

Racecar

A Programming Language for Kids



May 12, 2013

Team 19

Project Manager
Language Guru
System Architect
System Integrator
Verification and Validation
Mentor
Professor

Samuel Kohn
Alexander Fields
Jeremy Spencer
Mason Silber
Colfax Selby
Melanie Kambadur
Alfred Aho

Contents

1	Introduction	3
1.1	What Problem Does Racecar Solve?	3
1.2	Who Should Use Racecar?	3
1.3	Properties of Racecar	4
1.4	Similar Programming Languages	5
2	Lesson Plans	6
2.1	Lesson 1: How to Drive	8
2.2	Lesson 2: Comments	8
2.3	Lesson 3: Turning the Car	10
2.4	Lesson 4: More Turning	10
2.5	Lesson 5: Conditionals	11
2.6	Lesson 6: Variables	13
2.7	Lesson 7: Loops Part 1	16
2.8	Lesson 8: Loops Part 2	18
2.9	Lesson 9: Subroutines	19
2.10	Lesson 10: Variables in Subroutines	21
3	Language Reference Manual	23
3.1	Introduction	23
3.2	Identifiers and Scope	23
3.3	Data Types	24
3.4	Statements	24
3.5	Comments	25
3.6	Expressions	25
3.7	Control Flow	25
3.8	Built-in Functions	26
3.9	User-defined Functions	27
3.10	Reserved Words	28
3.11	Formal Grammar	28
4	Project Plan	31
4.1	Team Roles	31
4.2	Development Process	32
4.3	Project Schedule and Log	32
4.4	Implementation Style	33

5	Language Evolution	33
6	Translator Architecture	34
7	Development and Run-Time Environment	36
8	Test Plan	37
8.1	Segmented Testing	37
8.2	Iterative Testing	38
9	Conclusions	38
9.1	Lessons Learned as a Team	38
9.2	Lessons Learned by Each Team Member	39
9.3	Advice for Future Teams	41
9.4	Suggestions for the Instructor	41
10	Appendix	42
10.1	Launch Script	42
10.2	Graphical Interface	42
10.3	Lexer and Parser	66
10.4	Semantic Analyzer and Symbol Table	80
10.5	Translator	91

1 Introduction

Technology education for elementary school students is in its infancy. Teachers are put in the unenviable position of trying to teach children how to harness the potential of computers without actually knowing how computers work and how to write computer programs. As a consequence, many students are never exposed to computer programming and the algorithmic critical thinking style that is critical to writing programs as well as solving many other problems in life. Racecar is designed to solve this problem by providing a language that is *easy to teach*, even for non-experts; *readable*, so parents can easily involve themselves in their children’s education; and *engaging* for 8 to 10-year-old children so they are motivated to experiment and learn more about computers and programming, even outside of school.

In order to capture and maintain children’s interest in programming, Racecar is designed around a single goal: to write a program that will navigate a car through an obstacle course. Students will learn how to write programs that tell the car to move and turn in specific sequences, a process that allows students to think about concrete objects—an essential requirement for a language designed for 8 to 10-year-olds—while they solve a prototypical problem of the algorithmic style of thinking. More importantly, the programs students write can be run using the accompanying application, which shows the car as it executes the program’s instructions and navigates the obstacles. This immediate visual feedback is essential for keeping students on task and excited about their progress.

1.1 What Problem Does Racecar Solve?

Existing attempts to teach children about computer programming generally utilize languages from one of three categories: graphical “drag-and-drop” programming, simplistic text programming (with graphical output), and conventional programming languages. These various ideas each have their drawbacks, and Racecar is designed to improve on all of the positive aspects while minimizing the effects of the problems with these techniques. In general, these languages fall short in one of three categories: similarity to real-world programming (e.g. program is not a text file), readability for non-experts, and a level of engagement that captures children’s interest. Racecar is designed with these properties in mind, with the goal of making computer programming extremely easy to teach in schools.

1.2 Who Should Use Racecar?

The intended users of this language are elementary school children around the ages of 8 to 10, and their educators. Given that the purpose of Racecar is to introduce these children to programming, no previous experience is necessary. It is designed to be accessible and engaging to children of all different interests and backgrounds. The scope of the language is relatively small—it is clearly

domain-specific—and it is easy for a non-technical adult or teacher to pick up Racecar as well; any elementary school teacher should be able to learn the language quickly and to teach it effectively to others. Lastly, even children’s parents could learn how to write, or at least read, Racecar programs to help on homeworks or independent projects if necessary.

1.3 Properties of Racecar

Easy to Teach

Racecar is syntactically easy to understand so that instructors with minimal knowledge of computer science concepts will be able to teach the accompanying lessons and debug students’ programs quickly. The lessons included with the language tutorial build on each other, showing students that a complex task can be accomplished by breaking the problem into manageable parts. Each lesson adds a new movement (including “drive straight” and “turn”) or programming concept (including subroutines and looping/iteration) that can be integrated into the previous lesson’s program.

Readable

One of the biggest problems in the computer world is readability. At every step of a computer science education, teachers and professors beseech students to include whitespace, avoid long chains of function calls, and comment as often as possible. However, it still seems that people write indecipherable code that even experts have trouble understanding. Language designers have tried to remedy this problem proactively by writing “readable” languages. Existing readable languages include COBOL and Python. Both contain syntactic constructs that are useful for programmers, yet degrade readability. If you thought `OCCURS 12 TIMES` means “loop/repeat 12 times” in COBOL, you’d be wrong—it’s a declaration for a 12-element array! In Python (a vast improvement from COBOL), many keywords and functions such as `def`, `len`, and `str` are short, making them easy to type, but hard for non-experts to recognize. Racecar strives to be readable even to non-technical students, teachers and parents. For example, Racecar is a statically typed language, and there are two primitive types: `number` and `word`. No `ints`, `floats` or `doubles` are around to complicate things, and although `string` might be a more general term, `word` emphasizes the important difference between the data types more clearly than `string` does. All of Racecar’s keywords and constructs are designed with this kind of readability in mind. Table 1 demonstrates some examples of Racecar’s readability.

Engaging

The accompanying application will have a simple 2-D animation of a car following the commands the student programmed. The ability to control the outcome of an action, such as driving a car,

Concept	Python	Racecar	Comments
Method declaration	<code>def myMethod(var1, var2) :</code>	<code>define myMethod using var1 (number) and var2 (word)</code>	Uses full words instead of abbreviations and punctuation
Iteration	<code>for i in range(5) :</code>	<code>repeat 5 times</code>	Uses simple, clear, intuitive keywords
Method invocation	<code>car.drive(direction.FORWARD, 10)</code>	<code>drive forward 5 steps</code>	Syntax is similar to OCaml and Haskell: no dots or parentheses required!

Table 1: Examples of Racecar syntax compared to Python syntax

is engaging and teaches students to think creatively while still conforming to rules. Navigating a car visually may be trivial, but Racecar will show students that a precise definition of the car’s movement is necessary to achieve the desired outcome. The goal-oriented nature of the lessons combined with frequent positive feedback coming from the graphical application captures children’s attention. Accomplishing the complex goal at the end of the lesson sequence is rewarding and builds confidence to tackle subsequent challenges.

1.4 Similar Programming Languages

There are a number of technologies available whose goal is to teach children in elementary school to think algorithmically and programmatically. One such “language” is MIT’s Scratch platform, which presents a graphics-based language to children; code is constructed using a drag-and-drop interface. However, physically typing commands into a text editor is paramount in internalization of language constructs, programming style, and procedural thinking. Racecar’s language and platform combines these approaches, compelling children to write their programs in a normal text editor, but then compiling it and importing it into an application where they can see graphical output of their code.

Other approaches to teaching algorithmic thinking have taken the form of games, completely abstracting away the idea of programming from the user. For example, Armor Games’ Light Bot (<http://cache.armorgames.com/files/games/light-bot-2205.swf>) gives children a platform on which to build small programs with a limited number of instructions, forcing them to think in terms of subroutines. Again, this platform, while certainly engaging, fails to give children the irreplaceable experience of typing a procedure word for word, the importance of which was discussed earlier.

Older technologies like LOGO incorporate true algorithmic thinking and graphical output. How-

ever, LOGO's platform is not as engaging as children have come to expect from modern software. Furthermore, LOGO's language itself lacks human-readability compared to Racecar, a feature that is particularly important in conveying programmatic ideas and facilitating an easy transition into coding for children.

Finally, platforms like Lego Mindstorms give children the ability to program their legos to move, allowing creations like robots, self-driving cars, etc. Hardware-based approaches, however, have fundamental limitations in distribution. One set of legos can create only one robot at a time; there is no such limitation with a purely software-based platform, since, for example, a proud parent can send his child's program to relatives without making them buy physical kits.

2 Lesson Plans

Introduction: How to Use Racecar

Racecar is a programming language that allows you to control the motion of a virtual car on the computer screen. The following lessons are designed to teach both you and your students to learn how to write programs that tell the car to move in increasingly sophisticated patterns and routines. Although at first you will only be able to drive in a straight line, by the end of Lesson 10 you will know how to drive in any pattern you can think of and navigate around obstacles on the screen! Before we get to the programming lessons, there are a few things that you, the teacher, should be familiar with so that teaching and helping your students to program is as easy as possible.

First, let's look at the Racecar application. It consists of a window for writing Racecar code, a screen displaying the car and its surroundings, and a few menus and buttons. The "Run Program" button is what you click when you want to run the program sitting in the program window. The "Stop Program" button is how you stop a program that's running if you want it to finish early. The other menus have the usual Save, Open, and Quit buttons that you may be used to from other computer applications.

Second, here are some basic concepts in Racecar that are not exactly programming concepts, but are still invaluable when writing code. Racecar is case-sensitive, which means that the computer treats the expressions like `drive` and `DRIVE` as two completely different, completely unrelated words. Along the same lines, Racecar will not fix your spelling, so be sure to check for typos as you write your programs, as any typos will cause your program to not work as expected. Names that you come up with in Racecar to represent numbers, words, and actions (which are called "variable names" and "function names") cannot have any spaces in them. A standard trick programmers use to keep their names readable, even when they consist of more than one English word, is called Camel Case: keep the first letter lowercase, and then capitalize the first letter of every new word. For example, the phrase "You can write in Racecar" would be formatted in Camel Case as `youCanWriteInRacecar`.

This way, it is simple to see where the words begin and end even though there are no spaces. Another important restriction on these variable and function names is that they cannot be the same as Racecar's keywords. In the language reference manual, Section 10 contains a list of all of the keywords (reserved words) that would lead to ambiguity (and hence errors) if they were used as variable or function names.

Finally, we will share some important lessons about the human side of computer programming. An important part of programming that is not so obvious is that programs are supposed to be written so that *other people* can read them. That is, programs should not be written so that the computer finds them easy to read, but rather so that people find them easy to read. If we wanted programs to be easy for computers to read, we would write in 1's and 0's! (In fact, that was essentially how programs were written when computers were first invented.) A large part of achieving readability is through commenting (Lesson 2). Comments are lines of code that the computer *ignores*—that's right, they are just skipped over completely and mean literally nothing to the computer. However, they are indispensable for human readers since you can write comments in plain English describing what your code does, any problems you had getting the code to work correctly, and anything else that may be relevant for another person (or your future self!) who is reading the code. Although there is no way for us to force you and your students to use comments, *you* can definitely force your students to use them—their programs are not correct if they do not have comments. We will use comments in all of our code examples so that you can get a feel for what appropriate commenting looks like.

A second valuable lesson to learn about programming is that it is almost impossible to write the code you really want on the first try. In other words, you will always mess up in your first draft of a program. And probably in the second, third, and fourth ones, too. The errors you encounter are called **bugs**, and the art of finding and fixing bugs is called **debugging**. It is a practice which is difficult to teach, and consequently much of the time you are coding will actually consist of debugging and learning how to debug. Here are some common bugs that you could start searching for when your programs do not work as expected:

- case-sensitivity errors—are all of the words capitalized (or un-capitalized) correctly?
- typos—did you leave out letters? Misspell words?
- type errors—did you try to assign a **word** to a **number** variable? (Lesson 5)
- punctuation—make sure every open brace has a close brace and every open parenthesis has a close parenthesis

A simple way to hunt down errors after checking these basics is to have the computer print out (using **print** statements) the values that it is messing up at various points in your code. This way, you can see where in your code things start going awry. When you find where the problem is,

change only one thing at a time. Using this method, you will know exactly what the problem was, and can use that knowledge to help you in future programming projects.

In summary, the computer will do *exactly* what you tell it to, whether you want it to or not. As you become more experienced, it will become much easier to remember all of these rules. Nevertheless, we hope that this advice is helpful as you learn how to program and how to teach programming in Racecar.

2.1 Lesson 1: How to Drive

Lesson Summary

In the first lesson, you will learn how to write a simple program in Racecar. The keywords covered are `drive`, `forward`/`forwards` and `backward`/`backwards`, and `step`/`steps`.

The Program

```
drive forward 10 steps
```

This program tells the car to move forward 10 steps. The keyword `drive` is how you tell the car that you want it to move. The keyword `forward` tells the car to move forward (you could also say `forwards` if you want). After the direction keyword comes the number of steps: in this case, 10, followed optionally by the word `step` or `steps`. One step is the smallest distance the car can move.

Further Advice

The other direction keyword is `backward` or `backwards`. Have the students write a program that moves the car backwards 10 steps (or any other number of steps). It should look something like:

```
drive backward 10 steps
```

If the number of steps is large enough (in either direction), the car will reach the border of the window, at which point it will stop, and the students will have to restart the program. See if the students can figure out how many steps it takes to just barely reach the edge without losing the car.

2.2 Lesson 2: Comments

Lesson Summary

Comments are one of the most important parts of a computer program. They allow you to communicate directly to the reader in plain English (or other language of your choosing!) without having

to worry about how the computer will interpret the text, since the computer just ignores it. This lesson covers how to write both single-line and multiple-line comments.

The Program

```
:-( In this program, we will drive the car
forwards and then backwards, so that it
ends up in the same place it started in.
:-)

:) Here, we drive the car forwards
drive forwards 8 steps

:) Now, we will move the car back the same number of steps
drive backwards 4 steps
drive backwards 4 steps :) moving backwards again!
```

This program begins with a multi-line comment. These comments start with a frowny face, `:-(`, and then continue until the first hyphenated smiley face, `: -)`, possibly extending over multiple lines. They are particularly useful to put a summary of the program you are writing at the top of the program itself, as done in the above example, but can also be used anywhere in a program. We recommend that you put one space after the opening frowny face before you type your comments, and that you put enough spaces at the beginning of subsequent lines so that the first column of text lines up. The closing smiley face should then be lined up with the opening frowny face.

The other comments in this program are single-line comments. They begin with a smiley face, `:)`, and continue until the end of the line. Single-line comments are useful for explaining the purpose of shorter blocks of code that are only a few lines long. As you can see in the last example, you can also have a single-line comment on the same line as a piece of code—it extends from the smiley face `(:)` to the end of the line.

There is another big difference between this program and the programs from Lesson 1: there is more than one command the computer executes. The rules for multiple commands are very simple—you may already be able to guess them—each command goes on its own line, and the computer performs the actions in the order they appear on the screen.

Further Advice

Comments are easy to forget about when you are first learning how to program, since the programs are so short, and your students will probably not want to use them at first. The comment symbols are smiley faces to try to make commenting a bit more appealing to children—in many languages, comment symbols are boring, like `# comment`, `/* comment */`, or `(* comment *)`. As a teacher,

you have the opportunity to make sure your students comment by instructing them that their programs are not complete unless it is commented. All of the example programs in the subsequent lessons will be commented so that you can get a feel for what a commented program looks like. A second way to teach students about the necessity of comments is to have them read each other's programs and try to figure out what they do. Although the first few programs they write may be easily decipherable, students will quickly discover that comments are essential in helping them understand what a program does.

2.3 Lesson 3: Turning the Car

Lesson Summary

In this lesson, you will learn how to rotate the car. The car turns by rotating in place. The keywords covered are turn, left, and right.

The Program

```
:- ( This program drives the car forwards
for a bit, and then turns the car to the
left.
:-)
drive forward 10 steps
:) Here we rotate to the left but do not move the car
turn left

:) And now we move the car in its new "forward"
direction
drive forward 1 step
```

This program has three commands. The first command is identical to the entire program from Lesson 1, telling the car to move forward 10 steps. The second line tells the car to turn to the left (the other possibility is turn right). The car turns to point 45 degrees (diagonally) to the left. The last line takes one step forwards in the new forwards direction, moving the car diagonally.

2.4 Lesson 4: More Turning

Lesson Summary

Each turn command rotates the car 45 degrees – or $1/8$ of a complete turn. To turn more than 45 degrees, simply write the turn command multiple times.

The Program

```
:- ( This program will turn the car halfway around and move it
back to where it started.
:-)

:) drive around first just because I want to
drive forward 5 steps

:- ( Each instruction results in another 45 degree turn
so  $4 * 45 = 180$  degrees
:-)
turn right
turn right
turn right
turn right

drive forward 5 steps
```

This program will make the car move forward 5 steps, and then turn the car around completely! At the second command of the program, the car turns a bit to the right. Each subsequent command rotates the car at another 45 degree angle to the direction it is facing. After four turns, the car will have gone through 180 degrees and will be facing the opposite of its original direction. Then, the car moves 5 steps forwards, which is now the opposite direction from its original motion. As a result, the car ends up back where it started!

2.5 Lesson 5: Conditionals

Lesson Summary

In this lesson, you will learn how to instruct the computer to perform an action only if a certain condition is satisfied, and how to react if the condition is not true. You will also learn about the print function, which displays whatever you give it on the computer screen, and the canMove function, which tells you whether there is an obstacle in the car's path.

The Program

```
:- ( In this program we check if car can move to the left.
If it can, we will turn and then move left. If not,
we will print a message to the computer screen with
```

```

that information.
:-)

:) Checks if the car can move left
if canMove left
{
    turn left
    turn left
    drive forward 1 step
}
else :) If we cannot turn left, alert the programmer!
{
    print "Whoa! Don't try to move left!"
}

```

In this program, we explore conditional statements, where a block of code is run or not run depending on an expression that is either true or false. The first line checks if the car can move left by using the (new) keyword `canMove`, which, together with a direction, will be replaced by either true or false (depending on whether there is an obstacle) when the computer gets to evaluate that line of code. If it turns out the car can move left, the result will be true, and hence the following block of code surrounded by curly braces (`{}`) will be evaluated, so the car will turn to the left and move forward. If there is an obstacle in the way, instead of performing the actions inside the if curly braces, the computer skips to the else curly braces and performs the actions in those braces. In this case, the computer will print a message to the screen that alerts the person running the program that the car should not be moved to the left. Note that all conditional statements must be followed by curly braces denoting the corresponding block of code to perform.

