Racontr

The Racontr Programming Language

Programming Languages and Translators Final Report
Spring 2021

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

Table of Contents

1. Introduction

2. Racontr Tutorial

3. Language Reference Manual

4. Project Plan

1. Introduction

1.1 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 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.

1.2 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 a potential game 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 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. Racontr Tutorial

Start by downloading the folder containing the files for Racontr. The following tutorial contains instructions to set up your environment and compile Racontr.

2.1 Environment Setup

Before getting started, make sure to install Ocaml and LLVM, which can be installed on a Mac OS with the commands

```
brew install opam
brew install llvm
opam install llvm
```

Other methods include downloading the [Docker Desktop]() or installing Homebrew and running the command

```
brew install docker
```

We used the microc docker image provided by columbiasedwards/plt. Navigate to the directory of the project files and activate the docker container by running

```
docker run --rm -it -v `pwd`:/home/hello5 -w=/home/hello5
columbiasedwards/plt
```

This will activate the microc docker image and open a container that can be used to run Ocaml. Ensure that you are inside the docker container: /home/hello5#

2.2 Downloading and Building Racontr

First build the racontr.native file using the following command

```
ocamlbuild -use-ocamlfind racontr.native
```

You can compile the Racontr compiler using the following command

```
 make
```

2.3 Writing and Compiling a Simple Program: Helloworld

After compiling Racontr, you can write your own programs! The Language Reference Manual in the following section will provide syntax guidelines and instructions for writing programs in the Racontr language.

For now, here are instructions to implement your first Racontr program. Start by creating a file called helloworld.rac and copy and paste the following code onto it:

```
var helloworld : string = "helloworld"

print_string(helloworld)
```

To compile and run this code, type the following commands into your terminal

```
ocamlbuild -use-ocamlfind racontr.native
make
```

You can create a helloworld.out file containing the expected output “helloworld” to compare with the output of your helloworld.rac code.

2.4 Debugging Options

If you run into problems along the way, you will see errors listed in the terminal. Use the following command

```
cat testall.log
```

to access more details about the encountered errors, which can be used for debugging.

3. Language Reference Manual

3.1 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.

3.1.1 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.

3.1.2 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.

3.1.3 Keywords/Type Specifiers

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

```
return
if
elif
else
for
while
int
bool
string
extends
assert
scene
character
item
in
def
not
```

3.1.4 Literals/Constants

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

3.1.4.1 Integer constants

An integer constant is a sequence of digits.

3.1.4.2 Strings

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

3.1.4.3 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. (add boolean literals)

3.1.5 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:

{ } ( )
: , = >=
!= < <=
& |
+ - * /

3.2 Types and Values

Racontr has two types: primitive and reference, and two types of values: primitive values and reference values.

3.2.1 Primitive Types and Values

The integer type is i32.
The boolean type has two values: true and false.
The string type is a constant literal.
And the void type.

3.2.1.1 Integer Types and Values

The range for an int is from -2147483648 to 2147483647, inclusive.

3.2.1.2 Boolean Types and Values

The boolean type represents a logical quantity with two possible values, indicated by the literals true and false.

3.2.1.3 String Types and Values

The string type is a series of chars surrounded by double quotes.

3.2.2 Reference Types and Values

The reference type is the class type, of which there are two: the class scene and the class character.

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).

3.2.2.1 Inheritance

Racontr also supports inheritance between classes by allowing one class to inherit attributes from a superclass. This would allow situations involving the subclass to have access to the same instance variables as the superclass as well as additional values that the user can define.

```
class subclass_identifier extends superclass_identifier{}
```

3.2.2.2 Objects

An object is a class instance. The reference values are pointers to these objects, and a special null reference, which refers to no object.

```
class identifier {
/*type declarations*/
}
```

3.2.2.3 The Class Scene

Scene is an in built class 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. 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.

The syntax for defining a scene is:

```
class identifier extends Scene {
/*type declarations*/
}
```

3.2.2.3.1 Name

This contains a string of the scene’s name. Scene’s names are unique.

3.2.2.3.2 Description

This contains text that describes the scenes.

3.2.2.3.3 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.

3.2.2.4 The Class Character

Character is an in built class containing information about each player. The user would be expected to define basic attributes of each character, including what items they have access to. The class character contains sub-data types; some of them should be customized, while some of them should be selected from built-in property types. 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.

The syntax for defining a character is:

```
class identifier extends Character{
/* type declarations */
}
```

3.2.2.4.1 Name

This contains a String of the character’s name. Characters’ names are unique.

3.2.2.4.2 ID

This is an ID for the character. This differentiates different characters.

3.3 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.

3.3.1 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.

3.3.1.1 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.

The syntax for defining if, elif, and else conditional statements in Racontr would appear as follows:

```
if expression {
     /*then-statements*/
}
elif expression {
     /*then-statements*/
}
```

```
else {
     /*else-statements*/
}
```

3.3.1.2 For Loop Statements

Racontr supports for loop statements, which start with the word for, followed by an expression that indicates when the loop begins, an expression that indicates when the loop should end, and an expression that indicates how much the start expression should increment with each loop, all enclosed in parenthesis. Until the loop has incremented to the stop-expression value, the statements within the loop are evaluated.

```
for (start_expression; stop_expression; increment_expression) {
     /* statement */
}
```

The start-expression specifies the counter variable initialization for the loop; the stop-expression specifies when the loop should run, and this expression is checked before each iteration, so the loop only proceeds while the expression is true; the increment-expression specifies by how much the counter variable (initialized in the start_expression) should be incremented after each iteration.

3.3.1.3 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 expression {
     /*statements*/
}
```

3.3.2 Declaration Statements

3.3.2.1 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.

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.

```
var identifier : type = string literal
```

3.3.2.2 Function Calls and Declaration Statements

Functions are declared with the keyword def, followed by an identifier, parenthesis, and braces. 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 would appear as follows:

```
def identifier(parameter: type)-> return type{
     /*statements*/
}
```

3.3.3 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, which is:

{ } ( )
; , ==

= < <= & |
*
+ -

3.3.3.1 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
```

3.4. Standard Library

3.4.1 List

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

Method	Type of x	Behavior
list[x]	int	Returns the xth element
list.append[x]	object	Adds element x to the end of the list
list.remove[x]	object	Removes element x from the list

list.count()	-	Returns the length of the list

3.4.2 Strings

Class Strings in Racontr can be printed.

3.4.3 Properties

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

Method	Type of x	Behavior
getp[x]	object	Check if object has property; If there is a property, returns information on property x of an object
putp[x]	property	Add property x to an object
memp[x]	property	Change property x of an object to a newly defined one

3.5 Sample Code

Below is the code for Racontr’s Hello World game. It prints out “helloworld”.

```
var helloworld : string = “hello world”
print_string(helloworld)
```

Below is the code to write a Fibonacci program in Racontr.

```
def fib(n : i32) -> i32 {
     if n == 1 || n == 0 {
          return 1
     }
    return fib(n - 1) + fib(n - 2)
}
```

```
var fib_result : i32 = fib(10)
print_int(fib_result)
```

4 Project Plan

4.1 Planning, Specification and Development

Our group had weekly meetings to check-in with our progress and make sure that everyone was on the same page about next steps. We also met with our project advisor, Professor Edwards who ensured that the timeline and scope of our project was reasonable and advised us on how to implement our project ideas. He helped us debug and compile many of our files. At the beginning of the semester, we were very ambitious with our language design. With Professor Edwards' help, we were able to narrow down the scope of our language to ensure that we would be able to implement the most important features of our language.

During our team meetings, we discussed the goals and needed steps to implement Racontr. We also helped each other troubleshoot any issues that we were having and talked about options for resolving any bugs in our code.

Our day to day communication happened over GroupMe. This allowed us to work efficiently and communicate time sensitive concerns or questions about our language implementation.

4.2 Style Guide

We followed the following style guidelines while developing our compiler:

- Indent clearly.
- Use descriptive variable names to make it easier to understand the code.
- Simplify programs when and if possible.

4.3 Software Development Tools

We used the following programming and development environments when creating Racontr:

- Libraries and Languages: Ocaml Version 4.12.0 including Ocamlyacc and Ocammllex and LLVM was used.
- Software: Development was done in vim, Sublime and VSCode.
- OS: Development was done on MacOS 10.14.6.

4.4 Roles and Responsibilities

Team Member	Role & Responsibilities
Janelle Ponnor	Test Designer, LRM, Makefile, Parser, Scanner, AST, Semant, Codegen, Test Cases, Final Report
Morgan Zee	Manager, LRM, Parser, AST, SAST, Scanner, Semant, Codegen, Racontr.ml, Makefile, Final Report
Saumya Agarwal	System Architect, AST, Scanner, Parser, Semant, Final Report
Shirley Ye	Language Guru, LRM, Codegen, AST, Final Report, Test Cases
Xinye Jiang	System Architect, Codegen, Semant, Final Report

We wanted to take on a more collaborative approach and every team member was expected to contribute to every file in the compiler. We often worked on whatever needed to be completed over a Zoom call.

