to read the object. It exists for those objects which would otherwise need an action routine to handle READ but nothing else.

THINGS: Formerly known as the PSEUDO property, this property allows you to create "pseudo-objects" with some of the properties of real objects. They have three parts: a list of adjectives, a list of nouns, and an action routine. Here's an example:

(THINGS (RED CARMINE) (SCARF ASCOT) RED-SCARF-F)

Pseudo objects are very limited, however. They cannot have flags, and they cannot be moved. It is beneficial to use them whenever feasible, because (unlike real objects) they take up no pre-load space.

PICTURE: Contains the name of a graphic from the picture file associated with the room or object.

FLAGS: This is another fellow which looks just like a property but isn't actually a property. It contains a list of all the flags which are FSET in that object at the start of the game. Examples of common flags include ENTERBIT and EXITBIT, which could be applied to a scene object for example to allow a user to enter and exit.

7. Sample Code

This is a potential game users can write using Racontr. This snippet is adapted from a game in the GRIMM programming language.

```
/*declare scenes and characters*/
Scene butler
Scene lawn
Character Player1
Character librarian
Character barista

/*define the class scene butler, use as variable*/
Scene butler{
    name = "Butler Library"
    description = "You are currently in Butler Library. Find the
key and return it to the owner on the lawn"
    picture = readfile("butler.jpg")
    EXITS = SOUTH TO lawn, NORTH TO 113th Street, EAST TO Lerner
    ITEMS= [key, money, book]
    ACTIONS = butler() //display options
    FLAGS ENTERBIT, EXITBIT
    Global Stairs, Bookshelves, Trashcans
    Local Prezbo's bookshelf }

/*similar to main method, shows game is in session*/
while user inside butler:
     options = [$option 1, $option 2, $option 3]
     if $option 1:
     if user has BOOK:
         then librarian puts BOOK on Prezbo's bookshelf AND
gives KEY
        else
            librarian says "Your book is overdue."
            back to butler()
    if $option 2:
        then barista says "Cafe is closed."
        back to butler()
    if $option3:
        exit butler to lawn
endwhile
```