You may have noticed that commands inside the curly braces are indented four spaces. This is an extremely useful programming practice, as it allows you to see geometrically exactly how your program is laid out. If you had another if statement inside a curly brace block, you would indent the code inside those curly braces another four spaces, for a total of eight spaces.

Further Advice

Expressions that an if statement can evaluate to true or false can be formed in many ways. The easiest way is to simply use the actual words true and false, as in `if true {...}`, where the statements in curly braces will always be evaluated, and `if false {...}`, where the statements in curly braces will never be evaluated. More complicated expressions can be constructed using the true/false operators (called boolean operators in most other computer languages): and, or, and not. You will probably need to use parentheses to ensure the operators are evaluated in the order you intend, just like in

arithmetic. In particular, with no parentheses present, not is like a negative sign (i.e. -5) and is evaluated first. Then comes and, which is like multiplication, followed lastly by or, which is like addition. The last way to form expressions to give to an if statement is the most common: compare two values using comparison operators: is, is not, >, <, >=, and <=. For example, to check if the number 2 is greater than the number 1 or if the word "hello" is the same as the word "good bye", you could say if 2 > 1 or "hello" is "good bye" {...}. Since 2 is in fact greater than 1, the overall statement is true, so the statements in curly braces would be executed.

If you want to provide more than one alternative, you can use any number of elif statements after an if statement, like the following:

```
:) Print out a message describing any obstacles if canMove left
{
    print "You can move left!"
}
elif canMove forwards
{
    print "Can't move left, but you can move forwards!"
}
else :) maybe some other direction is available
{
    print "Left and forwards are bad."
    print "You're running out of options!"
}
```

Each condition is checked in order, and the corresponding code is skipped if the condition is false. As soon as one condition is satisfied, all of the following conditions are not checked at all and the code associated with them is skipped.

Finally, it is worth mentioning that it is impossible to have an else statement without an if or elif immediately preceding it, and it is also a problem if there is an elif without an if right before.

2.6 Lesson 6: Variables

Lesson Summary

In this lesson, you will learn one of the most important concepts in all of computer programming: variables. Variables allow you to describe actions without knowing everything about them.

The Programs

Program 1:

```
:- ( This program creates a variable "numberOfSteps"
that is a number, sets it to 10, and then
has the car drive that number of steps.
:-)

:) Create the variable
numberOfSteps is a number

:) Set it to a value
set numberOfSteps to 10

:) Command the car to drive forward that number of steps
drive forward numberOfSteps steps
```

Program 2:

```
:- ( This program creates a variable "myWord"
that is a word, sets it to "Hello, world!",
and then prints it out.
:-)

:) Create the variable
myWord is a word
:) Set its value
set myWord to "Hello, world!"

:) Print it out
print myWord
```

In this program, we learn about variables. Variables are like boxes which can hold a value. For example, the variable called `numberOfSteps` holds the value 10 after the second command is executed. Now we can type `numberOfSteps` instead of 10. Every variable has a name, which must be unique in the program, start with a letter, and contain only uppercase and lowercase letters as well as numbers (after the first letter). For example, `theseWordsMakeAVariable` is a valid variable name, but `my number`, `2ebra`, and `hi!` are not because they contain a space, start with a number,

and contain a non-alphanumeric symbol, respectively. Also, variables cannot have the same name as any of the keywords in Racecar, such as drive, turn, print, if, etc. Although you technically could use those words with different capitalizations, we recommend against it, as it is confusing. Similarly, you could have two different unique variables called myVariable and MyVarIaBLE, but again, we strongly, strongly recommend against it.

Every variable also must have a type: either a number or a word. Numbers can be positive or negative whole numbers. Words consist of an opening double quote, ", followed by 0 or more characters (i.e. letters, numbers, or symbols that are not double quotes), followed by a closing double quote. For example, "Sam said, 'Hello, world!'" is a valid word, but "Sam said, "Hello, world!"" is not because it contains double quotes. Confusingly, it is possible to have a word that is just a single number: "5" is still a word, because it is surrounded by double quotes. Before you can assign a value to a variable, you must declare the variable's name and type. This is done with statements like numberOfSteps is a number and myWord is a word.

Variables can be used in any statement as a substitute for an actual number or word. In the above program, for example, the variable numberOfSteps replaced the number that is usually found in drive statements in the statement drive forward numberOfSteps steps. The power of variables is that when you write a command that has a variable in it, you do not need to know what the value of the variable is! In this manner, you can write a program that does many different things depending on the actual value of the variable. Also, you can set variables to other variables, or even to modified versions of themselves. For example, set myNumber to myNumber * 2 will store in the myNumber "box" twice what was previously stored there!

Further Advice

Variables can be compared to other variables or to actual numbers or words in if statements using the keywords is and is not. For example, the following program checks if my name is Sam and if so, prints out the message "Hello, Sam!".

```
:- ( This program creates a variable 'myName'
to be a word, sets it to "John", and then
checks if it is "Sam" and if so, it prints
out "Hello, Sam!". If not, it prints out
"You're not Sam!".
:- )
```

```
: ) Create the variable
myName is a word
```



```

:) Set its value
set myName to "John"

:) Check if the variable is "Sam" and
print something if so
if myName is "Sam"
{
    :) Tell Sam hello
    print "Hello, Sam!"
}
else
{
    :) This person is not Sam, so tell him so
    print "You're not Sam! You must be " ++ myName
}

```

As an exercise, have the students write a program where the car will drive forward only if the variable `numberOfSteps` is set to 7. Of course, they must declare the variable before they write the conditional statement, and they are free to assign any value to it, as long as it is a number. It is illegal to try to compare a number to a word. Numbers also have additional ways of being compared: `>`, greater than; `<`, less than; `>=`, greater than or equal to; and `<=`, less than or equal to. Words cannot be compared this way.

This program also uses the "concatenation" or "joining" operator, `++`. Putting two plus signs in a row between two words (either variables or actual words) results in a new word which is just the first word followed by the second. So, the code inside the above `else` statement will print out a single message that ends with whatever word is stored in the `myName` variable.

2.7 Lesson 7: Loops Part 1

Lesson Summary

In this lesson, you will learn how to make the computer perform an action many times using a loop.

The Program

```

:-( This program creates a variable 'myCounter'
    that is a number, sets it to 1, and repeats a
    block of code until myCounter is 5.
:-)

```

```

myCounter is a number
set myCounter to 1

:) Here is our loop
repeat if myCounter is not 5
{
    :) The code in here will repeat until myCounter is 5
    drive forward 1 step
    set myCounter to myCounter + 1
}

```

In this program, we explore repeat-if loops. As you can see, we have a variable myCounter. The repeat-if loop is similar to a conditional statement in that it checks an expression to see if it is either true or false. If it is true, it will run the following block of code. It is different, though, because after it runs the block of code, it rechecks the truth value of the original expression, and if it is true again it will run the block of code again. It will continue to do so (checking the value of the expression and running the code) until the expression becomes false. It is possible that the first time the expression is checked, it is false, in which case the code inside curly braces is never executed. More dangerously, it is possible that the expression will always be true, in which case the loop will repeat forever! This is called an "infinite loop," and the only way to escape from an infinite loop is to force the program to stop using the STOP button on the Racecar application screen. It is up to you to ensure that your loops will always end eventually.

Further Advice

This kind of loop is especially useful when you do not know how many times you want to run the code (for example, if you are moving around looking for something). For example:

```

:-( This program is similar to our last, but this
    time we stop our loop when a word has changed
    instead of a number
:-)

location is a word
set location to "out "

:) save the car's current position in a variable
homePosition is a number
set homePosition to getCarPosition

```

```

:) drive away from home!
drive backwards 15 steps

:) try to find "home"
repeat if location is not "home"
{
    :) you are in the loop so you must not be "home"

    :) first, drive one step
    drive forward 1 step

    :) then, check to see if you are home
    if getCarPosition is homePosition
    {
        set location to "home"
    }
}

```

As long as the car's location is not equal to the home location, location will still be set to "out", so the code will continue to run and the car will continue to move forward until it reaches its home.

2.8 Lesson 8: Loops Part 2

Lesson Summary

In this lesson, you will learn about repeating an action a particular number of times using another kind of loop.

The Program

```

:-( This program is similar to our last, but we now use a loop that
    runs a specified amount of times
:-(

numCorners is a number
set numCorners to 8

:-( Instead of writing "drive forward 1 step (and)

```

```

        turn right" a bunch of times, just repeat the
        command numCorners times!
:-)

repeat numCorners times
{
    drive forward 1 step
    turn right
}

```

In this program, we explore repeat-times loops. Similar to a repeat-if loop, a repeat-times loop repeats the enclosed block of code multiple times. Rather than depending on a true or false expression to decide whether the code will be repeated, repeat-times loops always run the enclosed code a specific number of times. In the above program, that number is specified by the variable `myCounter`. You could also use a more complicated expression (for example, adding two numbers together) to get the number of times to repeat.

Further Advice

Instead of using a variable, you can also specify an actual number of times you would like the block to run. For example:

```

repeat 5 times
{
    drive forward 1 step
}

```

2.9 Lesson 9: Subroutines

Lesson Summary

In this lesson, you will learn how to give a nickname to a sequence of commands that you can use later. This sequence of commands is called a "subroutine."

The Program

```

:-( In this program, we create and use a subroutine,
    which defines a set of commands.
:-)

```

```

:-( Here is the creation of our subroutine. This code

```

```

    is not actually executed at this point in the
    program because of the keyword "define".
    Instead, these commands are saved for later.
:-)

:-( shiftLeft moves the car to the left by 1 step
    and returns it to face in its original
    direction.
:-)
define shiftLeft
{
    turn left
    turn left
    drive forward 1 step
    turn right
    turn right
}

```

:) Here is the code we want to actually run:

```
drive forward 10 steps
```

```

:-( Here is the first time we actually use
    or "call" our subroutine
:-)
shiftLeft
drive forward 10 steps
shiftLeft
drive forward 10 steps
shiftLeft
drive forward 10 steps
shiftLeft

```

In this program, we write a subroutine. Subroutines are like shortcuts or nicknames: they allow us to write code only once for something we will use multiple times in our program. (Also, if you prefer, you can teach your students the name "nickname" instead of subroutine, which is a big, scary word!) Our subroutine is called `shiftLeft`, which turns the car 90 degrees to the left, moves one step, then turns the car back to the right (i.e. straight). Notice how efficient it was to code the

program using our subroutine as opposed to writing out all the lines of code required for stepping to the left whenever we wanted to.

Further Advice

After we write out our program, it should be clear that we could have used a repeat-times loop instead to "drive 10 steps and then shift left," 4 times. You can ask your students to rewrite their code to incorporate this. Students may point out that now our subroutine does not make our program any shorter or easier to write. While this is true for this simple program, it often won't be for others, and it is important to emphasize this to students. And, even in this short demonstration, the subroutine's descriptive name results in more readable code.

2.10 Lesson 10: Variables in Subroutines

Lesson Summary

In this lesson, you will learn how to pass information into a subroutine to modify the subroutine's action.

The Program

```
:-( In this program, we create and use a subroutine
    that takes a variable when it is used.
:-)

:) This time, during our creation we specify a variable
define shiftLeft using distance (number)
{
    turn left
    turn left

    :) Here is where the "distance" variable will be used
    drive forward distance steps
    turn right
    turn right
}

:-( Now we use the subroutine, specifying the variable
    which gets plugged into the code
    we wrote above for our subroutine
```

```
:-)
shiftLeft 8
drive forward 10 steps
shiftLeft 4
```

Like the last program, we write a subroutine for stepping to the left called `shiftLeft`, but unlike the last program we now are using a new variable in our subroutine called `distance`. These kinds of variables that are defined at the same time as a subroutine are called parameters or arguments. When you define the subroutine, you must give the new parameter's name and type. This is equivalent to declaring a "normal" variable using the statement `variableName is a "type"`. You can only refer to parameters inside the subroutine that they are defined with, and you cannot refer to any outside variables inside the subroutine. Now we can have the car move any number of steps in its big shift to the left. We will decide exactly how many steps when we use the subroutine in our code. In this program, after we make the subroutine, we use it, first moving 8 steps. We then have the car drive forward a bit and then shift only 4 steps. It is much simpler to put a different number here than to copy the `stepLeft` subroutine from Lesson 9 each time you want to move a new number of steps! Instead of using an actual number, you can also use another (previously defined) variable as a parameter, as in `shiftLeft myNumberVariable`.

Further Advice

Subroutines with parameters can have any number of parameters. To add more, simply use and:

```
define shiftLeftThenDriveStraight using numStepsLeft (number) and numStepsStraight (num
{
    turn left
    turn left
    drive forward numStepsLeft steps
    turn right
    turn right
    drive forward numStepsStraight steps
}
```

:) How to call the subroutine:

```
shiftLeftThenDriveStraight 5 10
```

3 Language Reference Manual

3.1 Introduction

This manual contains the specification of the Racecar language. It is split into 9 sections: identifiers and scope, data types, statements, expressions, comments, control flow, built-in functions, user-defined functions, and the formal grammar. The notation used is as follows: text in typewriter represents code. Italicized and `<bracketed>` text represent placeholders for code. The actual code fragments to replace the placeholders are specified either before or after the code block with the placeholder.

3.2 Identifiers and Scope

Identifiers are function or variable names. Syntactically, identifiers are limited to alphanumeric strings beginning with a letter (no underscore), and they are case-sensitive. There is no limit to the length of an identifier, and all identifiers must be unique in their scope. Declaring an identifier which is already declared in the same scope is an error.

There are three kinds of scopes a variable can have in Racecar: global, function, and control-flow. The actual commands executed by Racecar are not contained in any function—they are "global." Any variables declared outside of all curly braces are said to have global scope. They can be accessed everywhere except for inside function definitions. There is no main method required, but it is recommended that in all but the most trivial programs there is a single main-like function which drives the program. This function would be called in the global scope, e.g. on the first line of the program after any comments.

A variable declared inside a function definition, but outside of any other curly braces, has function scope. So does a variable listed as a function parameter. These variables can have the same names as variables declared outside the function, since variables with function scope are isolated from variables whose scope is outside of the function (i.e. global scope). Function scope extends into any curly braces inside the function definition.

Control-flow statements, namely repeat-if, repeat-times, and if-elseIf-else, define the boundaries of control-flow scope. The scope of variables declared inside curly braces of a control-flow statement extends everywhere inside those curly braces. Additionally, variables whose scope extends directly up to a control-flow statement are also accessible inside that control-flow statement. Consequently, it is illegal to declare a variable inside a control-flow statement if a variable with the same name is accessible just outside that statement.

Identifiers representing functions have a scope which extends throughout the entire source file—into other functions and outside of all functions (i.e. global scope). Functions can be recursive, i.e. the scope of a function name identifier extends into the function definition itself. It is illegal to

declare a function inside another function, so all functions must be declared as global functions.

3.3 Data Types

There are three data types in Racecar: word, number, and boolean. A literal word is an opening double-quote, followed by any string of characters that is not a double quote, followed by another (closing) double-quote. The empty word is notated as `"`. There is no way to have a double-quote as a character in a word. Words can be concatenated with the concatenation operator `++`, which is left-associative, and their equality can be tested with the operators `is` and `is not`. Numbers are positive or negative integers. They can be combined using the arithmetic binary operators `*`, `/` (integer division), `+`, and `-`, under the standard arithmetic operator precedences and left associativity. They can be compared using any of the comparison operators: `is`, `is not`, `>`, `<`, `>=`, and `<=`. Arithmetic operations are evaluated before comparisons. Numbers can be concatenated into strings and into other numbers (resulting in a string). The left-associativity of the concatenation operator means that the following is still valid: `"hello" ++ 5 ++ 2 ++ 3`, and the result of the expression is the word `"hello523"`. To "cast" a number to a word, simply concatenate it with the empty word, as in `" ++ 5`, which evaluates to the word `"5"`. There is no way to cast a word to a number. Boolean values are not allowed to be stored in variables, and hence the word `"boolean"` is not a reserved word (although its use as an identifier is strongly discouraged!). The boolean literals are `true` and `false`. Booleans are used internally to evaluate the result of comparisons and boolean operations (`and`, `or`, and `not`) in `if-elseIf-else` and `repeat-if` statements.

3.4 Statements

All Racecar programs consist of zero or more statements. Statements end at the end of the line. There is no way to put two statements on a line, or to split a statement up over multiple lines. There are 10 types of statements: `drive`, `turn`, function definition, function invocation, `repeat-if`, `repeat-times`, `if-else`, `print`, declaration, and assignment. Drive and turn statements are covered in the build-in functions section, and `repeat-if`, `repeat-times`, and `if-elseIf-else` are covered in the control flow section. Function definition and function invocation statements are specified in the user-defined functions section. The remaining statements are `print`, declaration, and assignment.

Print statements cause the specified text to appear in the graphical "console" in the Racecar application. The syntax for print statements is: `print word_expression` (word expressions are covered in the expressions section). This command causes the value of the word expression to appear in the console. Print statements are newline-terminated automatically.

All variables must be explicitly declared before they can be used. The syntax for declarations is: `IDENTIFIER is a type`, where `identifier` is any valid identifier and `type` is either `number` or `word`. Before a variable has been assigned a value, using it (in an assignment or an expression) is an error.

Assignment statements store a value in a previously-declared variable. The syntax for variable assignments is: `set IDENTIFIER to expression`, where `expression` is either a numeric or word expression. Attempting to assign a word expression to a numeric identifier or vice versa will generate an error, as will assigning a boolean expression to any identifier, or any expression to an undeclared identifier or to a function.

3.5 Comments

There are both single-line and multi-line comments in Racecar. Single line comments begin with `:)` (smiley) and end at the end of the line. They do not need to begin at the beginning of the line. Multi-line comments begin with a hyphenated frowny face `:-)` and end with a hyphenated smiley face `:-)`. They cannot be nested (i.e. they are C-style).