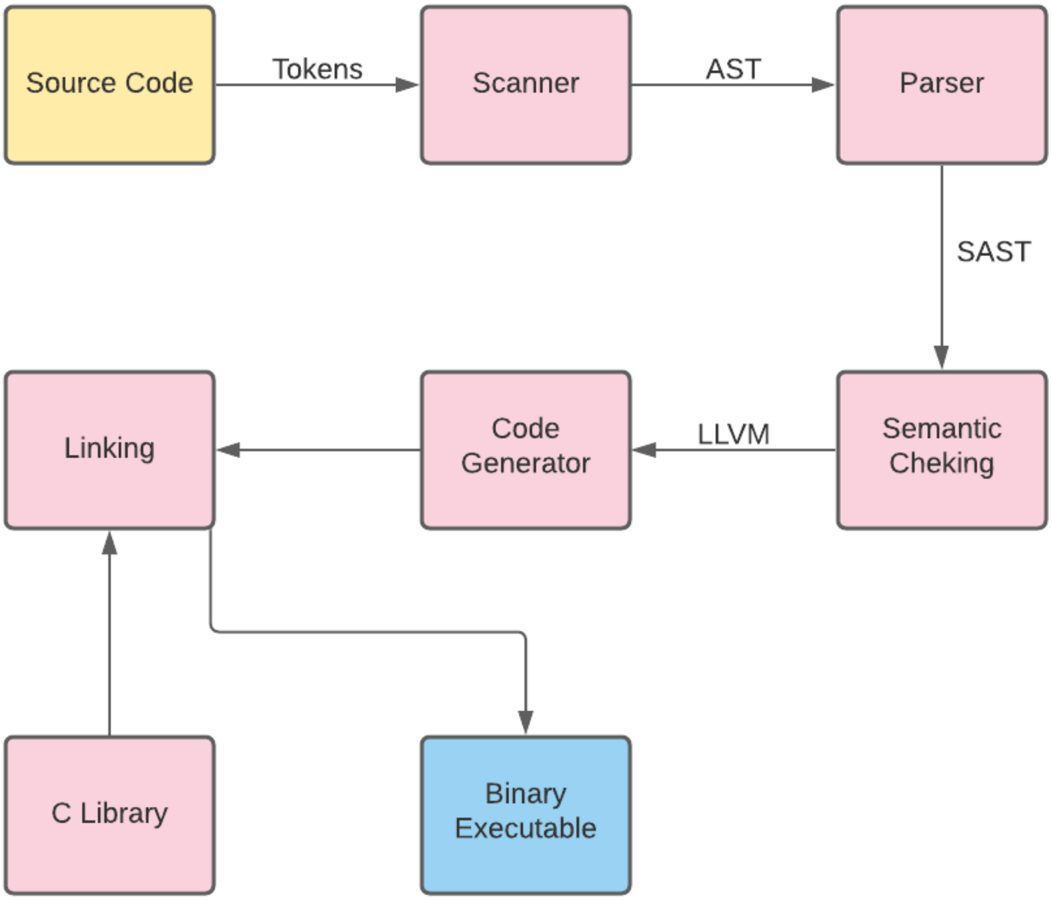
4.5 Project Timeline

Jan 26 - Initial Discussion to decide language
Feb 3 - Language Proposal
Feb 23 - LRM and parser
March 7 - Continued parser, first implementation of AST, scanner
March 15 - Continued parser, Semant, first implementation of Codegen, Makefile
March 24 - Continued Semant, SAST, Codegen
April 10 - Codegen
April 20 - Test cases
April 25 - Worked on presentation and final report

*Approximate timeline: continued working on each file in the compiler throughout the semester

5 Architectural Design

5.1 Block Diagram



5.2 Scanner

File: racontrscanner.mll

The scanner takes in the program file and tokenizes it into literals, identifiers and keywords. Comments are removed during this stage. The scanner throws an error for unimplemented python keywords and syntactically invalid identifiers or literals.

5.3 Parser

File: parser.mly

The parser is written in Ocamlyacc. The parser converts the tokens from the scanner to an abstract syntax tree (AST) based on Racontr’s context-free grammar rules for syntax described in the Language Reference Manual. If any violations are detected, such as unmatched parentheses, parser errors will be thrown.

5.4 Semantic Checker

File: semant.ml

The semantic checker recursively traverses the AST and converts it to a semantically - checked abstract syntax tree (SAST) consisting of objects. An environment record is used to map a string identifier to an object stack. If there are typing or scoping errors, messages will be printed to indicate the type of errors. For example, if variables are referenced before initialization or assigned a different type than what was declared, the semantic checker will generate errors.

5.5 Code generation

File: racontrcodegen.ml

The code generator takes in the semantically checked SAST and builds the LLVM. For objects with known types, the data itself is simply placed on the stack. External functions are declared, functions prototypes are defined, and allocates formal arguments and local variables inside the file. We also define some global variables uniquely for the language to build the storylines. Additionally, expressions, operators, built-in functions, and if/while statements are instructed to LLVM. If we want to add more functions to the language, we can easily extend the build_function_body with the added features.

6. Test Programs

Racontr’s test cases are in the tests/ folder. The successful test cases start with test-*.rac and the test cases that should fail start with fail-*. Janelle, Morgan, and Shirley worked on the test cases. The expected output for each testing file will have the same name as the testing file, but the extension is .out. testall.sh is a shell script taken from microc. For successful test cases, it compares the output file with the output achieved and for test cases supposed to fail, it compares the error achieved with the expected error in the corresponding .err file.

6.1 Motivation Behind Test Cases

We started off our testing by creating the hello world program, *test-basic.rac*. The hello world program simply prints out a variable which the string "hello world" is assigned to. There are no class declarations.

Then, we decided to create a test program that has only class declarations as a valid program. This program, *test-classdecls.rac*, contains two class declarations but no body statements and does not have any expected output. For each successful test case, we created a failing test case as well to ensure that the compiler was indeed searching for correct and valid syntax.

We then created a more complicated test case, *test-hello.rac*, that was a compilation of various features of Racontr. This included fibonacci, while loops, if else statements, class declarations, and printing a string. This not only ensured that each singular component worked on its own, but it also ensured the validity of the structure of our program and that one program is able to have multiple parts that function in different ways.

6.2 Example Test Programs

test-classdecls.rac

This program shows how having only class declarations still makes a valid program.

```
class Player1 extends Character {
     var name : string = "player1"
}

class Butler extends Scene {
     var name : string = "Butler Library"
}
```

Output of *test-classdecls.rac: No Output*

test-hello.rac

This program shows how different functionalities in Racontr can be successfully implemented in one program.

```
class Player1 extends Character {
     var name : string = "player1"
}
```

```
class Butler extends Scene {
     var name : string = "Butler Library"
}

var hello : string = "hello"
var world : string = "world"

var state : bool = false

var s : string = "aaa"

var one : i32 = 1
var two : i32 = 2
var one_bigger_two : bool = one > two

var count : i32 = 10
while count > 0 {
     print_int(count)
     count = count - 1
}

if one_bigger_two {
     s = hello
} else {
     s = world
}

print_string(s)
```

Output of *test-hello.rac:*

```
10
9
8
7
6
5
4
3
```

```
2
1
world
```

fail-helloworld.rac

This program contains the hello world program written incorrectly. It fails because our print function is print_string not print.

```
print("hello")
```

Error of *test-hello.rac: Fatal error: exception Not_found*

7. Language Evolution

7.1 Initial Thoughts

We first came up with the idea of making a text producing language since all of our team members are interested in text generators. Some of us wanted a story generator: a generator that preferably utilizes deep learning and natural language processing to write stories and poems according to the user's prompts. But after communicating with the professor and TAs, we decided to forego deep learning for now since it requires heavy workload but it is a feature we can consider to add in the future.

7.2 Narrowing Down the Scope

After deciding on making an interactive text editor, we started to explore what kind of stories we all enjoy and finally, decided on adventure stories. Soon after, we came up with a better idea, an adventure game editor: we have seen story generators, web-page based interactive text games, but not really a language designed for the general public to create text games. Moreover, some of us are really familiar with adventure text games, such as Dragon and Dungeon and Call of Cthulhu. Typically, in those kinds of games, there would be a "narrator" who tells the background story, some protagonists to unravel the truth of the story, some places for the protagonists to explore, and some villain for them to defeat in the end. Having this format in mind, we began to draw the blueprint of Racontr and started defining the prototypes.

7.3 Syntax Design Choices

Just as we mentioned in the last paragraph, inspired by Dragon and Dungeon and Call of Cthulhu, we defined two inbuilt classes: character and scene. We had initially planned for more classes like mission, achievement, and ending; however, we realized that they would be far too complicated for us to implement this semester. We decided to require all class declarations at the start of the program so the programmer can reference them later on. A character describes a player created or programmer created characters in the game: such as the protagonist that the player controls, or the villain that is pre-set by the programmer. Under the objects definition, we also have the character's name, type, description and more definitions to illustrate one instance. A scene, as we suggested above, describes a place that is usually interactable with the player created characters.

7.4 Design Conclusions

We still have many object types we wish to implement but have not, and features including natural language processing that we could not include for now. We built a simplified version of our initial design but we still believe in the potential and will develop on top of it with these ideas in mind. Moreover, for now our language only provides text-based interactions, but we can also use UNITY or UNREAL to create a visualized format for the user to interact with, simply by defining the data and function with our language. This could be complicated since it involved 3D modeling, but as we narrowed down our stories to adventure ones, visualization could be possible and should not be too difficult to achieve.

8. Lessons Learned

8.1 Janelle

I definitely learned a lot throughout the course of this project, be it through the mistakes or through the successes. The biggest lesson would be to be more considerate of implementation details from the start of the project because this makes our goals more realistic. Another lesson learned is to use the time zone difference to our advantage. Three group members are in EST and two are in China Standard Time (12 hours ahead of EST). Sometimes, such a drastic time zone difference allowed gaps in our communication because it was difficult to find a time when everyone was available. However, many times our group used this to our advantage by going to office hours that may have been at odd hours for other members.

Over the course of this project, I took on multiple roles whenever necessary and was not limited to my role designated at the start. I definitely pushed myself to be more confident in my skills during this project. I was initially intimidated by all of the moving parts, but soon I found myself and my teammates making progress simply by lots of trial and error and asking for help. Despite many of the setbacks, we soon found ourselves getting the hang of the different files and improving our debugging skills by understanding the flow of logic. My advice for future groups would be to set realistic goals, ask for help early on, communicate with your group every step of the way, and to not underestimate your own skills. Although this project has been incredibly challenging, it has been incredibly rewarding as well and I am grateful I was able to take this course.

8.2 Saumya

I learned how important it is to communicate and plan ahead in a group project. Since we were all in different time zones, it became difficult to find a meeting time that worked for all of us. We would have made all the deadlines if we would have communicated better.

I also realized how important it is to develop iteratively. Professor Edwards told us to do this from the beginning and I think this is something our group tried to follow as much as possible. Learning OCaml seemed impossible at first but we were fortunately able to get through the initial impediments. We realized how important it is to scale back and to focus on the most important features first. This project was truly a unique opportunity and although it was a lot of work, it was really rewarding!

8.3 Shirley

Two things that I learned from this experience is understanding the exact meaning of the code design and time management. There were several cases where we did not pay close enough attention to the code blocks of microc porgram and understand the role they played interacting with other parts of the language. And that resulted in a chain action of not properly coding the corresponding parts in Racontr. Though we planned to develop and test iteratively, it cost us way longer than expected to fully implement certain functions.

And beyond that, it probably would be a much pleasant experience if we could have taken into consideration the possibility of our program needing more time to develop and test. If we had started earlier on the actual programming, it would have given us more time later to adjust our language if we found aspects that were less preferable through the test cases.

8.4 Xinye

The first thing I think that I’ve learnt is how important for all the team members to come up with a good idea together. The project’s idea must interest all of us so we can share the motivation to work together; it also needs to be innovative and unique so our projects would have some real value even outside the class. We are pleased to think up an idea that we all agree on and put our time into advancing it.

Another important thing that I’ve learnt is how crucial making plans is for a group project. As we have iterated, all the team members live in different time zones and cooperation has not been as smooth as we had initially hoped to be, so we were required to make plans for the project’s progress so we could reach the checkpoints in time. Every step asks for cooperation and our cooperation requires planning ahead. This made me realize how important making plans are for a group project and at the same time, we all try hard to keep up with our plan and catch up when we cannot. Plans for the meetings… plans for the TA hours… and plans for the programming progress… Sometimes it can get tedious and demanding, but I think making plan greatly helps all of us to complete our project while working alone and working together.

8.5 Morgan

Over the course of the project, I learned not to be too ambitious with language features. As a group, we had many ideas about what a text-adventure language might look like but we quickly realized the amount of work required to implement even the most subtle features. I also learned the importance of time management, as the brainstorming and planning stages up a lot more time than anticipated in the development of the project because there were a lot of factors to consider. This saying has a lot of truth to it: sometimes less is more.

This project also pushed us to learn the details of functional programming and Ocaml, which I found both interesting and challenging. It is very different than the programming languages I have learned in previous courses, which made it especially rewarding when I started to understand. I ended up tapping into each of the roles, working on the system architecture and writing code for the files, writing a test case, and helping with the development of the language, so I learned how all of the moving parts work together. On a more personal level, doing the project under these virtual circumstances definitely posed new challenges, but I learned to have more confidence in my ability, adapt to frustrating situations, seek help when needed, and to be resilient. This project was truly a unique opportunity to explore functional programming by diving in and doing, which I appreciated and will apply to future thinking. For future students who take this course, my advice would be to start early, scale back, and channel your creativity!

9. Acknowledgements

There are many people who helped us in the creation of Racontr. First, we would like to thank Professor Edwards who was our project adviser. His office hours were incredibly helpful in making our goals more realistic and debugging our code when we were stuck.

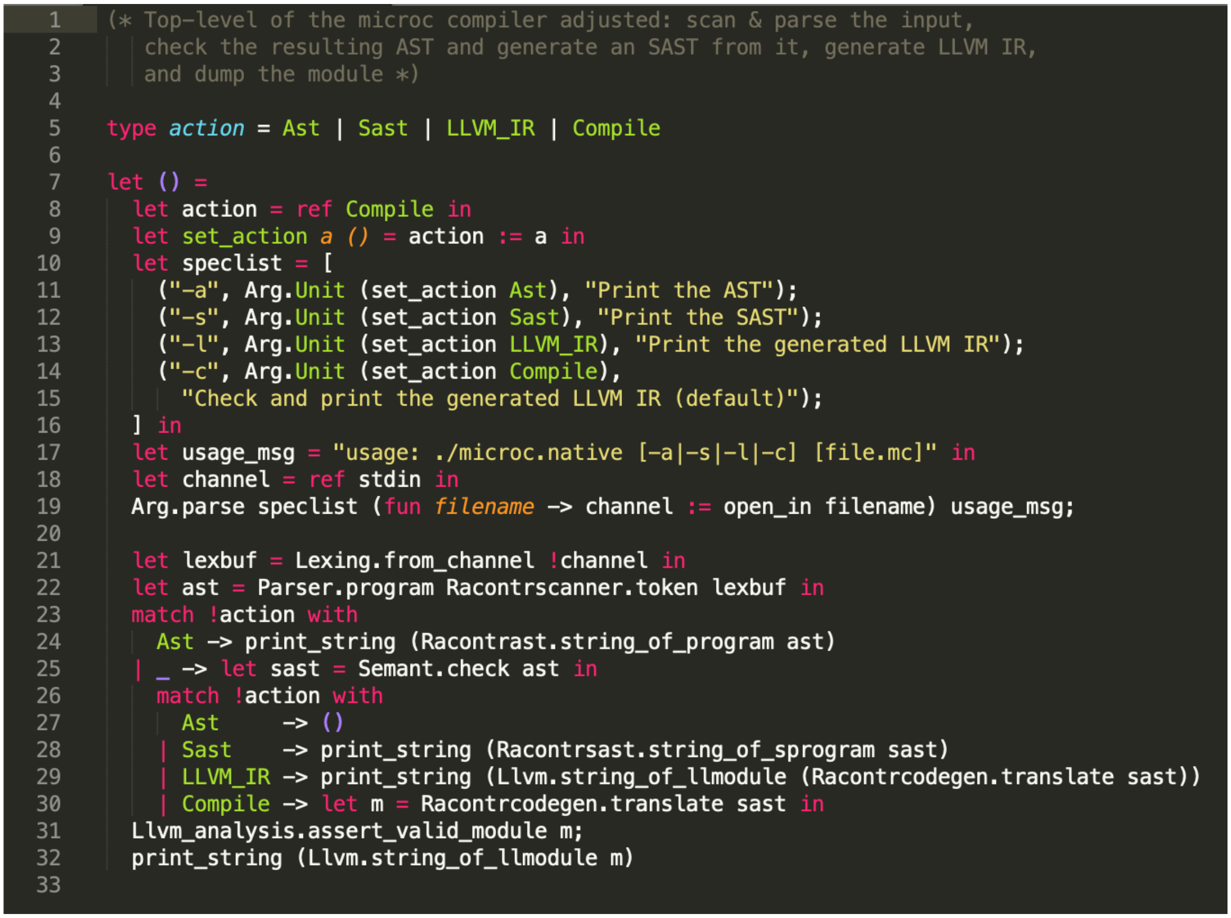
Languages that inspired Racontr include microc, GRIMM, Gawk, Coral, and Zil. We based many of our files, features, and goals off parts of these languages and customized them for Racontr.

We would also like to thank the AHOD project group, whose codegen we learned a lot from and then used to create ours. Specifically, we would like to give a shout out to Tiffeny from the AHOD group, who Professor Edwards introduced us to at office hours.