3.6 Expressions

An expression is a sequence of one or more identifiers, delimiters, and operators. Expressions can be word expressions, numeric expressions, or boolean expressions. Word expressions only contain string literals, identifiers of type `word`, numbers (only following a concatenation operator) and concatenation operators. Numeric expressions only contain arithmetic expressions, namely integer constants, identifiers of type `number`, the `+`, `-`, `*`, and `/` operators, and parentheses. Division is integer division. There are no floating-point numbers. Boolean expressions contain the constants `true` and `false`, comparisons of numeric and word expressions to similarly-typed expressions, and the logical `and`, `or`, and `not` operators. Valid comparison operators are `is`, `is not`, `>`, `<`, `>=`, and `<=`. There is no `==` or `!=` (use `is` and `is not` instead). Words can only be compared using `is` and `is not`. Delimiters are whitespace and parentheses.

3.7 Control Flow

The conditional statement construct in Racecar is defined as follows:

```
if <boolean expression>
{
    <code block>
}
elif <boolean expression>
{
    <code block>
}
else
```

```
{
    <code block>
}
```

This code executes the first code block if the first boolean expression is true. If it is false, the next boolean expression is checked and the corresponding statement is executed if it is true. If not, the next boolean-code block combination is checked, until any non-elseIf and else statement is reached. This is in exact analogy with most other programming languages' conditional constructs.

The elseIf and else statements and blocks are optional. Additionally, there can be any number of consecutive elseIf statements (including 0) following the initial if. There must be an if before an elseIf or else statement. There must be a newline between the statement line and the opening curly brace, and between the closing curly brace and the following statement (unless the closing curly brace is the last line of code).

There are two types of loops in Racecar: repeat-if (similar to while in other programming languages) and repeat-times (similar to for). The repeat-if statement looks like the following:

```
repeat if <boolean expression>
{
    <code block>
}
```

This loop checks the value of the boolean expression and executes the code block if it evaluates to true. It then checks the value of the boolean expression again and executes code block. This process repeats until the boolean expression evaluates to false. Any boolean expression can be used in these loops.

The second type of loop is the repeat-times loop. It looks like the following:

```
repeat <number expression> times
{
    <code block>
}
```

This loop will execute the code in the code block repeatedly the specified number of times. You can specify this number by any numerical expression.

3.8 Built-in Functions

There are 5 built in functions: drive, turn, canMove, getWheelDirection, and getLocation. The first two are complete statements when they are invoked correctly (similar to all user-defined functions), but the latter three return values and hence are expressions (unlike user-defined functions). The

direction arguments to the first three functions are not strings, but rather language keywords. The drive function is defined by the following:

```
drive <direction> <number expression> <optional step(s)>
```

drive tells the program to move the car <number expression> steps in the <direction> direction. The possible values for <direction> are: forward, forwards, backward, and backwards (all of these are reserved words). The user can either type step or steps, or omit the term entirely. Racecar does not check the English grammar of using step for only moving 1 step and steps for 0 steps or 2 or more steps.

The turn function is the following:

```
turn <direction>
```

The turn function turns the car 45 degrees in the direction specified. The possible values for <direction> in this function are: left and right. The <direction> is the relative direction the car will turn: left/counter-clockwise, or right/clockwise.

The canMove function is the following:

```
canMove <direction>
```

The canMove function returns a boolean (true or false) representing whether the car can move one step in the given direction without hitting an obstacle or the edge of the map.

The getCarPosition function takes no arguments and returns a number corresponding to the unique position of the car on the map. The exact representation of the car's coordinates is implementation-specific and subject to change, and hence should not be used other than for equality comparison with some previously stored position.

3.9 User-defined Functions

Users can define their own functions in Racecar. Function definition follows the signature:

```
define <identifier> <optional parameter list>
{
    <code block>
}
```

The function name must be a valid identifier and cannot conflict with any variable names that have global scope. The <optional parameter list> is defined by the following:

```
using <identifier> (type) and ... and <identifier> (type)
```

The keyword using is required if there is at least one parameter, and the keyword and is required whenever there are two or more parameters, between the type of one parameter and the name of the

next. Parameter names cannot conflict with any function names (even other functions), but may conflict with global variable names since global variables are inaccessible inside functions. Note that the parameter list is optional—it is omitted completely if there are no parameters to the function. Functions cannot return values.

User-defined functions are invoked using the same syntax as built-in functions:

```
function_name parameter_1 ... parameter_n
```

Unlike some built-in functions, user-defined functions are always complete statements on their own, and cannot be used as expressions. To use a non-trivial expression as a parameter, surround it with parentheses. For example, `myFunction (2 + 3) "hello" ("John" ++ " Doe")` is a function call where `myFunction` is called with three parameters.

3.10 Reserved Words

The following are reserved words and cannot be used as identifiers (function and variable names):

3.11 Formal Grammar

The following listing shows the Racecar grammar, as generated by the PLY parser. The grammar productions are formatted as follows: nonterminals are in lowercase; terminals are in UPPERCASE; literals are `(, {, and }`. The empty string is represented by the production ‘empty’, which produces the PLY `<empty>` statement. The start symbol for all Racecar programs is ‘statements’. Multi-line comments are treated as whitespace by the lexer and hence are not covered by the grammar.

```
statements -> statements statement
statement -> error NEWLINE
statements -> empty

statement_block -> { statements } newline_opt_comment

empty -> <empty>

newline_opt_comment -> opt_comment NEWLINE

opt_comment -> SINGLE_LINE_COMMENT
opt_comment -> empty

statement -> simple_statement
statement -> compound_statement
```

```

simple_statement -> statement_contents newline_opt_comment
simple_statement -> newline_opt_comment

statement_contents -> drive_command
statement_contents -> turn_command
statement_contents -> print_command
statement_contents -> assignment_command
statement_contents -> declaration_command
statement_contents -> function_command

compound_statement -> define_command
compound_statement -> repeat_if_command
compound_statement -> repeat_times_command
compound_statement -> if_command

expression -> can_drive_expression
expression -> comparison

can_drive_expression -> CAN_DRIVE drive_direction primary_expression opt_steps

comparison -> plus_expression comparison_operator plus_expression
comparison -> plus_expression

comparison_operator -> IS
comparison_operator -> IS NOT
comparison_operator -> GT
comparison_operator -> LT
comparison_operator -> GEQ
comparison_operator -> LEQ

plus_expression -> plus_expression + times_expression
plus_expression -> plus_expression - times_expression
plus_expression -> times_expression

times_expression -> times_expression * word_expression
times_expression -> times_expression / word_expression

```

```

times_expression -> word_expression

word_expression -> word_expression CONCAT primary_expression
word_expression -> primary_expression

primary_expression -> ( expression )
primary_expression -> NUMBER
primary_expression -> WORD
primary_expression -> GET_CAR_POSITION
primary_expression -> ID

function_command -> ID opt_parameters

opt_parameters -> opt_parameters primary_expression
opt_parameters -> empty

drive_command -> DRIVE drive_direction plus_expression opt_steps

drive_direction -> FORWARD
drive_direction -> BACKWARD

opt_steps -> STEP
opt_steps -> empty

turn_command -> TURN turn_direction

turn_direction -> LEFT
turn_direction -> RIGHT

define_command -> DEFINE ID opt_param_list newline_opt_comment statement_block

opt_param_list -> USING ID ( type_enum ) opt_extra_params
opt_param_list -> empty

opt_extra_params -> AND ID ( type_enum ) opt_extra_params
opt_extra_params -> empty

```

type_enum -> WORD_TYPE

type_enum -> NUMBER_TYPE

repeat_if_command -> REPEAT IF expression newline_opt_comment statement_block

repeat_times_command -> REPEAT plus_expression TIMES newline_opt_comment statement_block

if_command -> IF expression newline_opt_comment statement_block opt_else_if opt_else

opt_else_if -> ELSE_IF expression newline_opt_comment statement_block opt_else_if

opt_else_if -> empty

opt_else -> ELSE newline_opt_comment statement_block

opt_else -> empty

print_command -> PRINT word_expression

declaration_command -> ID IS A type_enum

assignment_command -> SET ID TO expression

4 Project Plan

By Sam Kohn

4.1 Team Roles

Responsibilities for the project components were divided based on the following roles:

Project Manager Project Deliverables (Sam Kohn)

Language Guru Language design and evolution (Alex Fields)

System Architect Interpreter structure (Jeremy Spencer)

System Integrator Development and runtime environments (Mason Silber)

Verification and Validation Test plan and test suites (Colfax Selby)

These roles were explicitly assigned as guidelines rather than absolutes. Consequently, the actual roles of each team member more closely resembled the following:

Messenger and Starter Send lots of messages and begin design of interpreter modules (Sam Kohn)

Simplifier Make the language simpler than humanly possible (Alex Fields)

Upgrader Add functionality to the minimal interpreter kernel (Jeremy Spencer)

GUI Guru Design and implement the GUI (Mason Silber)

Loose Ends Plan tests and then help out with literally everything else (Colfax Selby)

As a consequence of these roles, very few modules in our interpreter were developed by exactly one person.

4.2 Development Process

Our project schedule (below) set the pace for our development. There were not many design decisions that went into the overall system architecture, since there is a standard compiler structure that is suitable to most of our needs. The few decisions we did make (compiler or interpreter, and implementation language) were made by the entire team after discussion at one of our meetings.

4.3 Project Schedule and Log

Our entire project schedule was written at the beginning of the project. The actual completion dates (i.e. the project log) are in parentheses at the end of each task.

2/24 White Paper Draft, example of valid program, decide on implementation language (2/24)

2/26 White Paper Final (2/24)

3/01 Style guides: for us (implementation) and for Balloon code (2/28, updated 4/17)

3/10 Draft of grammar (3/10)

3/20 Tutorial and Reference draft (3/26)

3/26 Tutorial and Reference final (3/27)

4/02 Parser complete (4/28)

4/09 Draft/sketch of individual final report sections (the non-team ones) (5/7)

4/16 Translator complete (4/28)

4/23 Runtime environment complete (4/30)

4/30 Code freeze (just before finals start) (5/10)

5/07 Final of individual report sections, draft of team sections, draft of demo (5/10)

5/07-5/14 Practice demo on friends at least twice (4/29 friends, 4/30 Professor Aho)

5/13 Final Report and demo complete (4/30 demo, 5/10 report)

5/14 Demo (5/1)

Summary 5 early, 2 on time, and 9 late.

This project schedule was too optimistic. We were late on many of the coding tasks according to the schedule, but we finished a sufficient part of the project to demo it two weeks ahead of schedule.

4.4 Implementation Style

We used PEP-8, a Python style-checking utility, as the style sheet for our implementation code. The style guide can be found here: <http://www.python.org/dev/peps/pep-0008/>. All of our code passes the style checker utility for PEP-8, although it may not conform exactly to the PEP-8 specifications online.

5 Language Evolution

By Alex Fields

Our "buzz words" for our language are engaging, easy-to-teach, and readable. Engaging is based on what our language was developed to do, i.e. control a virtual racecar, and is based on our method of having the user interact with our language through our GUI. Easy-to-teach and readable, though, are at the core of our language design. Our language is meant to be simple, clean, and naturalistic.

We set out to develop a language that minimized the overhead in teaching children computer science concepts. Our goal was to teach things like algorithmic-thinking, the concept of a computer program, and basic understanding of use of variables, loops, etc. In our own programming experience, we were forced to approach these concepts through complex programming languages that required digesting extra material in order to begin to interact with the languages. We wanted a language in which a program could be created without an understanding of eight or more different 'types', without an understanding of objects, without an overabundance of syntax rules.

We decided that, in the goal of making our language as minimal as possible, we would use little punctuation. Like OCaml and Haskell, function calls are extremely bare-bones: the language requires the function name and the variables, all separated by a space (ex. "makeTurn right 10"). Like Python the language newline-terminated statements. Again, the goal of these syntax decisions was to attempt to assimilate the language as much to the children's means of understanding rather than have the children assimilate to the language.

Along the same vein, the language semantics are based off of English. This was quite easy to do since the language's built-in functions are commands for car movement. We wanted users to be able to think out in English what they wanted the car to do and then write that out with as little translation from one to the other as possible. This makes the language both easy-to-teach and readable. Reserved words are also all English words, the best example perhaps our types, which are simply 'word' and 'number'. The language uses the words 'set' and 'to' for assignment (ex. "set myVar to 10"). There are no foreign words in our language, such as 'int' or 'double'.

In incorporating these requirements into the language, the language did lose some functionality as compared to more complex languages such as C or Java. The language does not have boolean operators. The language lacks objects. The language does not deal at all with pointers and references. Luckily, the functionality that is built into the language while still meeting these language requirements more than satisfies the overall goals of the language.

6 Translator Architecture

By Jeremy Spencer

The Racecar language system architecture performs the task of taking racecar code and translating it into equivalent python code which can be executed within the Racecar GUI environment. There are a few modules within the system that enable this progression, namely a lexical analyzer, parser, and translator.

The first step is to pass the sequence of characters from the racecar code to the lexical analyzer which produces a sequence of tokens. In our system we utilized the Python Lex-Yacc (PLY) tool to implement both the lexer and the parser. The lexer identifies important tokens in the language such as drive and steer whose identification is useful in the parsing stage.

The parser takes the sequence of tokens from the lexer and produces an abstract syntax tree (AST). The parser forms the ast utilizing the tokens received from the lexer and builds the tree conforming to racecar's grammar specification. Any racecar code that fails to build a valid parse tree is considered malformed and not valid racecar code. Within this same module we also perform semantic analysis on the racecar code. This step performs type checking on variables and also ensures that the correct scoping rules are followed for variable use. If the racecar code passes the semantic analyzer and is error free, the ast can then be passed to the translator.

The translator translates each node (semantic element) of the ast at a time, yielding python code which can be utilized within the racecar gui. In Figure 1, the diagram on the left shows a block level diagram of the system architecture. The images on the right are the associated steps for each phase of the translator for the example racecar code drive forwards 5 steps.

Although each team member played a part in the construction and maintenance of the imple-

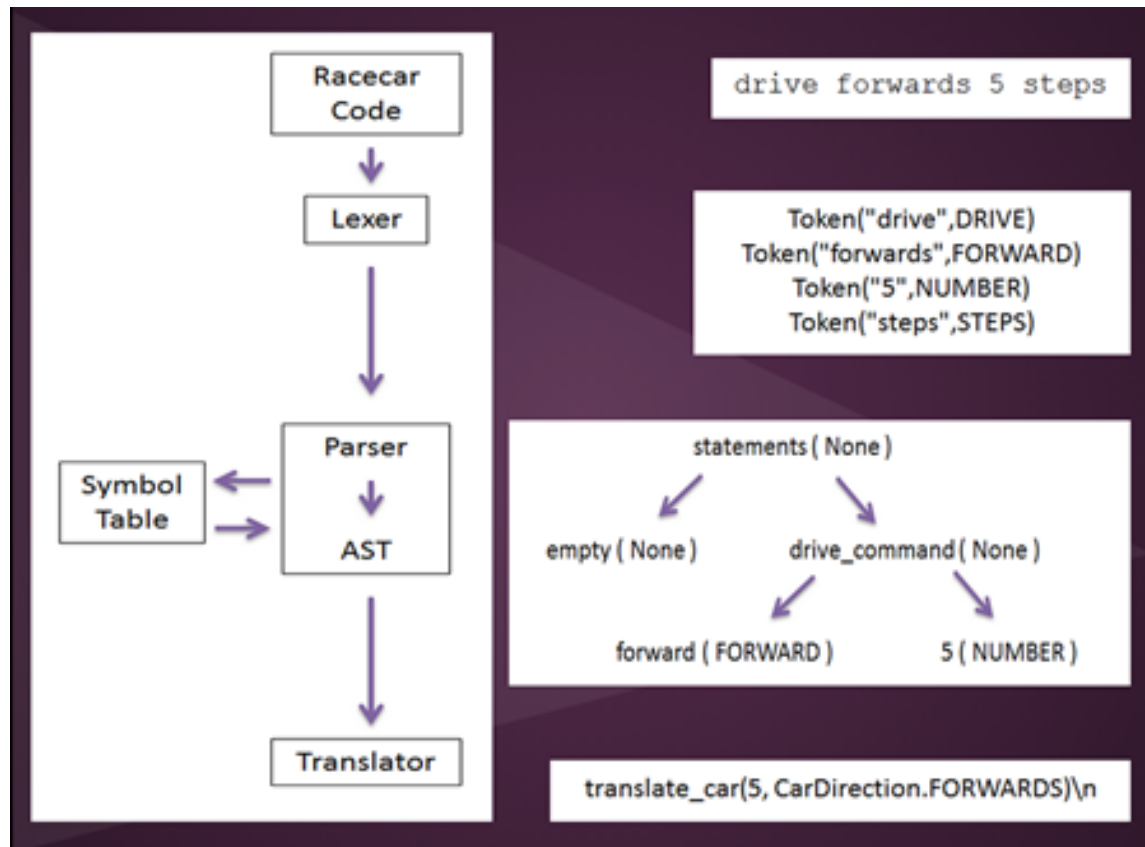


Figure 1: System Architecture

mentation of the architecture, the following shows the module and which team members had the most influential role in its development:

- Lexer (Sam Kohn)
- Parser-AST (Sam Kohn, Jeremy Spencer)
- Semantic Analyzer (Alex Fields, Colfax Selby)
- Translator (Sam Kohn, Jeremy Spencer)
- IDE (Mason Silber, Sam Kohn)

7 Development and Run-Time Environment

By Mason Silber

We developed our language in Python using standard text editors like Vim and Sublime Text. We also heavily relied on git and Github for version control; since each member of the team was working on a number of things at once, we wanted to make sure that we didn't step on anyone's toes when making changes, so having a central repository for our code base was essential. Our Integrated Development Environment (IDE) launches directly from a package, so no makefile is required to write code in our language.

The runtime environment is a simple IDE customized for the Racecar language and the targeted demographic. From the IDE's menu bar, the user has the ability to save their code to files, and to open previously saved files. The user can also place obstacles on screen through which the car must navigate. The IDE has three main components: the coding text area, the console, and the canvas on which the racecar moves. Each of these components will be briefly discussed.

The coding text area takes up the left side of the IDE, and is where users write their Racecar programs. Once the user has written the code they would like to run, they press the "Run Code" button located directly below the coding text area. Buttons are also available to clear the coding text area and to reset the car's position. Once "Run Code" has been pressed, another button labeled "Stop Program" becomes available, which will terminate the running program.

The console is more straightforward. It is used to display textual information to the reader. If a user attempts to run a program with errors, the errors are output to the console with information about where the error can be found. It also lets the user know when a program has begun to be executed, and when it has finished running. Additionally, users can put print statements in their code; those print statements get printed to the console.

The heart of the runtime environment is in the canvas. The canvas is the area in which the car drives, but more importantly, it's where users get to see visual output of their code. The

car animates across the screen, driving forward and turning. The canvas also holds a number of obstacles, should the user desire them (to be selected from the menu bar at the top of the IDE window). Obstacles from things as simple as blocks to things as complex as mazes are displayed on the canvas, and the user has the responsibility to write an algorithm to navigate through these obstacle courses. If the car collides with any of the obstacles, the car is reset to its original position and the user has to start over.

All of these features are implemented within the IDE to provide a self-contained, fully featured experience for users. No setup is required besides opening the IDE. No interaction with external files is required or necessary under any circumstances. The IDE provides the entire platform on which Racecar algorithms can be run, thereby simplifying the entire experience for students and teachers alike.

8 Test Plan

By Colfax Selby

We used a segmented and iterative testing methodology. As our compiler matured, we built up out tests to test functionality from end to end, to make sure that not only every part works fine by itself, but every part of the compiler works with each other.

We leveraged the power of the Python framework unittest to handle the testing suite and keep everything organized and easy to run. Below I will get into the details of how we planned and executed our tests.

8.1 Segmented Testing

We had different test classes for each part of the compiler (Semantic Analyzer, Translator, etc). We within each class we tested individual functionalities of the given part of the compiler, which I will explain below. The segmented nature of our test suite allowed for us to test specific parts of the compiler separately from everything else, therefore allowing us to solely test the Translator when it was implemented and do the same for other parts of the compiler. The test program would indicate how many errors and failed tests there were per class, so it was easy to tell which part of the compiler was causing problems.

Our three test classes were: Symbol Table, Translator, and Semantic Analyzer. We decided to test the Parser along with the Translator and the Semantic Analyzer, with parser specific error messages, in order to avoid rewriting many similar tests and utilizing the strategy of code reuse. Additionally, as each individual part was tested and working, we built up many of the tests to not only test their particular part of the compiler, but to test more steps along the way. We did this

so that we would have as many tests as possible, testing as many things as possible, to ensure that we would catch any errors that may have occurred.

8.2 Iterative Testing

We also took an iterative approach to testing. Rather than writing the whole compiler and then running tests at the end, we wrote tests along the way to ensure that each part and each bit of functionality was implemented properly. Each of the tests had descriptive names (such as `test_function_invocation_with_one_parameter` or `test_get_car_position`) so in the output, if a test did fail, we could tell exactly what wasn't working.

An example test, followed by a detailed explanation, is below:

```
def test_function_invocation_with_two_parameters(self): test_string = \ """define turn-
LeftThenDriveStraight using numStepsTurn \ (number) and numStepsDrive (number) { turn left
drive forward numStepsTurn steps turn right drive forward numStepsDrive steps } turnLeftThen-
DriveStraight 5 10 """ correct_translation = \ """def turnLeftThenDriveStraight(numStepsTurn,
numStepsDrive): rotate_car(WheelDirection.LEFT) translate_car(numStepsTurn, CarDirection.FORWARDS)
rotate_car(WheelDirection.RIGHT) translate_car(numStepsDrive, CarDirection.FORWARDS) turn-
LeftThenDriveStraight(5, 10) """ result = Compiler.getPythonCode(test_string) ast = Parser.parseString(test_string)
saErrors = SemanticAnalyzer.analyzeStart(ast) self.assertEqual(len(saErrors), 0)
self.assertEqual(len(ast.errors), 0) self.assertEqual(result[0], correct_translation)
```

As you can see, this test checks the functionality of the parser, semantic analyzer, and the translator. The `test_string` is an example code snippet in our Racecar language. It is then manually translated and stored in `correct_translation`. The test Racecar snippet is then run through the parser, semantic analyzer, and then the translator. The outputs of each of these steps is checked to ensure that there are no errors. Finally, the output of the translator is checked against the manual translation of the code, and if all of the above match as expected, the test passes. If any of the above do not function as expected, the particular assertion fails, and the test fails with a descriptive output.

9 Conclusions

9.1 Lessons Learned as a Team

Successes

Many aspects of our project went according to plan or better. The most important of these, in our opinion, was our choice of implementation language, Python. Although we decided on our implementation language (Python or OCaml) relatively late because we could not decide which

would be more helpful, all of us are glad we chose Python. The language never interfered with our design, and the issues we did run into during implementation were generally resolved by making the code simpler. That is, we tended to mess up when we tried to write un-Python-like code, which indicates to us that we made the right decision. A second organizational success was our decision to only loosely follow our team roles. We were worried at the beginning of the project that some people would get stuck with the less interesting tasks, and some people with much more work than the others, so we explicitly resolved to not strictly follow our team roles. This had the positive effect of encouraging us to look at each other's code without being forced to (by, for example, the project manager), since we did not have a one-to-one correspondence between people and modules. This means that every module has been read and approved by at least two people, yet this was accomplished not as its own task, but rather as part of the natural development process.

Opportunities for Improvement

Overall our project went very smoothly, but we did run into a few issues along the way. The biggest issue our team ran into was adhering to the timeline and completing all desired functionality on under the constraints. We did a good job of designing our language with a manageable scope, but as the semester progressed, we didn't have time to completely a few bits of the functionality we wanted to implement. This did not end up being an issue, though, because we made sure to implement the core functionality early on, so just auxiliary things had to be cut.

Secondly, not really an issue, but the TkInter Python GUI framework gave us a bit of trouble. This, however, was not necessarily a function of our team's organization, but rather a function of it being a GUI framework, therefore being inherently difficult to learn and use.

Lastly, we decided to develop our interpreter module by module, rather than outlining the foundation of each module and building up the language, full stack, along the way. This did not necessarily cause us any problems, but is worth mentioning because we could have potentially run out of time with before having a chance to develop a whole module. If we developed the modules concurrently, building up the language as we went, if we ran out of time we would only need to limit the scope of the language, which is comparably much less severe of a problem than missing an entire module.

9.2 Lessons Learned by Each Team Member

Sam Kohn

As project manager, I was responsible for keeping the project on schedule and ensuring sufficient communication between project members. I learned one major lesson from each of these responsibilities. For the first one, I discovered very quickly that I was more concerned on a day-to-day basis with the project's progress than the rest of the team, probably because I was responsible

for the project deliverables' on-time completion. Practically speaking, this meant that I was more motivated to begin work on the modules, and consequently I wrote the beginnings of most of the project's modules. The work balanced out since other team members were eager to work on the project as the deadlines approached and the modules were still not complete. The second lesson I learned involves communication. I was concerned at the beginning of the project that I would be too overbearing as project manager, in that I would be incessantly emailing the other project members and that they would become annoyed with hearing from me. Sure enough, I found myself contacting everyone often enough that I thought I was being too micromanaging. However, based on recent conversations, I learned that I was wrong, and that the communication frequency was appropriate and was welcomed by the other members of our team.

Alex Fields

I learned that it is important to choose carefully the language one is about to write code in before beginning. There was a decent amount of back and forth amongst our group regarding what we should code in before we decided on Python, but this choice seemed to be well worth it.

I learned about the importance of a test suite. Testing was never something that I had done before in a program. I was in charge of writing the Semantic Analyzer, so having the tests at my disposal made my job quite a bit easier. In fact, my Semantic Analyzer failed over half its tests the first time we ran it, and there were many more iterations of coding and testing until it was complete.

I learned that meeting regularly can make a big difference in the end result of a project. I was hesitant to agree that we needed to meet as much as Sam wanted to meet when we began, but I am glad that we did. It allowed us to not reach a time of panic in trying to complete our compiler as I had heard many teams do.

Lastly, I learned that there is always something left to do in a project. Every time we think we have completed something, a bug arises or we remember something we left out. I guess the lesson to be learned from that is plan for many more issues than you expect.

Jeremy Spencer

As system architect I learned the importance of modularization. Breaking a large project into manageable parts is essential for simplicity and clarity. Doing this abstraction early on and integrating the implementation goals to the project plan/deliverables helped to keep everyone on track. The cohesion of each module also made the language's maintenance and expansion easier.

Mason Silber

9.3 Advice for Future Teams

We would highly recommend future teams use Python. As Sam said, "Python was the language through which I felt it was least necessary to think about the rules of the language while coding." Python is, of course, Racecar for adults. That being said, the language being created may look nothing like Python and a different language may be more suitable to the development of that language.

We were satisfied with our decision to make our assigned roles flexible. Having the roles made it easy for us to assign a task if no one immediately volunteered to complete that task, but we were flexible enough to allow everyone to jump around to different roles within the project.

We also would recommend at least a semi-regular meeting schedule. Our team began work early and met regularly. This was key to our success.

9.4 Suggestions for the Instructor

We would like to frame our response in terms of which parts of the course we felt were helpful to us for our project, and which were not. We understand that there are important aspects of compiler design that were not relevant to our particular project but are still worth learning, so we intend this section to be more of a ranking of the impact each topic would have on our project if it were dropped.

Topics we would have liked to see

- More in-depth coverage of semantic analysis (more than just type checking). For example, common ways to keep track of a variable's scope and how to check the appropriate number of function parameters. We knew enough to figure these out on our own, but all of the other components of the compiler received much more thorough treatments, both in class and on the homeworks.
- More examples of lambda calculus (even though it was not directly related to our project). As you probably discovered in the days preceding the final exam, there was a lot of confusion about normal vs. applicative order evaluation. Although explanations are helpful to an extent, what really would have been great was a list of 3 or 4 lambda expressions with step-by-step evaluations in both normal and applicative order. We could not find good examples online.

Topics that directly helped our project

- Lexers

- Common grammar patterns for compilers
- Bottom-up parsing
- Synthetic syntax-directed definitions and translations

Topics that did not help our project, but maybe helped other projects

- Lambda Calculus
- Types and type-checking

Topics that did not help our project or seem relevant to any project

- Top-down parsing. Everyone was advised to use yacc, which uses bottom-up parsing.
- Inherited syntax-directed translation. Again, this would only be useful in top-down parsing.
- Three address codes
- Code optimizations

10 Appendix

The following sections show the source code for the Racecar interpreter and GUI. We have included a simple driver script which opens the GUI, although in actuality the program is packaged into an executable package like any other graphical application.

10.1 Launch Script

Listing 1: `launchRacecar.py`

```
1 import Racecar.RacecarGUI.racecar
```

10.2 Graphical Interface

Listing 2: `racecar.py`

```
1 # Written by Mason Silber and Sam Kohn
2
3 from Tkinter import *
4 from PIL import Image
5 from PIL import ImageTk
```

```

6  import tkinterFileDialog
7  import tkinterMessageBox
8  import re
9  import time
10 import Racecar.Tree
11 import Racecar.Compiler
12 import random
13 import pdb
14 import math
15
16 random.seed()
17
18 #current_program is used to store the current file open in order to save back
19 #to that file
20 current_program = None
21
22 #Variable that serves as an interrupt to stop the program
23 should_stop = False
24 collision_occurred = False
25
26 #List of obstacles on the course at any given time
27 obstacles = []
28
29 #list of walls on the course at any given time
30 walls = []
31
32 #grid ticks
33 grid_ticks = []
34
35
36 class Obstacle:
37     def __init__(self, x, y, width, height):
38         self.obstacle_object = canvas.create_oval(
39             x-width/2,
40             y-height/2,
41             x+width/2,
42             y+height/2,

```

```

43         fill="#000")
44     self.width = width
45     self.height = height
46     self.center = (x, y)
47     self.radius = width/2
48
49
50 #Wall class: can only be vertical or horizontal
51 class Wall:
52     def __init__(self, start_x, start_y, length, is_horizontal):
53         if is_horizontal:
54             self.wall_object = canvas.create_line(
55                 start_x,
56                 start_y,
57                 start_x+length,
58                 start_y)
59             self.start = start_x
60             self.end = start_x+length
61             self.constant_coord = start_y
62         else:
63             self.wall_object = canvas.create_line(
64                 start_x,
65                 start_y,
66                 start_x,
67                 start_y+length)
68             self.start = start_y
69             self.end = start_y+length
70             self.constant_coord = start_x
71         self.is_horizontal = is_horizontal
72
73
74 class Program:
75     def __init__(self):
76         self.name = ''
77         self.file_obj = None
78
79

```

```

80 #Static variables for turning the car
81 class WheelDirection:
82     LEFT = 1
83     RIGHT = -1
84
85
86 #Car direction object
87 #X and Y can be 1,0,-1 respectively. The only invalid combination is when x = 0
88 #and y = 0. Positive axes point right and up respectively
89 class CarDirection:
90     FORWARDS = 1
91     BACKWARDS = -1
92
93     def __init__(self):
94         self.direction = 0
95
96     DIRECTIONS = [(1, 0),
97                  (1, -1),
98                  (0, -1),
99                  (-1, -1),
100                 (-1, 0),
101                 (-1, 1),
102                 (0, 1),
103                 (1, 1)]
104
105     def get_direction(self):
106         return CarDirection.DIRECTIONS[self.direction]
107
108     def turn_right(self):
109         self.direction = (self.direction - 1) % len(CarDirection.DIRECTIONS)
110
111     def turn_left(self):
112         self.direction = (self.direction + 1) % len(CarDirection.DIRECTIONS)
113
114     def opposite_direction(self):
115         return DIRECTIONS[
116             (self.direction + len(CarDirection.DIRECTIONS)/2) %

```

```

117         len(CarDirection.DIRECTIONS)]
118
119
120 class Car:
121     def __init__(self):
122         self.position_x = 0
123         self.position_y = 0
124         #Car direction starts facing right
125         self.car_direction = CarDirection()
126         self.image = None
127         self.image_tk = None
128         self.car_object = None
129         self.width = 27
130         self.height = 27
131         self.radius = 25
132
133     #Drive method that updates the car's position (in the model, not on the UI)
134     #UI animation will need to be done moving x and y simultaneously
135     def update_position(self, steps, movement_direction):
136         self.position_x += (
137             self.car_direction.get_direction()[0]
138             * steps
139             * movement_direction)
140
141         self.position_y += (
142             self.car_direction.get_direction()[1]
143             * steps
144             * movement_direction)
145
146
147     #Function to get a unique position of object, in order to detect for collisions
148     def get_position(x, y):
149         return 1000 * int(x) + int(y)
150
151
152     def getCurrentPosition():
153         global car

```

```

154     return get_position(car.position_x, car.position_y)
155
156
157 #Checks if there is going to be a collision on the upcoming path
158 #drive_direction has to be CarDirection.FORWARDS or CarDirection.BACKWARDS
159 def can_move(num_steps, drive_direction):
160     global car
161     curr_x = int(car.position_x)
162     curr_y = int(car.position_y)
163     direction = car.car_direction.get_direction()
164     path = []
165
166     #If the direction is backwards, just reverse the direction
167     if drive_direction == CarDirection.BACKWARDS:
168         direction = car.car_direction.opposite_direction()
169
170     #Create path coordinates
171     for i in range(0, steps_to_pixels(num_steps)):
172         pos = (curr_x + i * direction[0], curr_y + i * direction[1])
173         path.append(pos)
174
175     #Check each point in the path to see if it collides with any of the
176     #obstacles
177     for pos in path:
178         if is_collision(pos[0], pos[1]):
179             return False
180
181     return True
182
183
184 #Number of steps on screen is proportional to screen size
185 def steps_to_pixels(steps):
186     return canvas_frame.winfo_reqwidth()/110*steps
187
188
189 #Function to find the distance between two points
190 def distance_between_points(x_1, y_1, x_2, y_2):

```



```

191     return math.sqrt(math.pow((x_2-x_1), 2) + math.pow((y_2-y_1), 2))
192
193
194 #API Functions
195 #direction must be either CarDirection.FORWARDS or CarDirection.BACKWARDS
196 def translate_car(steps, direction):
197     global car
198     global should_stop
199     global collision_occurred
200
201     steps = int(steps)
202     direction = int(direction)
203
204     curr_x = car.position_x
205     curr_y = car.position_y
206
207     one_step = steps_to_pixels(1)
208
209     for i in range(0, steps_to_pixels(int(steps))):
210         #Check interrupt variable
211         if should_stop and i % one_step == 0:
212             return
213
214         time.sleep(0.01)
215         #car_direction is FORWARDS or BACKWARDS (1 and -1 respectively)
216
217         if is_collision(curr_x, curr_y):
218             print_to_console("COLLISION")
219             #Stop execution of program
220             #TODO Deal with delay on collision
221             should_stop = True
222             collision_occurred = True
223             return
224         else:
225             canvas.move(
226                 car.car_object,
227                 direction * car.car_direction.get_direction()[0],

```

```

228         direction * car.car_direction.get_direction()[1])
229
230         curr_x = curr_x + direction * car.car_direction.get_direction()[0]
231         curr_y = curr_y + direction * car.car_direction.get_direction()[1]
232         canvas.update()
233
234     car.update_position(1, direction)
235
236
237     #direction must be WheelDirection.LEFT or WheelDirection.RIGHT
238     #Note: only check interrupt variable at the beginning, because
239     #we shouldn't allow partial rotations
240     def rotate_car(direction):
241         global car
242         global should_stop
243
244         #Check interrupt variable
245         if should_stop:
246             return
247
248         #This is current index in DIRECTIONS array
249         current_direction_deg = car.car_direction.direction*45
250
251         if direction == WheelDirection.LEFT:
252             car.car_direction.turn_left()
253         elif direction == WheelDirection.RIGHT:
254             car.car_direction.turn_right()
255         else:
256             return
257
258         for i in range(0, 45):
259             time.sleep(0.01)
260             canvas.delete(car.car_object)
261
262             if direction == WheelDirection.LEFT:
263                 car.image_tk = ImageTk.PhotoImage(
264                     car.image.rotate(current_direction_deg + i))