Special Shoutout to TA Xijiao who gave us hope--

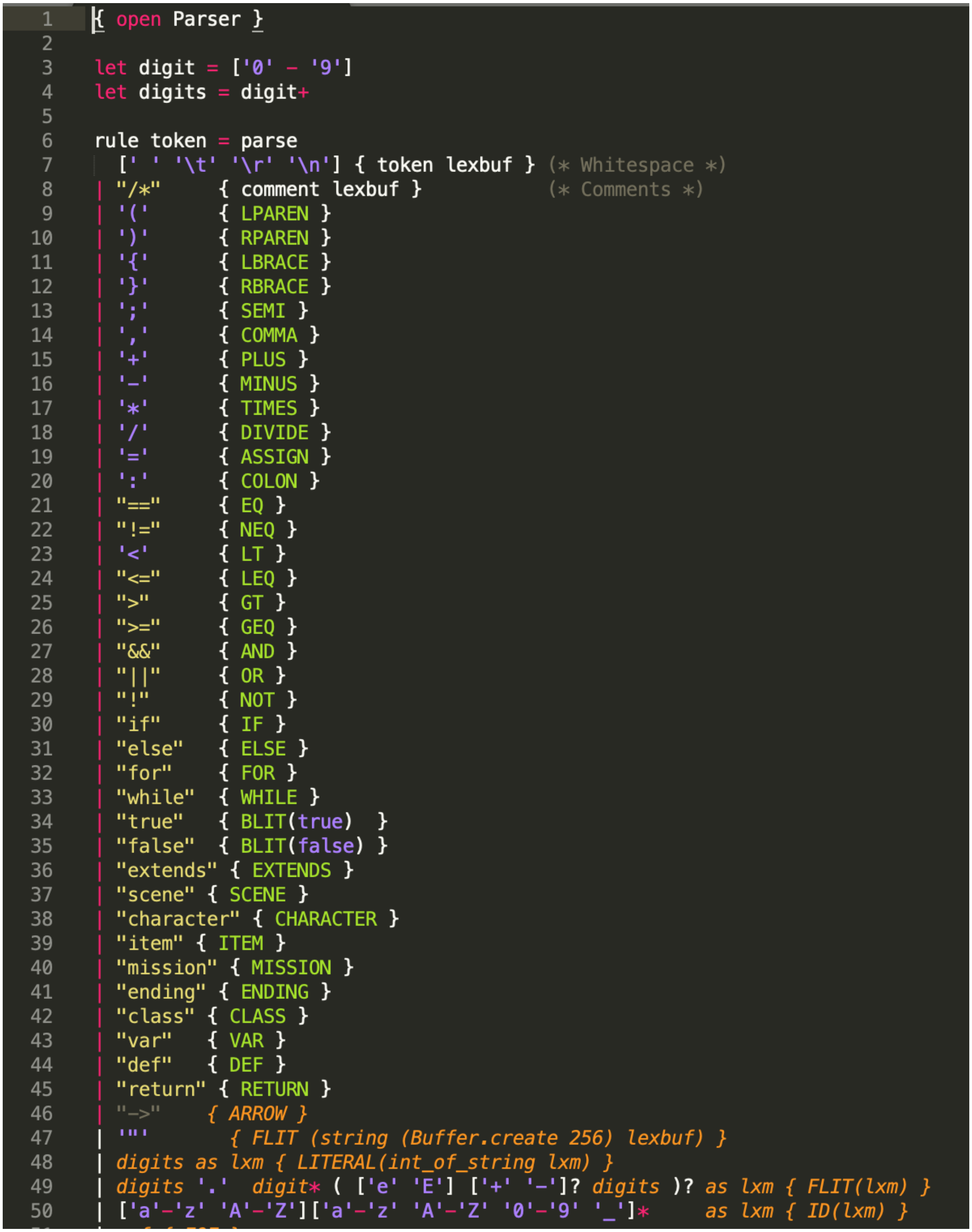
10. Appendix

10.1 Racontr.ml



```
(* Top-level of the microc compiler adjusted: scan & parse the input,
   check the resulting AST and generate an SAST from it, generate LLVM IR,
   and dump the module *)

type action = Ast | Sast | LLVM_IR | Compile

let () =
  let action = ref Compile in
  let set_action a () = action := a in
  let speclist = [
    ("-a", Arg.Unit (set_action Ast), "Print the AST");
    ("-s", Arg.Unit (set_action Sast), "Print the SAST");
    ("-l", Arg.Unit (set_action LLVM_IR), "Print the generated LLVM IR");
    ("-c", Arg.Unit (set_action Compile),
      "Check and print the generated LLVM IR (default)");
  ] in
  let usage_msg = "usage: ./microc.native [-a|-s|-l|-c] [file.mc]" in
  let channel = ref stdin in
  Arg.parse speclist (fun filename -> channel := open_in filename) usage_msg;

  let lexbuf = Lexing.from_channel !channel in
  let ast = Parser.program Racontrscanner.token lexbuf in
  match !action with
    Ast -> print_string (Racontrast.string_of_program ast)
  | _ -> let sast = Semant.check ast in
    match !action with
      Ast     -> ()
    | Sast    -> print_string (Racontrsast.string_of_sprogram sast)
    | LLVM_IR -> print_string (Llvm.string_of_llmodule (Racontrcodegen.translate sast))
    | Compile -> let m = Racontrcodegen.translate sast in
  Llvm_analysis.assert_valid_module m;
  print_string (Llvm.string_of_llmodule m)
```

10.2 scanner.mll



```
{ open Parser }

let digit = ['0' - '9']
let digits = digit+

rule token = parse
  [' ' '\t' '\r' '\n'] { token lexbuf } (* Whitespace *)
| "/*"     { comment lexbuf }          (* Comments *)
| '('      { LPAREN }
| ')'      { RPAREN }
| '{'      { LBRACE }
| '}'      { RBRACE }
| ';'      { SEMI }
| ','      { COMMA }
| '+'      { PLUS }
| '-'      { MINUS }
| '*'      { TIMES }
| '/'      { DIVIDE }
| '='      { ASSIGN }
| ':'      { COLON }
| "=="     { EQ }
| "!="     { NEQ }
| '<'      { LT }
| "<="     { LEQ }
| ">"      { GT }
| ">="     { GEQ }
| "&&"     { AND }
| "||"     { OR }
| "!"      { NOT }
| "if"     { IF }
| "else"   { ELSE }
| "for"    { FOR }
| "while"  { WHILE }
| "true"   { BLIT(true)  }
| "false"  { BLIT(false) }
| "extends" { EXTENDS }
| "scene" { SCENE }
| "character" { CHARACTER }
| "item" { ITEM }
| "mission" { MISSION }
| "ending" { ENDING }
| "class" { CLASS }
| "var"   { VAR }
| "def"   { DEF }
| "return" { RETURN }
| "->"    { ARROW }
| '"'       { FLIT (string (Buffer.create 256) lexbuf) }
| digits as lxm { LITERAL(int_of_string lxm) }
| digits '.'  digit* ( ['e' 'E'] ['+' '-']? digits )? as lxm { FLIT(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 }

and string buf = parse
  | '"'    { Buffer.contents buf }
  | _ as c { Buffer.add_char buf c; string buf lexbuf }

```

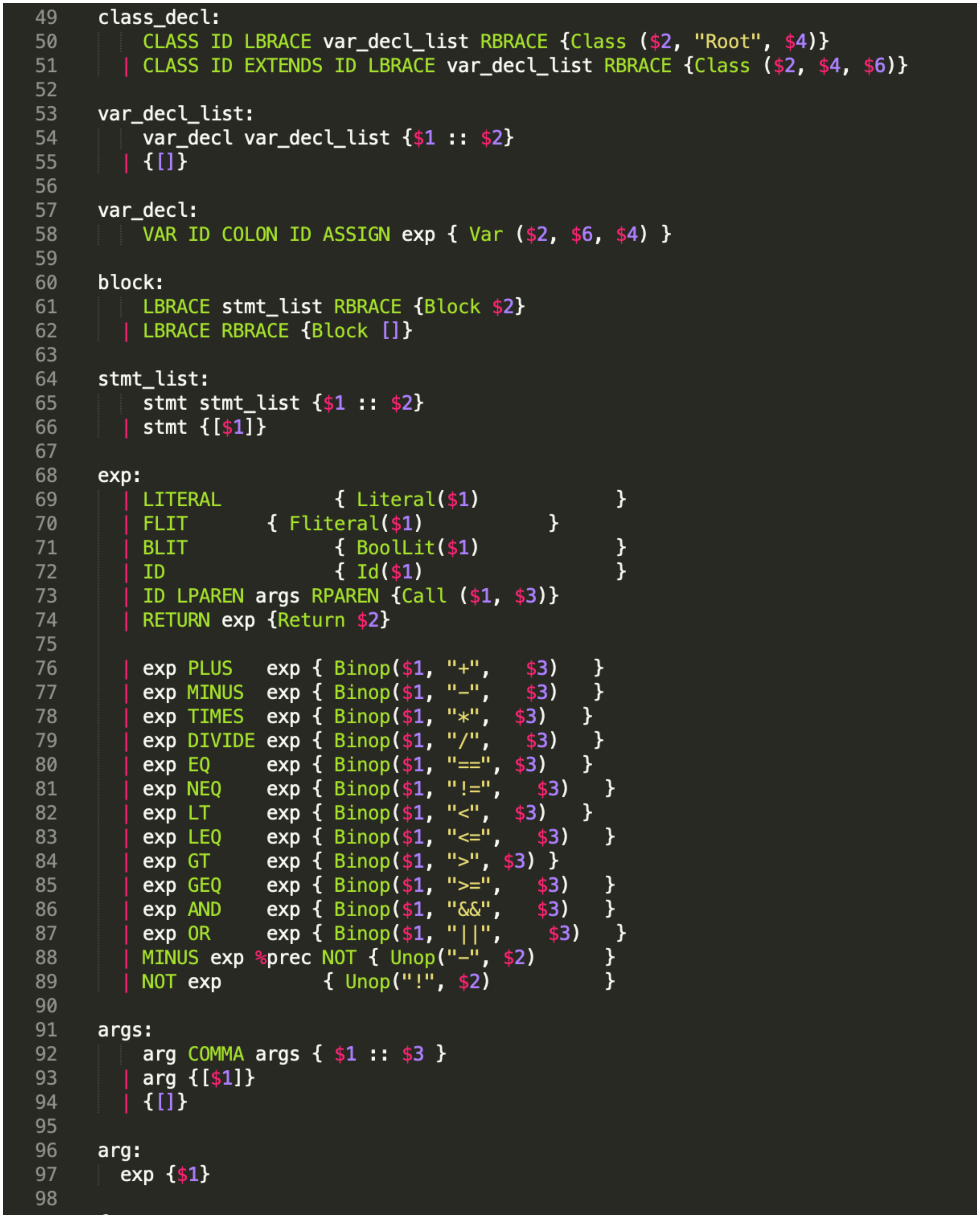
10.3 ast.ml

```
type id = string
type typ = string
type op = string

type formal_args = (id * typ) list

type stmt =
      Class of id * id * stmt list
    | Var of id * exp * typ
    | If of exp * stmt * stmt
    | While of exp * stmt
    | Block of stmt list
    | ExpStmt of exp
    | Assign of id * exp
    | Function of id * formal_args * typ * stmt
and exp =
      Literal of int
    | Fliteral of string
    | BoolLit of bool
    | Id of string
    | Call of id * exp list
    | Binop of exp * op * exp
    | Unop of op * exp
    | Return of exp


type program = stmt list

let string_of_expr = function
      Literal(l) -> string_of_int l
    | Fliteral(l) -> l
    | BoolLit(true) -> "true"
    | BoolLit(false) -> "false"
    | Id(s) -> s
```

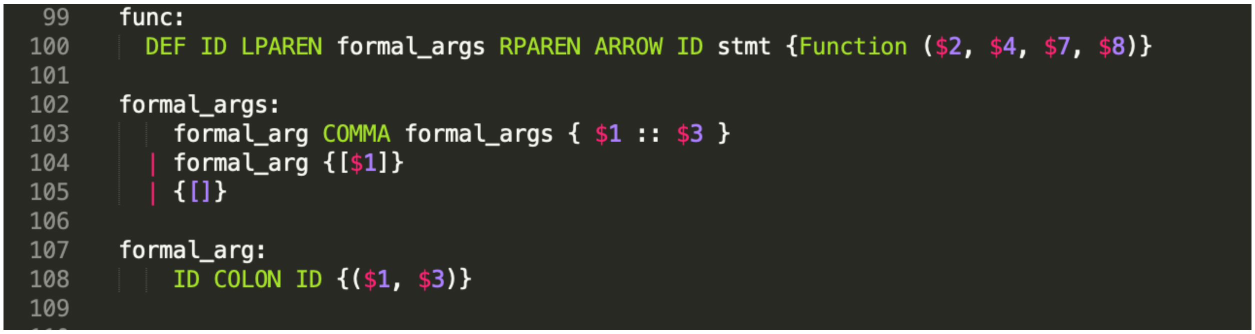
```
let rec string_of_exp exp =
  match exp with
  | Literal v -> string_of_int v
  | Fliteral s -> s
  | BoolLit v -> if v then "true" else "false"
  | Id id -> id
  | Call (f, arg::args) -> f ^ "(" ^
    List.fold_left (fun s e -> s ^ string_of_exp e) (string_of_exp arg) args
    ^ ")"
  | Call (f, []) -> f ^ "()"
  | Binop (e1, op, e2) ->
    string_of_exp e1 ^ op ^ string_of_exp e2
  | Unop (op, e) ->
    op ^ string_of_exp e
  | Return e -> "return " ^ string_of_exp e


let rec string_of_program ast =
  match ast with
  | [] -> ""
  | Class(name, super, stmts) :: rest ->
    "class: " ^ name ^ " <: " ^ super ^ " {\n"
      ^ string_of_program stmts ^
    "\n}\n"
    ^ (string_of_program rest)
  | Var(id, exp, typ) :: rest ->
    id ^ ":" ^ typ ^ " = " ^ (string_of_exp exp)  ^
    "\n"
    ^ (string_of_program rest)
  | If(e, t, f) :: rest ->
    "if (" ^ (string_of_exp e) ^ ")\n" ^
    (string_of_program [t]) ^ "else\n" ^
    (string_of_program [f]) ^ "\n"
    ^ (string_of_program rest)
  | Block stmts :: rest ->
    "{\n" ^ string_of_program stmts ^ "}\n"
    ^ (string_of_program rest)
  | ExpStmt e :: rest ->
    string_of_exp e
    ^ "\n"
    ^ (string_of_program rest)
  | Function (name, fargs, rettyp, stmt) :: rest ->
    "function " ^ name ^ "\n"
    ^ (string_of_program rest)
  | While (e, stmt) :: rest ->
    "while (" ^ string_of_exp e ^ ") {\n" ^
      (string_of_program [stmt]) ^
    "\n}" ^ "\n"
    ^ (string_of_program rest)
  | _ -> "not impl"