```

```

265         elif direction == WheelDirection.RIGHT:
266             car.image_tk = ImageTk.PhotoImage(
267                 car.image.rotate(current_direction_deg - i))
268         else:
269             return
270
271     car.car_object = canvas.create_image(
272         car.position_x,
273         car.position_y,
274         image=car.image_tk)
275     canvas.update()
276
277
278     #Check for collision with walls of the maze and a finish line
279     def collision_with_internal_walls(pos_x, pos_y):
280         for wall in walls:
281             #horizontal wall
282             if wall.is_horizontal:
283                 #in range of wall
284                 if wall.start <= pos_x <= wall.end:
285                     #Current direction of the car
286                     direction = car.car_direction.get_direction()
287                     #distance from wall
288                     dist_to_wall = math.fabs(pos_y-wall.constant_coord)
289                     #Car is horizontally oriented
290                     if direction == (1, 0) or direction == (-1, 0):
291                         if dist_to_wall < car.radius/2:
292                             return True
293                     #Check for collision with car
294                     #Car is not horizontally oriented
295                     else:
296                         if dist_to_wall < car.radius:
297                             return True
298             #vertical wall
299         else:
300             #in range of wall
301             if wall.start <= pos_y <= wall.end:

```

```

302         direction = car.car_direction.get_direction()
303         #distance from wall
304         dist_to_wall = math.fabs(pos_x-wall.constant_coord)
305         #Car is vertically oriented
306         if direction == (0, 1) or direction == (0, -1):
307             if dist_to_wall < car.radius/2:
308                 return True
309         #Car is not vertically oriented
310         else:
311             if dist_to_wall < car.radius:
312                 return True
313
314     return False
315
316
317 def is_collision(curr_x, curr_y):
318     #Check for collisions with obstacles and walls
319
320     #Check obstacles
321     for obstacle in obstacles:
322         distance = distance_between_points(
323             curr_x,
324             curr_y,
325             obstacle.center[0],
326             obstacle.center[1])
327         if distance < (car.radius + obstacle.radius):
328             return True
329     #check internal walls for collision
330     if collision_with_internal_walls(curr_x, curr_y):
331         return True
332     #Check boundary walls
333     elif not (origin[0] <= curr_x <= anti_origin[0]):
334         return True
335     elif not (origin[1] <= curr_y <= anti_origin[1]):
336         return True
337     else:
338         return False

```

```

339
340
341 def print_to_console(message):
342     #Should console be cleared each time the program is restart?
343     #Or should there be a button?
344     console.config(state=NORMAL)
345     console.insert(END, str(message) + '\n')
346     console.config(state=DISABLED)
347
348
349 #Course generation functions
350
351 #Course one is a slalom of blocks
352 def course_one():
353     clear_course()
354     obstacle_coord_x = 123
355     obstacle_coord_y = int(canvas.winfo_reqheight())/2
356     while obstacle_coord_x < anti_origin[0]:
357         obstacle = Obstacle(obstacle_coord_x, obstacle_coord_y, 30, 30)
358         obstacles.append(obstacle)
359         obstacle_coord_x = obstacle_coord_x + 150
360
361
362 #TODO -- Fill in the rest of the courses
363 #Course two is a simple maze
364 finish_line = None
365
366
367 def course_two():
368     global finish_line
369
370     clear_console()
371
372     message = "Try to navigate through the maze and cross the finish line!"
373     print_to_console(message)
374
375     clear_course()

```

```

376     wall_coord_x = 123
377     wall_length = 4*int(canvas.wininfo_reqheight())/5
378
379     #used to toggle position of line
380     put_wall_on_top = True
381
382     #walls
383     while wall_coord_x < anti_origin[0]:
384         if put_wall_on_top:
385             wall = Wall(
386                 wall_coord_x,
387                 0,
388                 wall_length,
389                 False)
390             walls.append(wall)
391         else:
392             wall = Wall(
393                 wall_coord_x,
394                 int(canvas.wininfo_reqheight())/5+23,
395                 wall_length,
396                 False)
397             walls.append(wall)
398             put_wall_on_top = not put_wall_on_top
399             wall_coord_x = wall_coord_x+100
400
401     #finish line
402     wall_coord_x = wall_coord_x-100
403     finish_line = canvas.create_line(
404         wall_coord_x,
405         wall_length,
406         wall_coord_x,
407         canvas.wininfo_reqheight()+23,
408         fill="black",
409         dash=(4, 4))
410
411
412     def course_three():

```

```

413     clear_course()
414
415     max_x = canvas.winfo_reqwidth()
416     max_y = canvas.winfo_reqheight()
417
418     while len(obstacles) < 30:
419         pos_x = random.randrange(0, max_x, 1)
420         pos_y = random.randrange(0, max_y, 1)
421         radius = random.randrange(10, 50, 1)
422
423         if is_collision(pos_x, pos_y):
424             continue
425         #Check for collision with car
426         elif distance_between_points(
427             pos_x,
428             pos_y,
429             car.position_x,
430             car.position_y) < (car.radius + radius):
431             continue
432         else:
433             obstacle = Obstacle(pos_x, pos_y, radius, radius)
434             obstacles.append(obstacle)
435
436
437     def course_four():
438         clear_course()
439         obstacle_coord_x = 123
440         obstacle_coord_y = 60
441         while obstacle_coord_y < anti_origin[1]:
442             while obstacle_coord_x < anti_origin[0]:
443                 obstacle = Obstacle(obstacle_coord_x, obstacle_coord_y, 30, 30)
444                 obstacles.append(obstacle)
445                 obstacle_coord_x = obstacle_coord_x + 150
446             obstacle_coord_y = obstacle_coord_y + 80
447             obstacle_coord_x = 123
448
449         for obstacle in obstacles:

```

```

450         print obstacle.center
451
452
453 def course_five():
454     clear_course()
455
456
457 def clear_course():
458     global obstacles
459     global walls
460     global finish_line
461     #remove obstacles from the course
462     for obstacle in obstacles:
463         canvas.delete(obstacle.obstacle_object)
464
465     #Needs to take care of finish line too, which isn't a wall object
466     for wall in walls:
467         canvas.delete(wall.wall_object)
468
469     if finish_line is not None:
470         canvas.delete(finish_line)
471     #clear the obstacles and walls array
472     obstacles = []
473     walls = []
474     finish_line = None
475
476
477 #Menu functions
478 def open_file():
479     global current_program
480
481     #Keep returning to the file dialog if they didn't select a .race file
482     while True:
483         file_name = tkFileDialog.askopenfilename(defaultextension=".race")
484         if file_name == '':
485             return
486

```



```

487         #Check validity of file being opened
488         file_regex = re.compile("\w*\.race$")
489         if len(file_regex.findall(file_name)) == 0:
490             tkinter.messagebox.showwarning(
491                 "Open File Error",
492                 "You must open a .race file")
493         else:
494             break
495
496         file_object = open(file_name, 'r')
497         current_program = Program()
498         current_program.name = file_name
499         current_program.file_obj = file_object
500         code.delete(1.0, END)
501         code.insert(1.0, file_object.read())
502         current_program.file_obj.close()
503
504
505     def save():
506         global current_program
507         if current_program is None:
508             save_file_as()
509         else:
510             save_file()
511
512
513     def save_file():
514         global current_program
515         if not current_program.file_obj.closed:
516             current_program.file_obj.close()
517         #Open file for writing (will clear it)
518         current_program.file_obj = open(current_program.name, 'w')
519         current_program.file_obj.truncate()
520         current_program.file_obj.write(code.get(1.0, END))
521         current_program.file_obj.close()
522
523

```

```

524 def save_file_as():
525     global current_program
526     file_name = tkFileDialog.asksaveasfilename(defaultextension=".race")
527
528     #Defaults to saving on the desktop
529     if file_name == '':
530         file_name = '~/Desktop/racecar_program.race'
531
532     current_program = Program()
533     current_program.name = file_name
534     current_program.file_obj = open(file_name, 'w')
535     current_program.file_obj.write(code.get(1.0, END))
536     current_program.file_obj.close()
537
538
539 def clear():
540     if code.get(1.0, END) == '':
541         return
542
543     if tkMessageBox.askyesno(
544         "Clear code",
545         "Are you sure you want to delete all of your code?"):
546         code.delete(1.0, END)
547
548
549 def clear_console():
550     console.config(state=NORMAL)
551     console.delete(1.0, END)
552     console.config(state=DISABLED)
553
554
555 #Triggers interrupt
556 def stop_program():
557     global should_stop
558     should_stop = True
559
560

```

```

561 #Code generation and compilation
562 #Runs code
563 def generate_program(code):
564     global should_stop
565     global collision_occurred
566
567     #Set the interrupt variable whenever a program is run
568     should_stop = False
569     collision_occurred = False
570     if len(code) > 1:
571         #print code[:-1]
572         #demo(code)
573         python_code, errors, correct = verify_program(code)
574         if(correct):
575             #Print message to console saying program is executing
576             print_to_console("Program executing")
577             console.tag_add("Correct", "1.0", "1.end")
578             console.tag_config("Correct", foreground="Green")
579
580             #Toggle the buttons on the bottom and run program
581             toggle_buttons(True)
582             tempGlobal = globals().copy()
583             exec(python_code, tempGlobal)
584             toggle_buttons(False)
585
586             #If collision occurred
587             if should_stop:
588                 if collision_occurred:
589                     if tkMessageBox.showwarning(
590                         "Oops!", "You crashed! Try again"):
591                         reset_car_position()
592             else:
593                 #Print message to console saying program is finished executing
594                 print_to_console("Done running program")
595                 console.tag_add("End", "end -2 1", END)
596                 console.tag_config("End", foreground="Green")
597                 print "OUTPUT: " + console.index("end -1 1")

```

```

598         else:
599             #Print message to console saying program has errors
600             print_to_console(
601                 "You have " +
602                 str(len(errors)) +
603                 " error(s) in your program")
604             console.tag_add("Error", "1.0", "1.end")
605             console.tag_config("Error", foreground="Red")
606
607             for error in errors:
608                 print_to_console(error)
609     else:
610         print "Blank"
611
612
613 #Checks if program is a valid Racecar program and returns corresponding python
614 #code if necessary
615 def verify_program(code):
616     clear_console()
617     if len(code) < 2:
618         return ("BLANK", False)
619     code, errors = Racecar.Compiler.getPythonCode(code)
620     if errors:
621         return (code, errors, False)
622     else:
623         return (code, errors, True)
624
625
626 #Called when verify program is called
627 def verify_program_callback(code):
628     verification = verify_program(code)
629     if verification[2]:
630         print_to_console("Program syntax correct")
631         console.tag_add("Correct", "1.0", "1.end")
632         console.tag_config("Correct", foreground="Green")
633     else:
634         errors = verification[1]

```

```

635         print_to_console(
636             "You have " +
637             str(len(errors)) +
638             " error(s) in your program")
639         console.tag_add("Error", "1.0", "1.end")
640         console.tag_config("Error", foreground="Red")
641
642         for error in errors:
643             print_to_console(error)
644
645
646     #Resets car's position and orientation to original
647     def reset_car_position():
648         global car
649         canvas.delete(car.car_object)
650         car.image_tk = ImageTk.PhotoImage(car.image)
651         car_height = int(canvas.winfo_reqheight())/2
652         car.car_object = canvas.create_image(
653             23,
654             car_height,
655             image=car.image_tk)
656         car.position_x = 23
657         car.position_y = car_height
658         car.car_direction = CarDirection()
659
660     #car object
661     car = Car()
662
663
664     #User interface
665     #Toggle enabled and disabled buttons when program is run and stopped
666     def toggle_buttons(stop_button_should_be_enabled):
667         if stop_button_should_be_enabled:
668             run_button.config(state=DISABLED)
669             stop_button.config(state=NORMAL)
670             reset_car_position_button.config(state=DISABLED)
671             clear_button.config(state=DISABLED)

```

```

672     else:
673         run_button.config(state=NORMAL)
674         stop_button.config(state=DISABLED)
675         reset_car_position_button.config(state=NORMAL)
676         clear_button.config(state=NORMAL)
677
678 root = Tk()
679 root.title('Racecar')
680 #Height is always three fourths the width of the window
681 window_width = root.winfo_screenwidth() - 100
682 window_height = 9*window_width/16
683 root.geometry("%dx%d" % (window_width, window_height))
684 root.resizable(width=FALSE, height=FALSE)
685
686 menu_bar = Menu(root)
687
688 menu = Menu(menu_bar, tearoff=0)
689 menu.add_command(label="Open", command=open_file)
690 menu.add_command(label="Save", command=save)
691 menu.add_command(label="Save As", command=save_file_as)
692 menu.add_separator()
693 menu.add_command(label="Quit", command=exit)
694 menu_bar.add_cascade(label="File", menu=menu)
695
696 menu = Menu(menu_bar, tearoff=0)
697
698 command = lambda: verify_program_callback(code.get(1.0, END))
699 menu.add_command(label="Verify Code", command=command)
700
701 command = lambda: generate_program(code.get(1.0, END))
702 menu.add_command(label="Run Code", command=command)
703
704 menu.add_command(label="Clear Code", command=clear)
705 menu.add_command(label="Clear Console", command=clear_console)
706 menu_bar.add_cascade(label="Code", menu=menu)
707
708 menu = Menu(menu_bar, tearoff=0)

```

```

709 menu.add_command(label="Course 1", command=course_one)
710 menu.add_command(label="Course 2", command=course_two)
711 menu.add_command(label="Course 3", command=course_three)
712 menu.add_command(label="Course 4", command=course_four)
713 menu.add_command(label="Course 5", command=course_five)
714 menu.add_separator()
715 menu.add_command(label="Clear course", command=clear_course)
716 menu_bar.add_cascade(label="Courses", menu=menu)
717
718 root.config(menu=menu_bar)
719
720 #frame for left side of window
721 left_frame = Frame(root)
722
723 #label for code window
724 code_label = Label(left_frame, text="Enter code here", anchor=W, pady=5)
725
726 #frame for code window to hold textbox and scrollbar
727 code_frame = Frame(
728     left_frame,
729     width=int(0.3*window_width),
730     height=9*window_height/10)
731 code_frame.grid_propagate(False)
732
733 #scrollbar for code window
734 code_scrollbar = Scrollbar(code_frame)
735 code_scrollbar.pack(side=RIGHT, fill=Y)
736
737 #code is the window in which the code is written
738 code = Text(
739     code_frame,
740     width=50,
741     #height=window_height/16-8,
742     wrap=WORD,
743     yscrollcommand=code_scrollbar.set)
744
745 #Frame for buttons

```

```

746 button_frame = Frame(left_frame)
747
748 #run_button passes code into a run program method
749 command = lambda: generate_program(code.get(1.0, END))
750 run_button = Button(
751     button_frame,
752     text="Run Code",
753     pady=5,
754     padx=5,
755     command=command)
756
757 #Stop execution of running program
758 stop_button = Button(
759     button_frame,
760     text="Stop Program",
761     padx=5,
762     pady=5,
763     command=stop_program)
764 stop_button.config(state=DISABLED)
765
766 #reset car position button puts the car back in its original position and
767 #orientation
768 reset_car_position_button = Button(
769     button_frame,
770     text="Reset Car Position",
771     pady=5,
772     padx=5,
773     command=reset_car_position)
774
775 #clear_button clears the code in the text box
776 clear_button = Button(
777     button_frame,
778     text="Clear Code",
779     command=clear)
780
781 #canvas is where the car will go
782 canvas_frame = Frame(

```



```

783     root,
784     width=window_width/1.5,
785     height=window_height/1.5,
786     padx=2,
787     pady=2)
788
789 canvas_frame.configure(borderwidth=1.5, background='black')
790 canvas = Canvas(
791     canvas_frame,
792     width=window_width/1.5,
793     height=window_height/1.5)
794
795 car.image = Image.open('Racecar/RacecarGUI/images/racecar.png')
796 car.image_tk = ImageTk.PhotoImage(car.image)
797
798 car.car_object = canvas.create_image(
799     23,
800     int(canvas.winfo_reqheight())/2,
801     image=car.image_tk)
802
803 car.position_x = 23
804 car.position_y = int(canvas.winfo_reqheight())/2
805
806 #label above the console
807 console_label = Label(root, text="Console", anchor=W, pady=5)
808
809 #frame for the console to hold the textbox and the scrollbar
810 console_frame = Frame(root)
811
812 #scrollbar for the console
813 console_scrollbar = Scrollbar(console_frame)
814 console_scrollbar.pack(side=RIGHT, fill=Y)
815
816 #console to print to
817 console = Text(
818     console_frame,
819     width=int(window_width/1.5),

```

```

820     height=8,
821     padx=2,
822     pady=2,
823     wrap=WORD,
824     yscrollcommand=console_scrollbar.set)
825
826 console.config(state=DISABLED)
827
828 #add them to GUI Window
829 #These are grouped logically in order to better see what's going on
830 left_frame.pack(side=LEFT, fill=BOTH)
831
832 code_label.pack()
833
834 code_frame.pack(expand=1, fill=BOTH)
835 code.pack(expand=1, fill=BOTH)
836
837 button_frame.pack(fill=BOTH)
838 run_button.grid(row=1, column=1)
839 stop_button.grid(row=1, column=2)
840 reset_car_position_button.grid(row=1, column=3)
841 clear_button.grid(row=1, column=4)
842
843 canvas_frame.pack(expand=1, fill=BOTH)
844 canvas.pack(expand=1, fill=BOTH)
845
846 console_label.pack()
847
848 console_frame.pack(expand=1, fill=BOTH, pady=(0, 10))
849 console.pack(expand=1, fill=BOTH)
850
851 code_scrollbar.config(command=code.yview)
852 console_scrollbar.config(command=console.yview)
853
854 root.update_idletasks()
855
856 #Origin and antiorigin are limits on the canvas where the car moves

```

```

857 origin = (23, 26)
858 anti_origin = (
859     23+106*canvas_frame.wininfo_width()/110,
860     26+56*canvas_frame.wininfo_width()/110)
861
862 #horizontal grid lines
863 position = 0
864 while position < anti_origin[0]:
865     tick = canvas.create_line(
866         position,
867         anti_origin[1]-5+35,
868         position,
869         anti_origin[1]+35,
870         fill="#000",
871         width=2)
872     grid_ticks.append(tick)
873     position += steps_to_pixels(5)
874
875 #vertical grid lines
876 position = 0
877 while position < anti_origin[1]:
878     tick = canvas.create_line(
879         0,
880         position,
881         5,
882         position,
883         fill="#000",
884         width=2)
885     grid_ticks.append(tick)
886     position += steps_to_pixels(5)
887
888 print anti_origin
889 #Run the GUI
890 root.mainloop()

```

10.3 Lexer and Parser

Listing 3: Parser.py

```

1  # Written by Sam Kohn and Jeremy Spencer
2
3  import ply.lex as lex
4  import ply.yacc as yacc
5  from Tree import *
6
7  reserved = {
8      'drive': 'DRIVE',
9      'forward': 'FORWARD',
10     'forwards': 'FORWARD',
11     'backward': 'BACKWARD',
12     'backwards': 'BACKWARD',
13     'number': 'NUMBER_TYPE',
14     'word': 'WORD_TYPE',
15     'step': 'STEP',
16     'steps': 'STEP',
17     'turn': 'TURN',
18     'left': 'LEFT',
19     'right': 'RIGHT',
20     'canDrive': 'CAN_DRIVE',
21     'getCarPosition': 'GET_CAR_POSITION',
22     'define': 'DEFINE',
23     'using': 'USING',
24     'and': 'AND',
25     'print': 'PRINT',
26     'elseIf': 'ELSE_IF',
27     'if': 'IF',
28     'else': 'ELSE',
29     'repeat': 'REPEAT',
30     'times': 'TIMES',
31     'a': 'A',
32     'is': 'IS',
33     'not': 'NOT',
34     'set': 'SET',
35     'to': 'TO',
36 }

```

```

37
38
39 tokens = [
40     "NUMBER",
41     "WORD",
42     "ID",
43     "GT",
44     "LT",
45     "GEQ",
46     "LEQ",
47     "CONCAT",
48     "NEWLINE",
49     "SINGLE_LINE_COMMENT",
50 ] + list(set(reserved.values()))
51
52 literals = "{}()+-*/"
53
54 #t_NUMBER = r'[0-9]+'
55 #t_WORD = r'".*?"'
56 t_GT = r'>'
57 t_LT = r'<'
58 t_GEQ = r'>='
59 t_LEQ = r'<='
60 t_CONCAT = r'\+\+'
61 t_SINGLE_LINE_COMMENT = r':\).*$\n'
62 t_ignore = ' \t'
63
64
65 def t_ID(t):
66     r'[A-Za-z][A-Za-z0-9]*'
67     t.type = reserved.get(t.value, 'ID')
68     t.value = (t.value, t.type, t.lexer.lineno)
69     return t
70
71
72 def t_NUMBER(t):
73     r'[0-9]+'

```

```

74     t.value = (t.value, t.type, t.lexer.lineno)
75     return t
76
77
78 def t_WORD(t):
79     r'".*?"'
80     t.value = (t.value, t.type, t.lexer.lineno)
81     return t
82
83
84 def t_NEWLINE(t):
85     r'\n|;|:-\((.|\\n)*?:-\\)'
86     # \n is for actual newlines
87     # ; is for debugging use
88     # the next expression is for multiline comments. it is an adaptation of
89     # hwl, problem 2.
90     # the last expression :\\).* matches single-line comments
91     t.lexer.lineno += 1
92     return t
93
94
95 def t_error(t):
96     print "Illegal character '%s' at line '%s'" % (t.value[0], t.lexer.lineno)
97     t.lexer.skip(1)
98     t.value = (t.value, "ERROR", t.lexer.lineno)
99     return t
100
101
102 def p_error(p):
103     if p is None:
104         raise SyntaxError("Reached end of file unexpectedly!")
105     elif p.value[0] is None:
106         print "Lexing Error with character ", p.value[1]
107         p.value = p.value[1]
108     else:
109         print "Syntax error at token ", p.type
110

```

```

111
112 def makeParseTreeNode(p, value):
113     '''Returns a Tree object containing
114         as children p[1:] and a value of value'''
115     toReturn = Tree()
116     for element in p[1:]:
117         if type(element) == type(toReturn):
118             toReturn.children.append(element)
119             toReturn.errors += element.errors
120         else:
121             # the element is not a tree. wrap it in a tree
122             newElement = Tree()
123             if isinstance(element, tuple):
124                 newElement.value = element[0]
125                 newElement.type = element[1]
126             else:
127                 newElement.value = element
128             toReturn.children.append(newElement)
129
130     if isinstance(value, tuple):
131         toReturn.value = value[0]
132         toReturn.type = value[1]
133     else:
134         toReturn.value = value
135     if value == "error":
136         errorMessage = str(p[1][2]) + ": " + p[1][0]
137         toReturn.errors.append(errorMessage)
138
139     return toReturn
140
141
142 def p_statements(p):
143     '''statements : statements statement'''
144     p[0] = makeParseTreeNode(p, "statements")
145
146
147 def p_error_statement(p):

```

```

148     '''statement : error NEWLINE'''
149     if not isinstance(p[1], tuple):
150         p[1] = p[1].value
151     p[0] = makeParseTreeNode(p, "error")
152
153
154 def p_statements_empty(p):
155     '''statements : empty'''
156     p[0] = p[1]
157
158
159 def p_statement_block(p):
160     """statement_block : '{' statements '}' newline_opt_comment"""
161     p[0] = makeParseTreeNode([p[0], p[2]], "statement_block")
162
163
164 def p_empty(p):
165     '''empty :'''
166     p[0] = Tree()
167     p[0].value = "empty"
168
169
170 def p_newline_opt_comment(p):
171     '''newline_opt_comment : opt_comment NEWLINE'''
172     p[0] = p[2]
173
174
175 def p_opt_comment(p):
176     '''opt_comment : SINGLE_LINE_COMMENT
177         | empty'''
178     p[0] = p[1]
179
180
181 def p_statement_simple_compound(p):
182     '''statement : simple_statement
183         | compound_statement'''
184     p[0] = p[1]

```



```

185
186
187 def p_simple_statement_command(p):
188     '''simple_statement : statement_contents newline_opt_comment'''
189     p[0] = p[1]
190
191
192 def p_statement_newline(p):
193     '''simple_statement : newline_opt_comment'''
194     p[0] = Tree()
195     p[0].value = "empty"
196
197
198 def p_statement_contents_drive(p):
199     '''statement_contents : drive_command'''
200     p[0] = p[1]
201
202
203 def p_statement_contents_turn(p):
204     '''statement_contents : turn_command'''
205     p[0] = p[1]
206
207
208 def p_compound_statement_define(p):
209     '''compound_statement : define_command'''
210     p[0] = p[1]
211
212
213 def p_compound_statement_repeat_if(p):
214     '''compound_statement : repeat_if_command'''
215     p[0] = p[1]
216
217
218 def p_compound_statement_repeat_times(p):
219     '''compound_statement : repeat_times_command'''
220     p[0] = p[1]
221

```

```

222
223 def p_compound_statement_if(p):
224     '''compound_statement : if_command'''
225     p[0] = p[1]
226
227
228 def p_statement_contents_print(p):
229     '''statement_contents : print_command'''
230     p[0] = p[1]
231
232
233 def p_statement_contents_assignment(p):
234     '''statement_contents : assignment_command'''
235     p[0] = p[1]
236
237
238 def p_statement_contents_declaration(p):
239     '''statement_contents : declaration_command'''
240     p[0] = p[1]
241
242
243 def p_statement_contents_function(p):
244     '''statement_contents : function_command'''
245     p[0] = p[1]
246
247
248 def p_expression_can_drive(p):
249     '''expression : can_drive_expression'''
250     p[0] = p[1]
251
252
253 def p_expression_comparison(p):
254     '''expression : comparison'''
255     p[0] = p[1]
256
257
258 def p_can_drive(p):

```

```

259     '''can_drive_expression : CAN_DRIVE drive_direction \
260     primary_expression opt_steps'''
261     p[0] = makeParseTreeNode([p[0], p[2], p[3]], "can_drive_expression")
262
263
264 def p_comparison_with_operator(p):
265     '''comparison : plus_expression comparison_operator plus_expression'''
266     p[0] = makeParseTreeNode(p, "comparison")
267
268
269 def p_comparison_plus(p):
270     '''comparison : plus_expression'''
271     p[0] = p[1]
272
273
274 def p_comparison_operator(p):
275     '''comparison_operator : IS
276         | IS NOT
277         | GT
278         | LT
279         | GEQ
280         | LEQ'''
281     if len(p) == 3: # i.e. token is IS NOT
282         p[0] = (p[1][0] + " " + p[2][0], p[1][1] + " " + p[2][1])
283     else: # any other token
284         p[0] = p[1]
285
286
287 def p_plus_expression_plus_minus(p):
288     '''plus_expression : plus_expression '+' times_expression
289         | plus_expression '-' times_expression'''
290     p[0] = makeParseTreeNode(p, "plus_expression")
291
292
293 def p_plus_expression_times_expression(p):
294     '''plus_expression : times_expression'''
295     p[0] = p[1]

```

```

296
297
298 def p_times_expression_times_divide(p):
299     '''times_expression : times_expression '*' word_expression
300       | times_expression '/' word_expression'''
301     p[0] = makeParseTreeNode(p, "times_expression")
302
303
304 def p_times_expression_word_expression(p):
305     '''times_expression : word_expression'''
306     p[0] = p[1]
307
308
309 def p_word_expression_concat(p):
310     '''word_expression : word_expression CONCAT primary_expression'''
311     p[0] = makeParseTreeNode(p, "word_expression")
312
313
314 def p_word_expression_primary_expression(p):
315     '''word_expression : primary_expression'''
316     p[0] = p[1]
317
318
319 def p_primary_expression_parens(p):
320     '''primary_expression : '(' expression ')' '''
321     p[0] = p[2]
322
323
324 def p_primary_expression_token(p):
325     '''primary_expression : NUMBER
326       | WORD
327       | GET_CAR_POSITION
328       | ID'''
329     p[0] = p[1]
330
331
332 def p_function_command(p):

```

```

333     '''function_command : ID opt_parameters'''
334     if p[2].value == "empty":
335         p[0] = makeParseTreeNode([p[0], p[1]], "function_command")
336     else:
337         p[0] = makeParseTreeNode(p, "function_command")
338
339
340 def p_opt_parameters(p):
341     '''opt_parameters : opt_parameters primary_expression'''
342     if p[1].value == "empty":
343         p[0] = makeParseTreeNode([p[0], p[2]], "opt_parameters")
344     else:
345         p[0] = makeParseTreeNode(p, "opt_parameters")
346
347
348 def p_opt_parameters_empty(p):
349     '''opt_parameters : empty'''
350     p[0] = p[1]
351
352
353 def p_drive_command(p):
354     '''drive_command : DRIVE drive_direction plus_expression opt_steps'''
355     p[0] = makeParseTreeNode([p[0], p[2], p[3]], "drive_command")
356
357
358 def p_drive_direction(p):
359     '''drive_direction : FORWARD
360     | BACKWARD'''
361     p[0] = p[1]
362
363
364 def p_opt_steps(p):
365     '''opt_steps : STEP
366     | empty'''
367     p[0] = p[1]
368
369

```

```

370 def p_turn_command(p):
371     '''turn_command : TURN turn_direction'''
372     p[0] = makeParseTreeNode(p, "turn_command")
373
374
375 def p_turn_direction(p):
376     '''turn_direction : LEFT
377         | RIGHT'''
378     p[0] = p[1]
379
380
381 def p_define_command(p):
382     """define_command : DEFINE ID opt_param_list \
383     newline_opt_comment statement_block"""
384     p[0] = makeParseTreeNode([p[0], p[2], p[3], p[5]], "define_command")
385
386
387 def p_opt_param_list(p):
388     '''opt_param_list : USING ID '(' type_enum ')' opt_extra_params'''
389     p[0] = makeParseTreeNode(p, "opt_param_list")
390
391
392 def p_opt_param_list_empty(p):
393     '''opt_param_list : empty'''
394     p[0] = p[1]
395
396
397 def p_opt_extra_params(p):
398     '''opt_extra_params : AND ID '(' type_enum ')' opt_extra_params'''
399     p[0] = makeParseTreeNode(p, "opt_extra_params")
400
401
402 def p_opt_extra_params_empty(p):
403     '''opt_extra_params : empty'''
404     p[0] = p[1]
405
406

```

```

407 def p_type_enum(p):
408     '''type_enum : WORD_TYPE
409         | NUMBER_TYPE'''
410     p[0] = p[1]
411
412
413 def p_repeat_if_command(p):
414     """repeat_if_command : REPEAT IF expression newline_opt_comment \
415     statement_block"""
416     p[0] = makeParseTreeNode(p, "repeat_if_command")
417
418
419 def p_repeat_times_command(p):
420     """repeat_times_command : REPEAT plus_expression \
421     TIMES newline_opt_comment statement_block"""
422     p[0] = makeParseTreeNode(p, "repeat_times_command")
423
424
425 def p_if_command(p):
426     """if_command : IF expression newline_opt_comment statement_block \
427     opt_else_if opt_else"""
428     p[0] = makeParseTreeNode(p, "if_command")
429
430
431 def p_opt_else_if(p):
432     """opt_else_if : ELSE_IF expression newline_opt_comment \
433     statement_block opt_else_if
434     | empty"""
435
436     if len(p) == 2:
437         p[0] = p[1]
438     else:
439         p[0] = makeParseTreeNode(p, "opt_else_if")
440
441
442 def p_opt_else(p):
443     """opt_else : ELSE newline_opt_comment statement_block

```

```

444         | empty"""
445
446     if len(p) == 2:
447         p[0] = p[1]
448     else:
449         p[0] = makeParseTreeNode(p, "opt_else")
450
451
452 def p_print_command(p):
453     """print_command : PRINT word_expression"""
454     p[0] = makeParseTreeNode(p, "print")
455
456
457 def p_declaration_command(p):
458     """declaration_command : ID IS A type_enum"""
459     p[0] = makeParseTreeNode(p, "declaration_command")
460
461
462 def p_assignment_command(p):
463     """assignment_command : SET ID TO expression"""
464     p[0] = makeParseTreeNode(p, "assignment_command")
465
466
467 def parseString(stringToParse):
468     '''Returns the parse tree for the given string'''
469     lexer = lex.lex()
470     parser = yacc.yacc()
471     return parser.parse(stringToParse)
472
473 if __name__ == "__main__":
474     lexer = lex.lex()
475     parser = yacc.yacc()
476     inputString = ''
477     while True:
478
479         inputString = raw_input('enter expression > ')
480

```



```

481         if inputString == 'exit':
482             break
483
484         else:
485             try:
486                 result = parser.parse(inputString)
487             except SyntaxError as e:
488                 print "Error: ", e
489             else:
490                 result.printTree()
491                 print
492                 print "errors: ", result.errors

```

10.4 Semantic Analyzer and Symbol Table

Listing 4: SemanticAnalyzer.py

```

1  # Written by Alex Fields and Colfax Selby
2
3  # Scoping done using a universal count, which is a unique number for
4  # every single scope
5
6  from SymbolTable import *
7  import Parser
8
9  table = None
10 count = 0
11 function = None
12 scopeList = [0]
13 errorList = []
14 firstPass = True
15
16
17 def analyzeStart(ast):
18     # this block for testing purposes
19     global table, count, function, scopeList, errorList, firstPass
20     table = SymbolLookupTable()
21     count = 0
22     function = None

```

```

23     scopeList = [0]
24     errorList = []
25     firstPass = True
26
27     analyze(ast)
28     # TODO uncomment after removing testing code
29     # global firstPass, count, function, scopeList
30     count = 0
31     function = None
32     firstPass = False
33     scopeList = [0]
34     analyze(ast)
35     return errorList
36
37
38 def analyze(ast):
39     '''Traverse the AST and check for semantic errors.'''
40
41     # potential AST values and their associated analysis functions
42     # use astAnalyzers.get() instead of a long chain of else-ifs
43     astAnalyzers = {
44         "assignment_command": assignmentCommandAnalyzer,
45         "comparison": comparisonAnalyzer,
46         "declaration_command": declarationCommandAnalyzer,
47         "define_command": defineCommandAnalyzer,
48         "drive_command": driveCommandAnalyzer,
49         "empty": emptyAnalyzer,
50         "function_command": functionCommandAnalyzer,
51         "if_command": ifCommandAnalyzer,
52         "opt_else": optElseAnalyzer,
53         "opt_else_if": optElseIfAnalyzer,
54         "opt_extra_params": optExtraParamsAnalyzer,
55         "opt_param_list": optParamListAnalyzer,
56         "plus_expression": plusExpressionAnalyzer,
57         "print": printAnalyzer,
58         "repeat_if_command": repeatIfAnalyzer,
59         "repeat_times_command": repeatTimesAnalyzer,

```

```

60     "statement_block": statementBlockAnalyzer,
61     "statements": statementsAnalyzer,
62     "times_expression": timesExpressionAnalyzer,
63     "turn_command": turnCommandAnalyzer,
64     "word_expression": wordExpressionAnalyzer,
65 }
66
67 # Fetch the appropriate analyzer function from astAnalyzers
68 # If there is no analyzer for ast.value, then just let the
69 # "analyzer" be ast.value
70 analyzer = astAnalyzers.get(ast.value, ast.value)
71
72 # If the "analyzer" is just a string (inherits from basestring)
73 if isinstance(analyzer, basestring):
74     # this should only be useful for evaluating the type of an expression
75     if (ast.type == "WORD"):
76         return "word"
77     elif (ast.type == "NUMBER"):
78         return "number"
79     elif (ast.type == "ID"):
80         # do existence and scope checking right here
81         # return type if passes
82         id = ast.value
83         idEntry = table.getEntry(SymbolTableEntry(
84             id, None, list(scopeList), function, None))
85         if idEntry is None:
86             # ID does not exist or exists but the scoping is wrong
87             # this is to be returned to binaryOperatorAnalyzer
88             return "ERROR"
89         if function is None and not idEntry.initialized:
90             return "ERROR"
91         else:
92             return idEntry.type
93
94 # if the translator is a real function, then invoke it
95 else:
96     return analyzer(ast)

```

```

97
98
99 def statementsAnalyzer(ast):
100     if firstPass:
101         if (ast.children[0].value == "define_command" or
102             ast.children[0].value == "statements"):
103             analyze(ast.children[0])
104         if (ast.children[1].value == "define_command" or
105             ast.children[1].value == "statements"):
106             analyze(ast.children[1])
107     else:
108         analyze(ast.children[0])
109         analyze(ast.children[1])
110
111
112 def driveCommandAnalyzer(ast):
113     # for "plus_expression"
114     result = analyze(ast.children[1])
115     if result != "number":
116         errorList.append("Error in drive command: \
117             need to use valid variable or number")
118
119
120 def turnCommandAnalyzer(ast):
121     # nothing to do here
122     return
123
124
125 def emptyAnalyzer(ast):
126     return []
127
128
129 def comparisonAnalyzer(ast):
130     result = binaryOperatorAnalyzer(ast)
131     if result == "number":
132         return
133     elif result == "word":