```

10.4 parser.mly

```

/* Ocamlyacc parser for Racontr */

%{
open Racontrast
%}

%token SEMI LPAREN RPAREN LBRACE RBRACE COMMA PLUS MINUS TIMES DIVIDE ASSIGN
%token NOT EQ NEQ LT LEQ GT GEQ AND OR IN
%token RETURN IF ELSE FOR WHILE
%token EXTENDS SCENE ITEM CHARACTER MISSION ENDING CLASS VAR COLON
%token DEF ARROW
%token <int> LITERAL
%token <bool> BLIT
%token <string> ID FLIT
%token EOF

%start program
%type <Racontrast.program> program

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

%%

program:
    stmt program {$1 :: $2}
  | EOF {[]}

stmt:
    class_decl {$1}
  | var_decl {$1}
  | IF exp stmt %prec NOELSE {If ($2, $3, Block [])}
  | IF exp stmt ELSE stmt {If ($2, $3, $5)}
  | WHILE exp stmt {While ($2, $3)}
  | exp {ExpStmt $1}
  | block {$1}
  | ID ASSIGN exp { Assign ($1, $3) }
  | func {$1}

```



```
class_decl:
    CLASS ID LBRACE var_decl_list RBRACE {Class ($2, "Root", $4)}
  | CLASS ID EXTENDS ID LBRACE var_decl_list RBRACE {Class ($2, $4, $6)}

var_decl_list:
    var_decl var_decl_list {$1 :: $2}
  | {[]}

var_decl:
    VAR ID COLON ID ASSIGN exp { Var ($2, $6, $4) }

block:
    LBRACE stmt_list RBRACE {Block $2}
  | LBRACE RBRACE {Block []}

stmt_list:
    stmt stmt_list {$1 :: $2}
  | stmt {[$1]}

exp:
  | LITERAL          { Literal($1)            }
  | FLIT       { Fliteral($1)           }
  | BLIT             { BoolLit($1)            }
  | ID               { Id($1)                 }
  | ID LPAREN args RPAREN {Call ($1, $3)}
  | RETURN exp {Return $2}

  | exp PLUS   exp { Binop($1, "+",   $3)   }
  | exp MINUS  exp { Binop($1, "-",   $3)   }
  | exp TIMES  exp { Binop($1, "*",  $3)   }
  | exp DIVIDE exp { Binop($1, "/",   $3)   }
  | exp EQ     exp { Binop($1, "==", $3)   }
  | exp NEQ    exp { Binop($1, "!=",   $3)   }
  | exp LT     exp { Binop($1, "<",  $3)   }
  | exp LEQ    exp { Binop($1, "<=",   $3)   }
  | exp GT     exp { Binop($1, ">", $3) }
  | exp GEQ    exp { Binop($1, ">=",   $3)   }
  | exp AND    exp { Binop($1, "&&",   $3)   }
  | exp OR     exp { Binop($1, "||",    $3)   }
  | MINUS exp %prec NOT { Unop("-", $2)      }
  | NOT exp         { Unop("!", $2)          }

args:
    arg COMMA args { $1 :: $3 }
  | arg {[$1]}
  | {[]}

arg:
  exp {$1}
```



```
func:
  DEF ID LPAREN formal_args RPAREN ARROW ID stmt {Function ($2, $4, $7, $8)}

formal_args:
    formal_arg COMMA formal_args { $1 :: $3 }
  | formal_arg {[$1]}
  | {[]}

formal_arg:
    ID COLON ID {($1, $3)}

```

10.5 sast.ml

```
open Racontrast

type sexpr = typ * sx
and sx =
    SLiteral of int
  | SFliteral of string
  | SBoolLit of bool
  | SId of string
  | SPrint of sexpr list
  | SNoexpr

type sstmt =
    SExpr of sexpr
  | SWhile of sexpr * sstmt

type sstatement = {
  styp : typ;
  sfname : string;
}

type sprogram = sstatement list
(*might need to add semantic check to class decl later *)


let string_of_sprogram sast = "not impl"
```

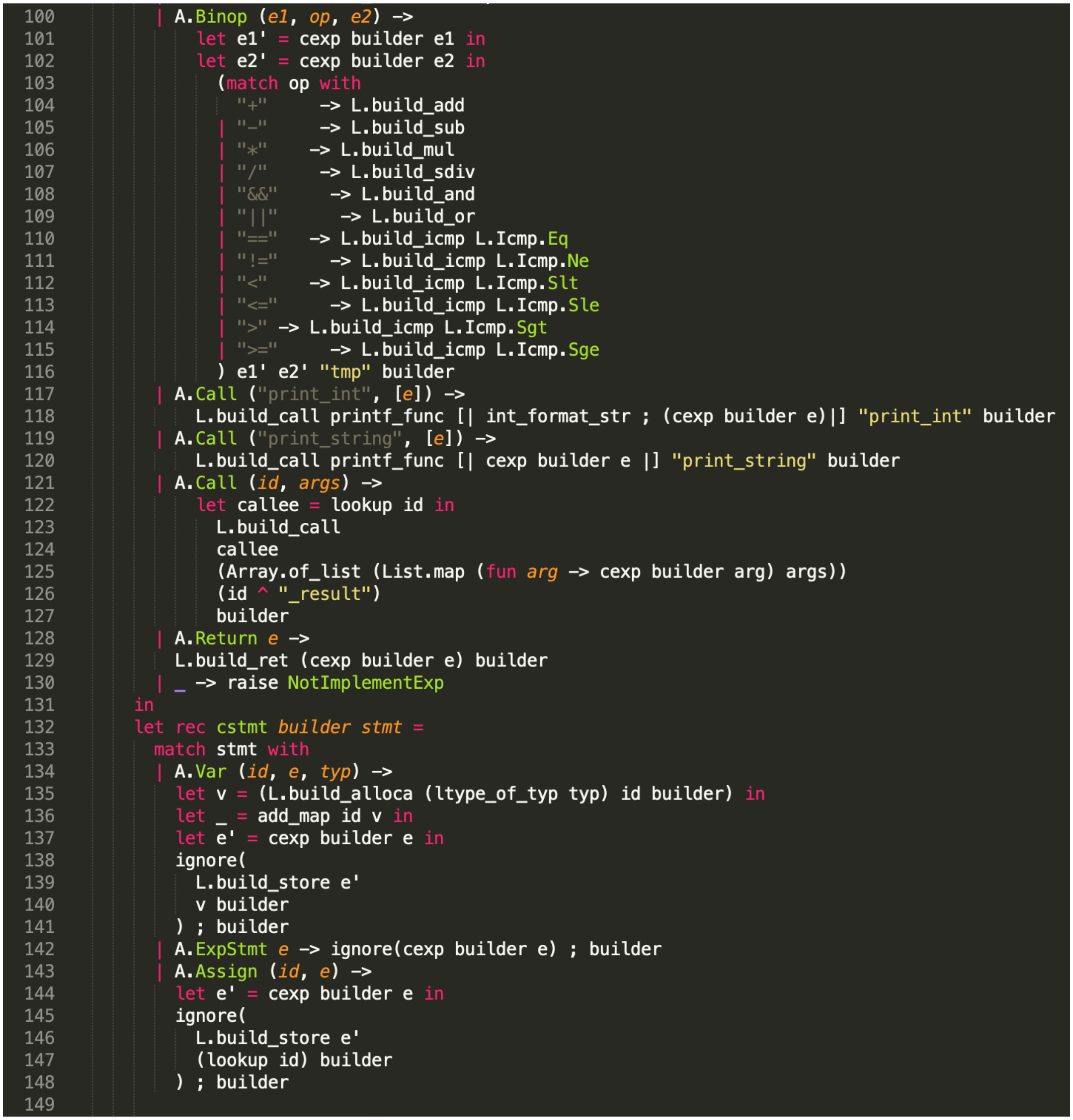
10.6 semant.ml

```
open Racontrast
open Racontrsast

module StringMap = Map.Make(String)

let check prog = prog

  (* let built_in_decls =
    let add_bind map (name, ty) = StringMap.add name {
      typ = ty;
      fname = name;
      } map
    in List.fold_left add_bind StringMap.empty [ ("print", String) ]
  in

    let rec expr = function
        Literal  l -> (Int, SLiteral l)
      | Fliteral l -> (String, SFliteral l)
      | BoolLit l  -> (Bool, SBoolLit l)
      | Noexpr     -> (Void, SNoexpr)
      | Id s       -> (String, SId s)
      | Print(args) as print -> (Void, SPrint args)
    in

    let check_bool_expr e =
      let (t', e') = expr e
      and err = "expected Boolean expression in " ^ string_of_expr e
      in if t' != Bool then raise (Failure err) else (t', e')
    in

    let rec check_stmt = function
        Expr e -> SExpr (expr e)
      | While(p, s) -> SWhile(check_bool_expr p, check_stmt s)

  (*| SBlock(sl) -> sl
      | _ -> raise (Failure ("internal error: block didn't become a block?")) *)
    in
    (check_stmt class_declarations, [])  *)