```

```

134         if ast.children[1].type == "IS" or ast.children[1].type == "IS NOT":
135             return
136         else:
137             errorList.append("Error in comparison: \
138                 words must be compared using 'is' or 'is not'")
139     else:
140         errorList.append("Error in comparison: \
141             use only words or only numbers; cannot mix both")
142
143
144 def optElseIfAnalyzer(ast):
145     # for "expression"
146     if ast.children[1].value != "empty":
147         analyze(ast.children[1])
148     # for "statement_block"
149     if ast.children[3].value != "empty":
150         analyze(ast.children[3])
151     # for "optional_else_if"
152     if ast.children[4].value != "empty":
153         analyze(ast.children[4])
154
155
156 def optElseAnalyzer(ast):
157     # for "statement_block"
158     if ast.children[2].value != "empty":
159         analyze(ast.children[2])
160
161
162 def ifCommandAnalyzer(ast):
163     # for "expression"
164     analyze(ast.children[1])
165     # for "statement_block"
166     analyze(ast.children[3])
167
168     if ast.children[4].value != "empty":
169         analyze(ast.children[4])
170

```

```

171     if ast.children[5].value != "empty":
172         analyze(ast.children[5])
173
174
175 def repeatTimesAnalyzer(ast):
176     # for "plus_expression"
177     if analyze(ast.children[1]) != "number":
178         errorList.append("Error in repeat loop: \
179             need to use valid variable or number")
180     # for "statement_block"
181     analyze(ast.children[4])
182
183
184 def repeatIfAnalyzer(ast):
185     # for "expression"
186     analyze(ast.children[2])
187     # for "statement_block"
188     analyze(ast.children[4])
189
190
191 def declarationCommandAnalyzer(ast):
192     # Note ast.children[3].type is word
193     table.addEntry(SymbolTableEntry(
194         ast.children[0].value,
195         ast.children[3].value,
196         list(scopeList),
197         function,
198         None))
199
200
201 def assignmentCommandAnalyzer(ast):
202     # check for the existence of ID - child 1
203     # and that it can be accessed in this block
204     idNoneBool = False
205     id = ast.children[1].value
206     idEntry = table.getEntry(SymbolTableEntry(
207         id,

```

```

208         None,
209         list(scopeList),
210         function,
211         None))
212     if idEntry is None:
213         idNoneBool = True
214         # ID does not exist or exists but the scoping is wrong
215         errorList.append("Error1 in assignment: \
216             variable does not exist or cannot be used here")
217     else:
218         idEntry.initialized = True
219
220     # do type checking
221     # child 3 is an expression - it needs to be evaluated to a type
222     child3Evaluation = analyze(ast.children[3])
223     if child3Evaluation == "ERROR":
224         # type check in expression failed
225         errorList.append("Error2 in assignment: \
226             use only words or only numbers; cannot mix both")
227     else:
228         if (not idNoneBool) and idEntry.type != child3Evaluation:
229             # type check failed
230             errorList.append("Error3 in assignment: \
231                 variable and value must have the same type")
232
233
234 def printAnalyzer(ast):
235     # for the word or identifier
236     if analyze(ast.children[1]) == "ERROR":
237         errorList.append("Error in an expression: \
238             use only words or only numbers; cannot mix both")
239     # check will be done in analyze
240
241
242 def defineCommandAnalyzer(ast):
243     global function
244     id = ast.children[0].value

```

```

245     if scopeList[-1] != 0:
246         errorList.append("Error in function creation: \
247             functions cannot be created in other \
248             functions or a nested block")
249     function = id
250     if firstPass:
251         paramList = []
252         if ast.children[1].value != "empty":
253             paramList = optParamListAnalyzer(ast.children[1])
254             scopeList.pop()
255             table.addEntry(SymbolTableEntry(
256                 id,
257                 "function",
258                 list(scopeList),
259                 None,
260                 paramList))
261         return
262     # for "statement_block"
263     analyze(ast.children[2])
264     function = None
265
266
267 def optParamListAnalyzer(ast):
268     scopeList.append(count + 1)
269     parameterTypeList = []
270     toAdd = SymbolTableEntry(
271         ast.children[1].value,
272         ast.children[3].value,
273         list(scopeList),
274         function,
275         None)
276     toAdd.functionParamBool = True
277     table.addEntry(toAdd)
278     parameterTypeList.append(ast.children[3].value)
279     if ast.children[5].value == "opt_extra_params":
280         return optExtraParamsAnalyzer(ast.children[5], parameterTypeList)
281     else:

```



```

282         return parameterTypeList
283
284
285 def optExtraParamsAnalyzer(ast, parameterTypeList):
286     toAdd = SymbolTableEntry(
287         ast.children[1].value,
288         ast.children[3].value,
289         list(scopeList),
290         function,
291         None)
292     toAdd.functionParamBool = True
293     table.addEntry(toAdd)
294     parameterTypeList.append(ast.children[3].value)
295     if ast.children[5].value == "opt_extra_params":
296         return optParametersAnalyzer(ast.children[5], parameterTypeList)
297     else:
298         return list(parameterTypeList)
299
300
301 def statementBlockAnalyzer(ast):
302     global count
303     count += 1
304     scopeList.append(count)
305     analyze(ast.children[0])
306     scopeList.pop()
307
308
309 def functionCommandAnalyzer(ast):
310     # check existence of function ID
311     idEntry = table.getEntry(SymbolTableEntry(
312         ast.children[0].value,
313         "function",
314         list(scopeList),
315         function,
316         None))
317     if idEntry is None:
318         errorList.append("Error in attempt to use function: \

```

```

319         function does not exist")
320     return
321 elif idEntry.type == "function":
322     parameterTypeList = list(idEntry.functionParameterTypes)
323 else:
324     errorList.append("Error in attempt to use function: \
325         function does not exist")
326     return
327 if len(ast.children) == 2:
328     optParametersAnalyzer(ast.children[1], parameterTypeList)
329
330
331 # this is for user-defined function parameters
332 def optParametersAnalyzer(ast, parameterTypeList):
333     if len(ast.children) == 1:
334         if (len(parameterTypeList) != 1 or
335             analyze(ast.children[0]) != parameterTypeList[0]):
336             # Type checking error
337             errorList.append("Error in attempt to use function: \
338                 wrong type of parameter used")
339     else:
340         # More parameters left
341         if analyze(ast.children[1]) != parameterTypeList.pop():
342             # Type checking error
343             errorList.append("Error1 in attempt to use function: \
344                 wrong type of parameter used")
345     else:
346         optParametersAnalyzer(ast.children[0], parameterTypeList)
347
348
349 def binaryOperatorAnalyzer(ast):
350     result1 = analyze(ast.children[0])
351     result3 = analyze(ast.children[2])
352
353     if result1 == "ERROR" or result3 == "ERROR":
354         return "ERROR"
355

```

```

356     elif result1 == result3:
357         return result1
358
359     elif ast.children[1].type == "CONCAT":
360         return "word"
361
362     else:
363         return "ERROR"
364
365
366 def plusExpressionAnalyzer(ast):
367     return binaryOperatorAnalyzer(ast)
368
369
370 def timesExpressionAnalyzer(ast):
371     return binaryOperatorAnalyzer(ast)
372
373
374 def wordExpressionAnalyzer(ast):
375     return binaryOperatorAnalyzer(ast)
376
377 if __name__ == "__main__":
378     inputString = ''
379     while True:
380
381         inputString = raw_input('enter expression > ')
382
383         if inputString == 'exit':
384             break
385
386         else:
387             # first parse the string
388             ast = Parser.parseString(inputString)
389
390             ast.printTree()
391             print
392

```

```

393         # then check for errors
394         if len(ast.errors) > 0:
395             print ast.errors
396             break
397
398     analyzeStart(ast)

```

10.5 Translator

Listing 5: Compiler.py

```

1  # Written by Sam Kohn and Jeremy Spencer
2
3  from Parser import parseString
4  from SemanticAnalyzer import analyzeStart
5
6
7  def getPythonCode(code):
8      '''Convert the given Racecar code into the Python code that will
9      run in the GUI.'''
10
11     # first parse the string
12     ast = parseString(code)
13
14     # then run the string through the semantic analyzer
15     semanticErrors = analyzeStart(ast)
16
17     # then check for errors
18     if len(ast.errors) > 0 or len(semanticErrors) > 0:
19         return (None, ast.errors + semanticErrors)
20
21     # then generate python code!
22     pythonCode = generatePythonCode(ast)
23
24     return (pythonCode, None)
25
26
27  def generatePythonCode(ast):
28      '''Traverse the AST and output a string containing the python code

```

```

29     to execute in the GUI.'''
30
31     # potential AST values and their associated translation functions
32     # use astTranslators.get() instead of a long chain of else-ifs
33     astTranslators = {
34         "ID": idTranslator,
35         "assignment_command": assignmentCommandTranslator,
36         "backward": backwardTranslator,
37         "backwards": backwardTranslator,
38         "comparison": comparisonTranslator,
39         "can_drive_expression": canDriveExpressionTranslator,
40         "declaration_command": declarationCommandTranslator,
41         "define_command": defineCommandTranslator,
42         "drive_command": driveCommandTranslator,
43         "empty": emptyTranslator,
44         "forward": forwardTranslator,
45         "forwards": forwardTranslator,
46         "function_command": functionCommandTranslator,
47         "getCarPosition": getCarPositionTranslator,
48         "if_command": ifCommandTranslator,
49         "left": leftTranslator,
50         "opt_else": optElseTranslator,
51         "opt_else_if": optElseIfTranslator,
52         "opt_extra_params": optExtraParamsTranslator,
53         "opt_param_list": optParamListTranslator,
54         "opt_parameters": optParametersTranslator,
55         "plus_expression": plusExpressionTranslator,
56         "print": printTranslator,
57         "repeat_if_command": repeatIfTranslator,
58         "repeat_times_command": repeatTimesTranslator,
59         "right": rightTranslator,
60         "statement_block": statementBlockTranslator,
61         "statements": statementsTranslator,
62         "times_expression": timesExpressionTranslator,
63         "turn_command": turnCommandTranslator,
64         "word_expression": wordExpressionTranslator,
65     }

```

```

66
67     # "declare" pythonCode since otherwise its first use is inside
68     # an if statement
69     pythonCode = ""
70
71     # Fetch the appropriate translator function from astTranslators
72     # If there is no translator for ast.value then just let the
73     # "translator" be ast.value
74     translator = astTranslators.get(ast.value, ast.value)
75
76     # If the "translator" is just a string (inherits from basestring),
77     # then return that translator
78     if isinstance(translator, basestring):
79         pythonCode = ast.value
80
81     # if the translator is a real function then invoke it
82     else:
83         pythonCode = translator(ast)
84
85     return pythonCode
86
87
88 def indentLines(unindentedLines):
89     '''Insert 4 spaces (i.e. 1 tab) at the beginning of every line'''
90
91     splitCode = unindentedLines.splitlines(True)
92
93     pythonCode = "    " + "    ".join(splitCode)
94     return pythonCode
95
96
97 def emptyTranslator(ast):
98     return ""
99
100
101 def statementsTranslator(ast):
102     pythonCode = generatePythonCode(ast.children[0])

```

```

103     pythonCode += generatePythonCode(ast.children[1])
104     return pythonCode
105
106
107 def driveCommandTranslator(ast):
108     # drive numSteps direction steps -->
109     # translate_car(numSteps, direction)\n
110     pythonCode = "translate_car("
111     pythonCode += generatePythonCode(ast.children[1])
112     pythonCode += ", " + generatePythonCode(ast.children[0])
113     pythonCode += ")\n"
114     return pythonCode
115
116
117 def forwardTranslator(ast):
118     pythonCode = "CarDirection.FORWARDS"
119     return pythonCode
120
121
122 def backwardTranslator(ast):
123     pythonCode = "CarDirection.BACKWARDS"
124     return pythonCode
125
126
127 def turnCommandTranslator(ast):
128     pythonCode = "rotate_car("
129     pythonCode += generatePythonCode(ast.children[1])
130     pythonCode += ")\n"
131     return pythonCode
132
133
134 def comparisonTranslator(ast):
135     pythonCode = generatePythonCode(ast.children[0])
136     if ast.children[1].value == "is not":
137         pythonCode += " != "
138     pythonCode += ast.children[2].value
139     elif ast.children[1].value == "is":

```

```

140         pythonCode += " == "
141         pythonCode += generatePythonCode(ast.children[2])
142     else:
143         pythonCode += " " + generatePythonCode(ast.children[1])
144         pythonCode += " " + generatePythonCode(ast.children[2])
145
146     return pythonCode
147
148
149 def optElseIfTranslator(ast):
150     pythonCode = "elif "
151     pythonCode += generatePythonCode(ast.children[1]) + ":\n"
152     pythonCode += generatePythonCode(ast.children[3])
153
154     if ast.children[4].value != "empty":
155         pythonCode += generatePythonCode(ast.children[4])
156
157     return pythonCode
158
159
160 def optElseTranslator(ast):
161     pythonCode = "else:\n"
162     prelimPythonCode = generatePythonCode(ast.children[2])
163     pythonCode += generatePythonCode(ast.children[2])
164     return pythonCode
165
166
167 def ifCommandTranslator(ast):
168     pythonCode = "if " + generatePythonCode(ast.children[1]) + ":\n"
169     pythonCode += generatePythonCode(ast.children[3])
170
171     if ast.children[4].value != "empty":
172         pythonCode += generatePythonCode(ast.children[4])
173
174     if ast.children[5].value != "empty":
175         pythonCode += generatePythonCode(ast.children[5])
176     return pythonCode

```



```

177
178
179 def leftTranslator(ast):
180     pythonCode = "WheelDirection.LEFT"
181     return pythonCode
182
183
184 def rightTranslator(ast):
185     pythonCode = "WheelDirection.RIGHT"
186     return pythonCode
187
188
189 def repeatTimesTranslator(ast):
190     if ast.children[2].value == "times":
191         pythonCode = "for x in range(" + ast.children[1].value + "):\n"
192         pythonCode += generatePythonCode(ast.children[4])
193     return pythonCode
194
195
196 def repeatIfTranslator(ast):
197     pythonCode = "while " + generatePythonCode(ast.children[2]) + ":\n"
198     pythonCode += generatePythonCode(ast.children[4])
199     return pythonCode
200
201
202 def declarationCommandTranslator(ast):
203     # id is a whatever -->
204     # id = None
205     pythonCode = generatePythonCode(ast.children[0])
206     pythonCode += " = None\n"
207     return pythonCode
208
209
210 def idTranslator(ast):
211     pythonCode = ast.value
212     return pythonCode
213

```

```

214
215 def assignmentCommandTranslator(ast):
216     pythonCode = generatePythonCode(ast.children[1])
217     pythonCode += " = "
218     pythonCode += generatePythonCode(ast.children[3])
219     pythonCode += "\n"
220     return pythonCode
221
222
223 def printTranslator(ast):
224     pythonCode = "print_to_console("
225     pythonCode += generatePythonCode(ast.children[1])
226     pythonCode += ")\n"
227     return pythonCode
228
229
230 def defineCommandTranslator(ast):
231     pythonCode = "def "
232     pythonCode += generatePythonCode(ast.children[0])
233     pythonCode += "("
234     if ast.children[1].value == "opt_param_list":
235         pythonCode += generatePythonCode(ast.children[1])
236     pythonCode += "):\n"
237     pythonCode += generatePythonCode(ast.children[2])
238     return pythonCode
239
240
241 def optParamListTranslator(ast):
242     pythonCode = generatePythonCode(ast.children[1])
243     if ast.children[5].value == "opt_extra_params":
244         pythonCode += generatePythonCode(ast.children[5])
245     return pythonCode
246
247
248 def optExtraParamsTranslator(ast):
249     pythonCode = ", "
250     pythonCode += generatePythonCode(ast.children[1])

```

```

251     if ast.children[5].value == "opt_extra_params":
252         pythonCode += generatePythonCode(ast.children[5])
253     return pythonCode
254
255
256 def statementBlockTranslator(ast):
257     prelimPythonCode = generatePythonCode(ast.children[0])
258
259     pythonCode = indentLines(prelimPythonCode)
260
261     return pythonCode
262
263
264 def functionCommandTranslator(ast):
265     pythonCode = generatePythonCode(ast.children[0])
266     pythonCode += "("
267     if len(ast.children) > 1:
268         pythonCode += generatePythonCode(ast.children[1])
269     pythonCode += ")\n"
270     return pythonCode
271
272
273 def optParametersTranslator(ast):
274     numChildren = len(ast.children)
275     if numChildren > 0:
276         pythonCode = generatePythonCode(ast.children[0])
277         if numChildren == 2:
278             pythonCode += ", "
279             pythonCode += generatePythonCode(ast.children[1])
280         return pythonCode
281     else:
282         return ""
283
284
285 def binaryOperatorTranslator(ast):
286     pythonCode = "("
287     pythonCode += generatePythonCode(ast.children[0])

```

```

288     pythonCode += ") "
289     pythonCode += generatePythonCode(ast.children[1])
290     pythonCode += " ("
291     pythonCode += generatePythonCode(ast.children[2])
292     pythonCode += ")) "
293     return pythonCode
294
295
296 def plusExpressionTranslator(ast):
297     return binaryOperatorTranslator(ast)
298
299
300 def timesExpressionTranslator(ast):
301     return binaryOperatorTranslator(ast)
302
303
304 def wordExpressionTranslator(ast):
305     pythonCode = "(str("
306     pythonCode += generatePythonCode(ast.children[0])
307     pythonCode += ") + str("
308     pythonCode += generatePythonCode(ast.children[2])
309     pythonCode += ")) "
310     return pythonCode
311
312
313 def getCarPositionTranslator(ast):
314     return "getCurrentPosition()"
315
316
317 def canDriveExpressionTranslator(ast):
318     pythonCode = "can_move("
319     pythonCode += generatePythonCode(ast.children[1])
320     pythonCode += ", " + generatePythonCode(ast.children[0])
321     pythonCode += ") "
322     return pythonCode
323
324 if __name__ == "__main__":