```

10.7 codegen.ml

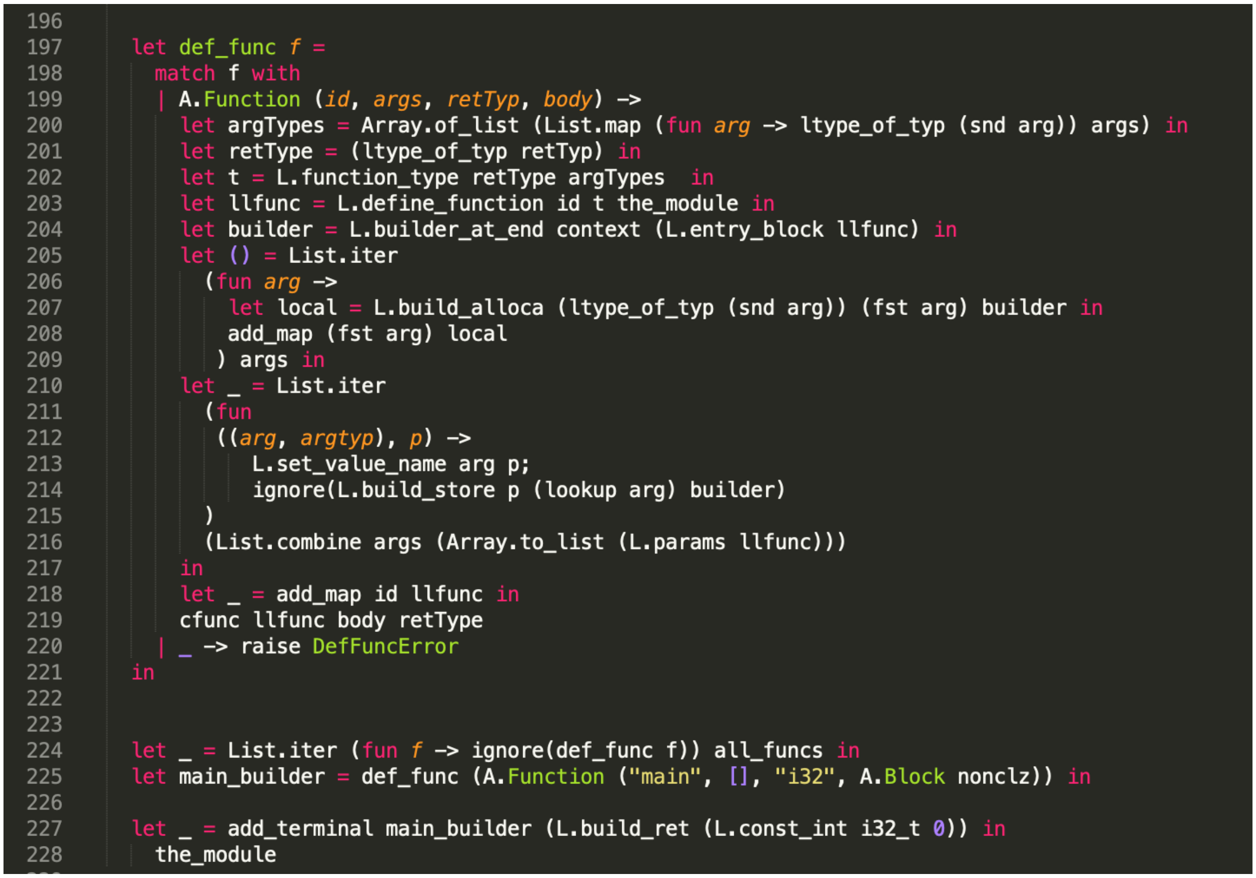
```
module L = Llvm
module A = Racontrast
(* open Racontrsast *)
module StringMap = Map.Make(String)

exception NotImplementExp
exception NotImplementStmt
exception DefFuncError

let rec get_clzs prog =
  match prog with
  | [] -> []
  | (A.Class _) :: rest -> (List.hd prog) :: (get_clzs rest)
  | _ :: rest -> get_clzs rest


let rec get_noclz prog =
  match prog with
  | [] -> []
  | A.Class _ :: rest -> get_noclz rest
  | A.Function _ :: rest -> get_noclz rest
  | _ :: rest -> (List.hd prog) :: (get_noclz rest)

let rec get_funcs prog =
  match prog with
  | [] -> []
  | A.Function _ :: rest -> (List.hd prog) :: (get_funcs rest)
  | _ :: rest -> get_funcs rest




exception TypNotImpl

let translate prog =
  let context = L.global_context() in
  let the_module = L.create_module context "Racontr" in

  (* Get types from the context *)
  let i32_t      = L.i32_type    context
  and i8_t       = L.i8_type     context
  and i1_t       = L.i1_type     context
  and float_t    = L.double_type context
  and string_t   = L.pointer_type (L.i8_type context)
  and void_t     = L.void_type   context in

```

```
let ltype_of_typ = function
    "i32"   -> i32_t
  | "string"  -> string_t
  | "bool" -> i1_t
  | "unit" -> void_t
  | _ -> raise TypNotImpl
in



let var_map = ref StringMap.empty in
let add_map n v = var_map := StringMap.add n v (!var_map) in

let lookup n =
  StringMap.find n !var_map in



let printf_t : L.lltype =
  L.var_arg_function_type i32_t [| L.pointer_type i8_t |]
in

let printf_func : L.llvalue =
  L.declare_function "printf" printf_t the_module
in

let clzs = get_clzs prog in
let nonclz = get_noclz prog in




let add_terminal builder instr =
  match L.block_terminator (L.insertion_block builder) with
    Some _ -> ()
  | None -> ignore (instr builder) in


let all_funcs = get_funcs prog in

let cfunc main_func body retType =
  let main_builder = L.builder_at_end context (L.entry_block main_func) in
  let int_format_str = L.build_global_stringptr "%d\n" "fmt" main_builder
  and float_format_str = L.build_global_stringptr "%g\n" "fmt" main_builder in
  let rec cexp builder e =
    match e with
    | A.Literal v -> L.const_int i32_t v
    | A.Fliteral s -> L.build_global_stringptr s s builder
    | A.BoolLit b -> L.const_int i1_t (if b then 1 else 0)
    | A.Id id -> L.build_load (lookup id) id builder
```



```
        | A.Binop (e1, op, e2) ->
            let e1' = cexp builder e1 in
            let e2' = cexp builder e2 in
              (match op with
                "+"     -> L.build_add
              | "-"     -> L.build_sub
              | "*"    -> L.build_mul
              | "/"     -> L.build_sdiv
              | "&&"     -> L.build_and
              | "||"      -> L.build_or
              | "=="   -> L.build_icmp L.Icmp.Eq
              | "!="     -> L.build_icmp L.Icmp.Ne
              | "<"    -> L.build_icmp L.Icmp.Slt
              | "<="     -> L.build_icmp L.Icmp.Sle
              | ">" -> L.build_icmp L.Icmp.Sgt
              | ">="     -> L.build_icmp L.Icmp.Sge
              ) e1' e2' "tmp" builder
        | A.Call ("print_int", [e]) ->
            L.build_call printf_func [| int_format_str ; (cexp builder e)|] "print_int" builder
        | A.Call ("print_string", [e]) ->
            L.build_call printf_func [| cexp builder e |] "print_string" builder
        | A.Call (id, args) ->
            let callee = lookup id in
              L.build_call
              callee
              (Array.of_list (List.map (fun arg -> cexp builder arg) args))
              (id ^ "_result")
              builder
        | A.Return e ->
          L.build_ret (cexp builder e) builder
        | _ -> raise NotImplementExp
      in
      let rec cstmt builder stmt =
        match stmt with
        | A.Var (id, e, typ) ->
          let v = (L.build_alloca (ltype_of_typ typ) id builder) in
          let _ = add_map id v in
          let e' = cexp builder e in
          ignore(
            L.build_store e'
            v builder
          ) ; builder
        | A.ExpStmt e -> ignore(cexp builder e) ; builder
        | A.Assign (id, e) ->
          let e' = cexp builder e in
          ignore(
            L.build_store e'
            (lookup id) builder
          ) ; builder

```

```
      | A.If (predicate, t, f) ->
          let bool_val = cexp builder predicate in
          let merge_bb = L.append_block context "merge" main_func in
          let build_br_merge = L.build_br merge_bb in (* partial function *)

          let then_bb = L.append_block context "then" main_func in
            add_terminal (cstmt (L.builder_at_end context then_bb) t)
              build_br_merge;

          let else_bb = L.append_block context "else" main_func in
            add_terminal (cstmt (L.builder_at_end context else_bb) f)
              build_br_merge;

          ignore(L.build_cond_br bool_val then_bb else_bb builder);
          L.builder_at_end context merge_bb
      | A.While (e, stmt) ->
          let pred_bb = L.append_block context "while" main_func in
          ignore(L.build_br pred_bb builder);

          let body_bb = L.append_block context "while_body" main_func in
          add_terminal (cstmt (L.builder_at_end context body_bb) stmt)
            (L.build_br pred_bb);

          let pred_builder = L.builder_at_end context pred_bb in
          let bool_val = cexp pred_builder e in

          let merge_bb = L.append_block context "merge" main_func in
          ignore(L.build_cond_br bool_val body_bb merge_bb pred_builder);
          L.builder_at_end context merge_bb

      | A.Block ss ->
          List.fold_left (fun builder s -> cstmt builder s) builder ss

      | _ -> raise NotImplementStmt
    in
      match body with
      | A.Block body ->
        let builder = List.fold_left
          (fun builder stmt -> cstmt builder stmt)
          main_builder
          body
        in
        builder

      | _ -> raise DefFuncError
  in
```



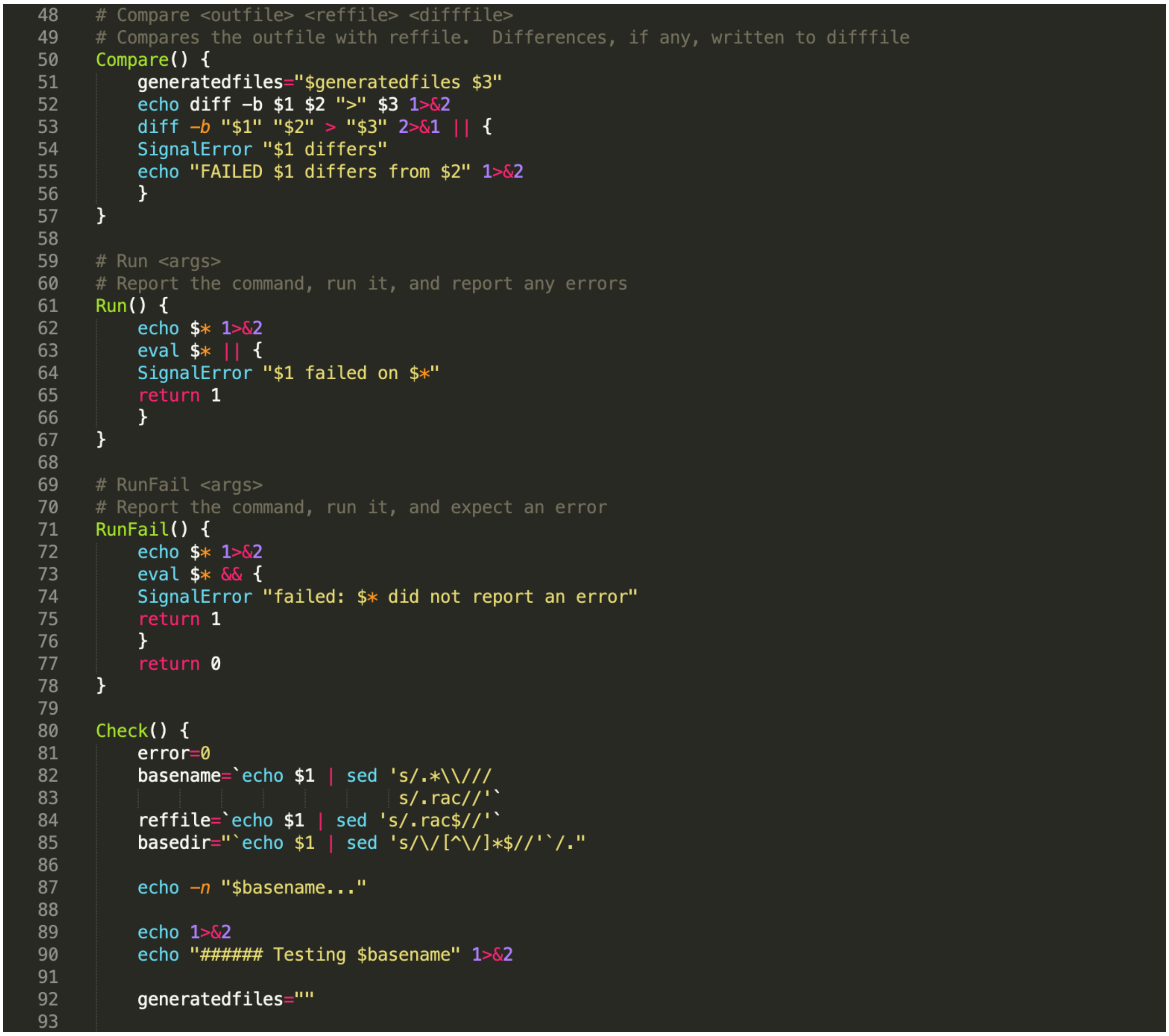
```
  let def_func f =
    match f with
    | A.Function (id, args, retTyp, body) ->
      let argTypes = Array.of_list (List.map (fun arg -> ltype_of_typ (snd arg)) args) in
      let retType = (ltype_of_typ retTyp) in
      let t = L.function_type retType argTypes  in
      let llfunc = L.define_function id t the_module in
      let builder = L.builder_at_end context (L.entry_block llfunc) in
      let () = List.iter
        (fun arg ->
          let local = L.build_alloca (ltype_of_typ (snd arg)) (fst arg) builder in
          add_map (fst arg) local
         ) args in
      let _ = List.iter
        (fun
         ((arg, argtyp), p) ->
            L.set_value_name arg p;
            ignore(L.build_store p (lookup arg) builder)
        )
        (List.combine args (Array.to_list (L.params llfunc)))
      in
      let _ = add_map id llfunc in
      cfunc llfunc body retType
    | _ -> raise DefFuncError
  in


  let _ = List.iter (fun f -> ignore(def_func f)) all_funcs in
  let main_builder = def_func (A.Function ("main", [], "i32", A.Block nonclz)) in

  let _ = add_terminal main_builder (L.build_ret (L.const_int i32_t 0)) in
    the_module
```

10.8 Makefile

```
.PHONY : test
test : all testall.sh
    ./testall.sh

.PHONY : all
all : racontr.native

racontr.native :
    opam config exec -- \
    ocamlbuild -use-ocamlfind racontr.native

.PHONY : clean
clean :
    ocamlbuild -clean
    rm -rf testall.log ocamlllvm *.diff

TARFILES = racontrast.ml racontrsast.ml racontrcodegen.ml Makefile _tags racontr.ml parser.mly \
    README racontrscanner.mll semant.ml testall.sh \
    Dockerfile \

racontr.tar.gz : $(TARFILES)
    cd .. && tar czf racontr/racontr.tar.gz \
        $(TARFILES:%=racontr/%)

```

10.9 testall.sh

```sh
#!/bin/sh

# Regression testing script for MicroC
# Step through a list of files
#  Compile, run, and check the output of each expected-to-work test
#  Compile and check the error of each expected-to-fail test

# Path to the LLVM interpreter
LLI="lli"
#LLI="/usr/local/opt/llvm/bin/lli"

# Path to the LLVM compiler
LLC="llc"

# Path to the C compiler
CC="cc"

# Path to the microc compiler.  Usually "./microc.native"
# Try "_build/microc.native" if ocamlbuild was unable to create a symbolic link.
RACONTR="./racontr.native"
#MICROC="_build/microc.native"

# Set time limit for all operations
ulimit -t 30

globallog=testall.log
rm -f $globallog
error=0
globalerror=0

keep=0

Usage() {
    echo "Usage: testall.sh [options] [.rac files]"
    echo "-k    Keep intermediate files"
    echo "-h    Print this help"
    exit 1
}

SignalError() {
    if [ $error -eq 0 ] ; then
    echo "FAILED"
    error=1
    fi
    echo "  $1"
}

```



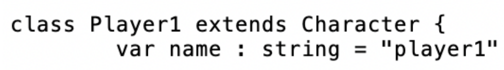
```
# Compare <outfile> <reffile> <difffile>
# Compares the outfile with reffile.  Differences, if any, written to difffile
Compare() {
    generatedfiles="$generatedfiles $3"
    echo diff -b $1 $2 ">" $3 1>&2
    diff -b "$1" "$2" > "$3" 2>&1 || {
    SignalError "$1 differs"
    echo "FAILED $1 differs from $2" 1>&2
    }
}

# Run <args>
# Report the command, run it, and report any errors
Run() {
    echo $* 1>&2
    eval $* || {
    SignalError "$1 failed on $*"
    return 1
    }
}

# RunFail <args>
# Report the command, run it, and expect an error
RunFail() {
    echo $* 1>&2
    eval $* && {
    SignalError "failed: $* did not report an error"
    return 1
    }
    return 0
}

Check() {
    error=0
    basename=`echo $1 | sed 's/.*\\///
                             s/.rac//'`
    reffile=`echo $1 | sed 's/.rac$//'`
    basedir="`echo $1 | sed 's/\/[^\/]*$//'`/."

    echo -n "$basename..."

    echo 1>&2
    echo "###### Testing $basename" 1>&2

    generatedfiles=""

```

```
    generatedfiles="$generatedfiles ${basename}.ll ${basename}.s ${basename}.exe ${basename}.out" &&
    Run "$RACONTR" "$1" ">" "${basename}.ll" &&
    Run "$LLC" "-relocation-model=pic" "${basename}.ll" ">" "${basename}.s" &&
    Run "$CC" "-o" "${basename}.exe" "${basename}.s" &&
    Run "./${basename}.exe" > "${basename}.out" &&
    Compare ${basename}.out ${reffile}.out ${basename}.diff

    # Report the status and clean up the generated files

    if [ $error -eq 0 ] ; then
    if [ $keep -eq 0 ] ; then
        rm -f $generatedfiles
    fi
    echo "OK"
    echo "###### SUCCESS" 1>&2
    else
    echo "###### FAILED" 1>&2
    globalerror=$error
    fi
}

CheckFail() {
    error=0
    basename=`echo $1 | sed 's/.*\\///
                             s/.rac//'`
    reffile=`echo $1 | sed 's/.rac$//'`
    basedir="`echo $1 | sed 's/\/[^\/]*$//'`/."

    echo -n "$basename..."

    echo 1>&2
    echo "###### Testing $basename" 1>&2

    generatedfiles=""

    generatedfiles="$generatedfiles ${basename}.err ${basename}.diff" &&
    RunFail "$RACONTR" "<" $1 "2>" "${basename}.err" ">>" $globallog &&
    Compare ${basename}.err ${reffile}.err ${basename}.diff

```

```
    if [ $error -eq 0 ] ; then
    if [ $keep -eq 0 ] ; then
        rm -f $generatedfiles
    fi
    echo "OK"
    echo "###### SUCCESS" 1>&2
    else
    echo "###### FAILED" 1>&2
    globalerror=$error
    fi
}

while getopts kdpsh c; do
    case $c in
    k) # Keep intermediate files
        keep=1
        ;;
    h) # Help
        Usage
        ;;
    esac
done

shift `expr $OPTIND - 1`

if [ $# -ge 1 ]
then
    files=$@
else
    files="tests/test-*.rac tests/fail-*.rac"
fi

for file in $files
do
    case $file in
    *test-*)
        Check $file 2>> $globallog
        ;;
    *fail-*)
        CheckFail $file 2>> $globallog
        ;;
    *)
        echo "unknown file type $file"
        globalerror=1
        ;;
    esac
done

exit $globalerror
```

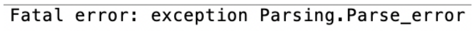
10.10 Tests

fail-classdecls.rac



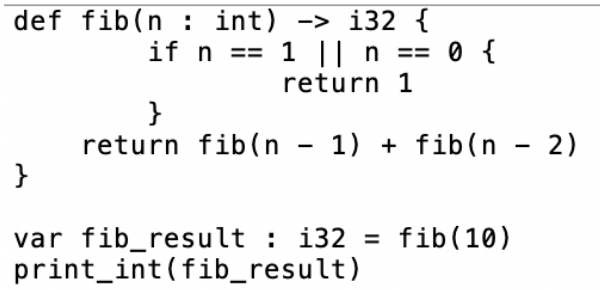
```
class Player1 extends Character {
        var name : string = "player1"
```

fail-classdecls.err



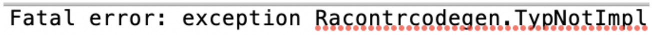
```
Fatal error: exception Parsing.Parse_error
```

fail-fib.rac



```
def fib(n : int) -> i32 {
        if n == 1 || n == 0 {
                return 1
        }
    return fib(n - 1) + fib(n - 2)
}

var fib_result : i32 = fib(10)
print_int(fib_result)
```

fail-fib.err



```
Fatal error: exception Racontrcodegen.TypNotImpl
```

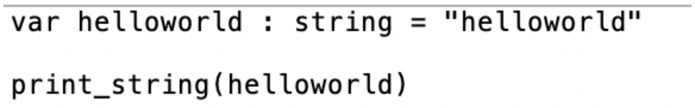
fail-helloworld.rac

```
print("hello")
```

fail-helloworld.rac

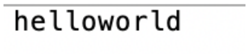
```
Fatal error: exception Not_found
```

test-basic.rac



```
var helloworld : string = "helloworld"

print_string(helloworld)
```

test-basic.out



```
helloworld
```

test-classdecls.rac

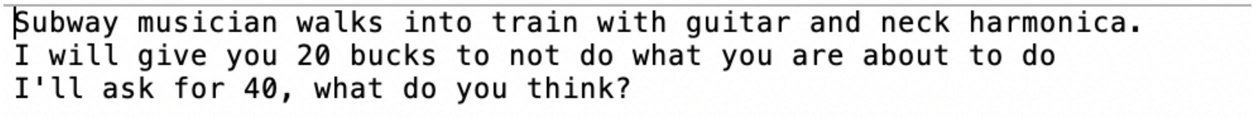
test-classdecls.out

*Empty file, doesn’t print anything

test-convo.rac

```
class Player1 {
        var name: string = "Tom"
        var line1: string = "I will give you 20 bucks to not do what you are about to do"
        var location: string = "110th street"
}

class Player2 {
        var name: string = "Jerry"
        var line1: string = "I'll ask for 40, what do you think?"
        var description: string = “Subway musician walks into train with guitar and neck
harmonica.”
}

print_string(Player2.description)
print_string(Player1.line1)
print_string(Player2.line1)
```

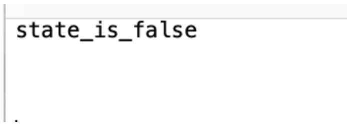
test-convo.out



```
Subway musician walks into train with guitar and neck harmonica.
I will give you 20 bucks to not do what you are about to do
I'll ask for 40, what do you think?
```

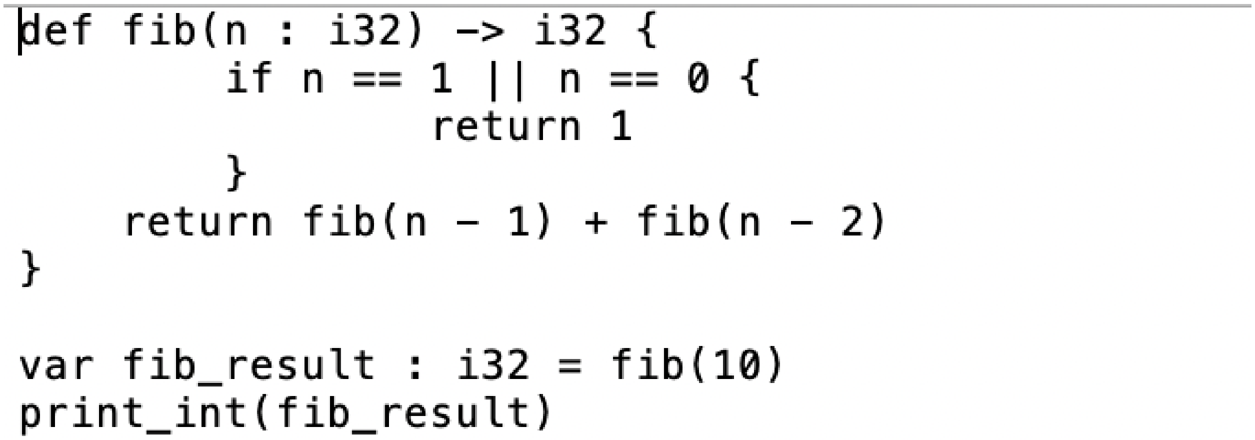
test-printbool.rac

test-printbool.out



```
state_is_false
```

test-fib.rac

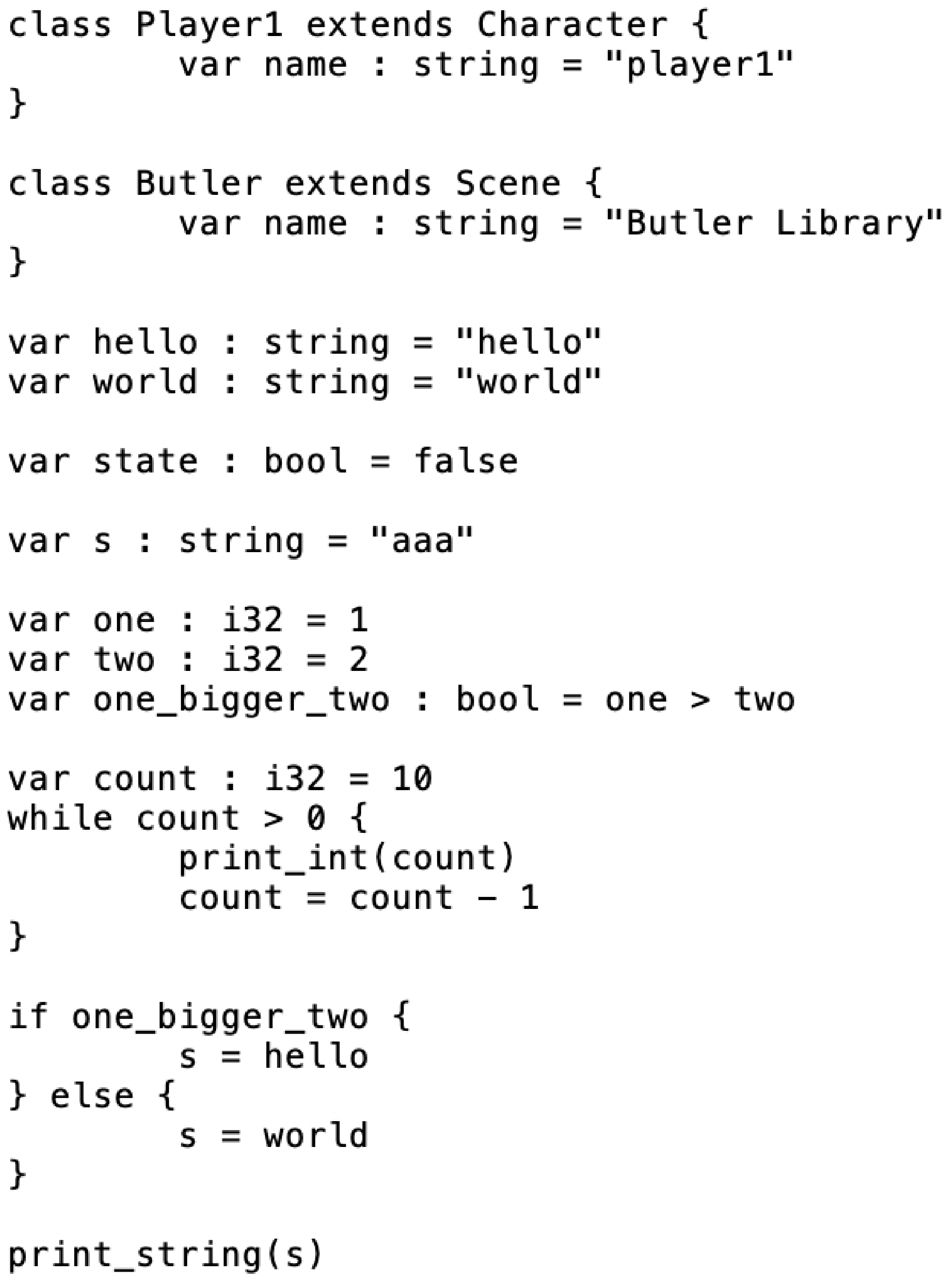


```
def fib(n : i32) -> i32 {
        if n == 1 || n == 0 {
                return 1
        }
    return fib(n - 1) + fib(n - 2)
}

var fib_result : i32 = fib(10)
print_int(fib_result)
```

test-fib.out

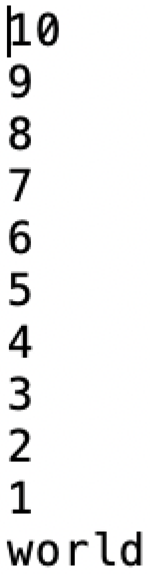
```
89
```

test-hello.rac



```
class Player1 extends Character {
        var name : string = "player1"
}

class Butler extends Scene {
        var name : string = "Butler Library"
}

var hello : string = "hello"
var world : string = "world"

var state : bool = false

var s : string = "aaa"

var one : i32 = 1
var two : i32 = 2
var one_bigger_two : bool = one > two

var count : i32 = 10
while count > 0 {
        print_int(count)
        count = count - 1
}

if one_bigger_two {
        s = hello
} else {
        s = world
}

print_string(s)
```

test-hello.out



```
10
9
8
7
6
5
4
3
2
1
world
```