```

```

325     inputString = ''
326     while True:
327
328         inputString = raw_input('enter expression > ')
329
330         if inputString == 'exit':
331             break
332
333         else:
334             print getPythonCode(inputString)

```

10.6 Tests

Listing 6: Tests.py

```

1  # Written by Colfax Selby, Sam Kohn, and Alex Fields
2  import random
3  import unittest
4  import Racecar.Compiler as Compiler
5  import Racecar.SymbolTable as SymbolTable
6  import Racecar.SemanticAnalyzer as SemanticAnalyzer
7  import Racecar.Parser as Parser
8
9
10 class TranslatorTests(unittest.TestCase):
11     def test_empty_statement(self):
12         test_string = \
13             """
14             """
15         correct_translation = \
16             """
17         result = Compiler.getPythonCode(test_string)
18         ast = Parser.parseString(test_string)
19
20         saErrors = SemanticAnalyzer.analyzeStart(ast)
21         self.assertEqual(len(saErrors), 0)
22
23         self.assertEqual(len(ast.errors), 0)
24         self.assertEqual(result[0], correct_translation)

```

```

25
26     def test_drive_forwards(self):
27         test_string1 = \
28             """drive forwards 10 steps
29         """
30         test_string2 = \
31             """drive forward 10 steps
32         """
33         test_string3 = \
34             """drive forwards 10 step
35         """
36         test_string4 = \
37             """drive forward 10 step
38         """
39         correct_translation = \
40             """translate_car(10, CarDirection.FORWARDS)
41         """
42
43         ast1 = Parser.parseString(test_string1)
44         ast2 = Parser.parseString(test_string1)
45         ast3 = Parser.parseString(test_string1)
46         ast4 = Parser.parseString(test_string1)
47
48         self.assertEqual(len(ast1.errors), 0)
49         self.assertEqual(len(ast2.errors), 0)
50         self.assertEqual(len(ast3.errors), 0)
51         self.assertEqual(len(ast4.errors), 0)
52
53         result1 = Compiler.getPythonCode(test_string1)
54         result2 = Compiler.getPythonCode(test_string2)
55         result3 = Compiler.getPythonCode(test_string3)
56         result4 = Compiler.getPythonCode(test_string4)
57
58         saErrors1 = SemanticAnalyzer.analyzeStart(ast1)
59         saErrors2 = SemanticAnalyzer.analyzeStart(ast2)
60         saErrors3 = SemanticAnalyzer.analyzeStart(ast3)
61         saErrors4 = SemanticAnalyzer.analyzeStart(ast4)

```

```

62
63     self.assertEqual(len(saErrors1), 0)
64     self.assertEqual(len(saErrors2), 0)
65     self.assertEqual(len(saErrors3), 0)
66     self.assertEqual(len(saErrors4), 0)
67
68     self.assertEqual(result1[0], correct_translation)
69     self.assertEqual(result2[0], correct_translation)
70     self.assertEqual(result3[0], correct_translation)
71     self.assertEqual(result4[0], correct_translation)
72
73     def test_drive_backwards(self):
74         test_string1 = \
75             """drive backwards 10 steps
76 """
77         test_string2 = \
78             """drive backward 10 steps
79 """
80         test_string3 = \
81             """drive backwards 10 step
82 """
83         test_string4 = \
84             """drive backward 10 step
85 """
86         correct_translation = \
87             """translate_car(10, CarDirection.BACKWARDS)
88 """
89
90         ast1 = Parser.parseString(test_string1)
91         ast2 = Parser.parseString(test_string1)
92         ast3 = Parser.parseString(test_string1)
93         ast4 = Parser.parseString(test_string1)
94
95         self.assertEqual(len(ast1.errors), 0)
96         self.assertEqual(len(ast2.errors), 0)
97         self.assertEqual(len(ast3.errors), 0)
98         self.assertEqual(len(ast4.errors), 0)

```

```

99
100     result1 = Compiler.getPythonCode(test_string1)
101     result2 = Compiler.getPythonCode(test_string2)
102     result3 = Compiler.getPythonCode(test_string3)
103     result4 = Compiler.getPythonCode(test_string4)
104
105     saErrors1 = SemanticAnalyzer.analyzeStart(ast1)
106     saErrors2 = SemanticAnalyzer.analyzeStart(ast2)
107     saErrors3 = SemanticAnalyzer.analyzeStart(ast3)
108     saErrors4 = SemanticAnalyzer.analyzeStart(ast4)
109
110     self.assertEqual(len(saErrors1), 0)
111     self.assertEqual(len(saErrors2), 0)
112     self.assertEqual(len(saErrors3), 0)
113     self.assertEqual(len(saErrors4), 0)
114
115     self.assertEqual(result1[0], correct_translation)
116     self.assertEqual(result2[0], correct_translation)
117     self.assertEqual(result3[0], correct_translation)
118     self.assertEqual(result4[0], correct_translation)
119
120     def test_turn_left(self):
121         test_string = \
122             """turn left
123         """
124         correct_translation = \
125             """rotate_car(WheelDirection.LEFT)
126         """
127         result = Compiler.getPythonCode(test_string)
128         ast = Parser.parseString(test_string)
129
130         saErrors = SemanticAnalyzer.analyzeStart(ast)
131         self.assertEqual(len(saErrors), 0)
132
133         self.assertEqual(len(ast.errors), 0)
134         self.assertEqual(result[0], correct_translation)
135

```



```

136     def test_turn_right(self):
137         test_string = \
138             """turn right
139         """
140         correct_translation = \
141             """rotate_car(WheelDirection.RIGHT)
142         """
143         result = Compiler.getPythonCode(test_string)
144         ast = Parser.parseString(test_string)
145
146         saErrors = SemanticAnalyzer.analyzeStart(ast)
147         self.assertEqual(len(saErrors), 0)
148
149         self.assertEqual(len(ast.errors), 0)
150         self.assertEqual(result[0], correct_translation)
151
152     def test_print(self):
153         test_string = \
154             """print "hello world"
155         """
156         correct_translation = \
157             """print_to_console("hello world")
158         """
159         result = Compiler.getPythonCode(test_string)
160         ast = Parser.parseString(test_string)
161
162         saErrors = SemanticAnalyzer.analyzeStart(ast)
163         self.assertEqual(len(saErrors), 0)
164
165         self.assertEqual(len(ast.errors), 0)
166         self.assertEqual(result[0], correct_translation)
167
168     def test_declare(self):
169         test_string = \
170             """myNum is a number
171         """
172         correct_translation = \

```

```

173         """myNum = None
174     """
175     result = Compiler.getPythonCode(test_string)
176     ast = Parser.parseString(test_string)
177
178     saErrors = SemanticAnalyzer.analyzeStart(ast)
179     self.assertEqual(len(saErrors), 0)
180
181     self.assertEqual(len(ast.errors), 0)
182     self.assertEqual(result[0], correct_translation)
183
184     def test_assign(self):
185         test_string = \
186             """myVar is a number
187 otherThing is a number
188 set myVar to 10
189 set otherThing to 11
190 set myVar to otherThing
191 """
192         correct_translation = \
193             """myVar = None
194 otherThing = None
195 myVar = 10
196 otherThing = 11
197 myVar = otherThing
198 """
199         result = Compiler.getPythonCode(test_string)
200         ast = Parser.parseString(test_string)
201
202         saErrors = SemanticAnalyzer.analyzeStart(ast)
203         self.assertEqual(len(saErrors), 0)
204
205         self.assertEqual(len(ast.errors), 0)
206         self.assertEqual(result[0], correct_translation)
207
208     def test_define(self):
209         test_string = \

```

```

210         """define moveForwardFive
211     {
212         drive forward 5 steps
213     }
214     """
215     correct_translation = \
216         """def moveForwardFive():
217     translate_car(5, CarDirection.FORWARDS)
218     """
219
220     result = Compiler.getPythonCode(test_string)
221     ast = Parser.parseString(test_string)
222
223     saErrors = SemanticAnalyzer.analyzeStart(ast)
224     self.assertEqual(len(saErrors), 0)
225
226     self.assertEqual(len(ast.errors), 0)
227     self.assertEqual(result[0], correct_translation)
228
229     def test_function_invocation_no_params(self):
230         test_string = \
231             """define moveBackwardFive
232     {
233         drive backward 5
234     }
235     define moveForwardThenBackward
236     {
237         drive forward 5
238         moveBackwardFive
239     }
240     moveForwardThenBackward
241     """
242     correct_translation = \
243         """def moveBackwardFive():
244     translate_car(5, CarDirection.BACKWARDS)
245     def moveForwardThenBackward():
246     translate_car(5, CarDirection.FORWARDS)

```

```

247     moveBackwardFive()
248 moveForwardThenBackward()
249 """
250     result = Compiler.getPythonCode(test_string)
251     ast = Parser.parseString(test_string)
252
253     saErrors = SemanticAnalyzer.analyzeStart(ast)
254     self.assertEqual(len(saErrors), 0)
255
256     self.assertEqual(len(ast.errors), 0)
257     self.assertEqual(result[0], correct_translation)
258
259     def test_function_invocation_with_one_parameter(self):
260         test_string = \
261             """define move5Steps using direction (word)
262 {
263     print direction
264 }
265 move5Steps "forwards"
266 """
267         correct_translation = \
268             """def move5Steps(direction):
269     print_to_console(direction)
270 move5Steps("forwards")
271 """
272         result = Compiler.getPythonCode(test_string)
273         ast = Parser.parseString(test_string)
274
275         saErrors = SemanticAnalyzer.analyzeStart(ast)
276         self.assertEqual(len(saErrors), 0)
277
278         self.assertEqual(len(ast.errors), 0)
279         self.assertEqual(result[0], correct_translation)
280
281     def test_function_invocation_with_two_parameters(self):
282         test_string = \
283             """define turnLeftThenDriveStraight using numStepsTurn \

```

```

284         (number) and numStepsDrive (number)
285     {
286     turn left
287     drive forward numStepsTurn steps
288     turn right
289     drive forward numStepsDrive steps
290     }
291     turnLeftThenDriveStraight 5 10
292     """
293     correct_translation = \
294         """def turnLeftThenDriveStraight(numStepsTurn, numStepsDrive):
295         rotate_car(WheelDirection.LEFT)
296         translate_car(numStepsTurn, CarDirection.FORWARDS)
297         rotate_car(WheelDirection.RIGHT)
298         translate_car(numStepsDrive, CarDirection.FORWARDS)
299     turnLeftThenDriveStraight(5, 10)
300     """
301     result = Compiler.getPythonCode(test_string)
302     ast = Parser.parseString(test_string)
303
304     saErrors = SemanticAnalyzer.analyzeStart(ast)
305     self.assertEqual(len(saErrors), 0)
306
307     self.assertEqual(len(ast.errors), 0)
308     self.assertEqual(result[0], correct_translation)
309
310     def test_plus_expression(self):
311         test_string = \
312             """print (2 + 3)
313         """
314         correct_translation = \
315             """print_to_console(((2) + (3)))
316         """
317         result = Compiler.getPythonCode(test_string)
318         ast = Parser.parseString(test_string)
319
320         saErrors = SemanticAnalyzer.analyzeStart(ast)

```

```

321         self.assertEqual(len(saErrors), 0)
322
323         self.assertEqual(len(ast.errors), 0)
324         self.assertEqual(result[0], correct_translation)
325
326     def test_times_expression(self):
327         test_string = \
328             """print (2 * 3)
329         """
330         correct_translation = \
331             """print_to_console(((2) * (3)))
332         """
333         result = Compiler.getPythonCode(test_string)
334         ast = Parser.parseString(test_string)
335
336         saErrors = SemanticAnalyzer.analyzeStart(ast)
337         self.assertEqual(len(saErrors), 0)
338
339         self.assertEqual(len(ast.errors), 0)
340         self.assertEqual(result[0], correct_translation)
341
342     def test_minus_expression(self):
343         test_string = \
344             """print (2 - 3)
345         """
346         correct_translation = \
347             """print_to_console(((2) - (3)))
348         """
349         result = Compiler.getPythonCode(test_string)
350         ast = Parser.parseString(test_string)
351
352         saErrors = SemanticAnalyzer.analyzeStart(ast)
353         self.assertEqual(len(saErrors), 0)
354
355         self.assertEqual(len(ast.errors), 0)
356         self.assertEqual(result[0], correct_translation)
357

```

```

358     def test_divide_expression(self):
359         test_string = \
360             """print (2 / 3)
361         """
362         correct_translation = \
363             """print_to_console(((2) / (3)))
364         """
365         result = Compiler.getPythonCode(test_string)
366         ast = Parser.parseString(test_string)
367
368         saErrors = SemanticAnalyzer.analyzeStart(ast)
369         self.assertEqual(len(saErrors), 0)
370
371         self.assertEqual(len(ast.errors), 0)
372         self.assertEqual(result[0], correct_translation)
373
374     def test_all_expression(self):
375         test_string = \
376             """print (1 + 2 * (3 + 4))
377         """
378         correct_translation = \
379             """print_to_console(((1) + ((2) * ((3) + (4))))))
380         """
381         result = Compiler.getPythonCode(test_string)
382         ast = Parser.parseString(test_string)
383
384         saErrors = SemanticAnalyzer.analyzeStart(ast)
385         self.assertEqual(len(saErrors), 0)
386
387         self.assertEqual(len(ast.errors), 0)
388         self.assertEqual(result[0], correct_translation)
389
390     def test_assign_num_change(self):
391         test_string = \
392             """num is a number
393 set num to 10
394 set num to num*2

```

```

395 """
396         correct_translation = \
397             """num = None
398 num = 10
399 num = ((num) * (2))
400 """
401         result = Compiler.getPythonCode(test_string)
402         ast = Parser.parseString(test_string)
403
404         saErrors = SemanticAnalyzer.analyzeStart(ast)
405         self.assertEqual(len(saErrors), 0)
406
407         self.assertEqual(len(ast.errors), 0)
408         self.assertEqual(result[0], correct_translation)
409
410     def test_assign_easy_num_change(self):
411         test_string = \
412             """num is a number
413 set num to 10
414 num2 is a number
415 set num2 to 20
416 set num to num2
417 """
418         correct_translation = \
419             """num = None
420 num = 10
421 num2 = None
422 num2 = 20
423 num = num2
424 """
425         result = Compiler.getPythonCode(test_string)
426         ast = Parser.parseString(test_string)
427
428         saErrors = SemanticAnalyzer.analyzeStart(ast)
429         self.assertEqual(len(saErrors), 0)
430
431         self.assertEqual(len(ast.errors), 0)

```



```

432         self.assertEqual(result[0], correct_translation)
433
434     def test_assign_word_print(self):
435         test_string = \
436             """color is a word
437 set color to "blue"
438 print color
439 """
440         correct_translation = \
441             """color = None
442 color = "blue"
443 print_to_console(color)
444 """
445         result = Compiler.getPythonCode(test_string)
446         ast = Parser.parseString(test_string)
447
448         saErrors = SemanticAnalyzer.analyzeStart(ast)
449         self.assertEqual(len(saErrors), 0)
450
451         self.assertEqual(len(ast.errors), 0)
452         self.assertEqual(result[0], correct_translation)
453
454     def test_assign_word_print_complicated(self):
455         test_string = \
456             """color is a word
457 set color to "blue"
458 print color
459 c2 is a word
460 set c2 to "green"
461 set color to c2
462 """
463         correct_translation = \
464             """color = None
465 color = "blue"
466 print_to_console(color)
467 c2 = None
468 c2 = "green"

```

```

469 color = c2
470 """
471     result = Compiler.getPythonCode(test_string)
472     ast = Parser.parseString(test_string)
473
474     saErrors = SemanticAnalyzer.analyzeStart(ast)
475     self.assertEqual(len(saErrors), 0)
476
477     self.assertEqual(len(ast.errors), 0)
478     self.assertEqual(result[0], correct_translation)
479
480     def test_if_statement(self):
481         test_string = \
482             """if 1
483 {
484     print "yay"
485 }
486 """
487         correct_translation = \
488             """if 1:
489     print_to_console("yay")
490 """
491         result = Compiler.getPythonCode(test_string)
492         ast = Parser.parseString(test_string)
493
494         saErrors = SemanticAnalyzer.analyzeStart(ast)
495         self.assertEqual(len(saErrors), 0)
496
497         self.assertEqual(len(ast.errors), 0)
498         self.assertEqual(result[0], correct_translation)
499
500     def test_if_else_statement(self):
501         test_string = \
502             """if 1
503 {
504     print "yay"
505 }

```

```

506 else
507 {
508     print "no"
509 }
510 """
511         correct_translation = \
512             """if 1:
513     print_to_console("yay")
514 else:
515     print_to_console("no")
516 """
517         result = Compiler.getPythonCode(test_string)
518         ast = Parser.parseString(test_string)
519
520         saErrors = SemanticAnalyzer.analyzeStart(ast)
521         self.assertEqual(len(saErrors), 0)
522
523         self.assertEqual(len(ast.errors), 0)
524         self.assertEqual(result[0], correct_translation)
525
526     def test_if_statement_nested(self):
527         test_string = \
528             """if 1
529 {
530     print "yay"
531     if 1
532     {
533         print "yahoo"
534     }
535 }
536 """
537         correct_translation = \
538             """if 1:
539     print_to_console("yay")
540     if 1:
541         print_to_console("yahoo")
542 """

```

```

543         result = Compiler.getPythonCode(test_string)
544         ast = Parser.parseString(test_string)
545
546         saErrors = SemanticAnalyzer.analyzeStart(ast)
547         self.assertEqual(len(saErrors), 0)
548
549         self.assertEqual(len(ast.errors), 0)
550         self.assertEqual(result[0], correct_translation)
551
552     def test_if_else_statement_nested(self):
553         test_string = \
554             """if 1
555 {
556     print "yay"
557     if 1
558     {
559         print "yahoo"
560     }
561     else
562     {
563         print "oh no"
564     }
565 }
566 else
567 {
568     print "no"
569 }
570 """
571         correct_translation = \
572             """if 1:
573     print_to_console("yay")
574     if 1:
575         print_to_console("yahoo")
576     else:
577         print_to_console("oh no")
578 else:
579     print_to_console("no")

```

```

580 """
581         result = Compiler.getPythonCode(test_string)
582         ast = Parser.parseString(test_string)
583
584         saErrors = SemanticAnalyzer.analyzeStart(ast)
585         self.assertEqual(len(saErrors), 0)
586
587         self.assertEqual(len(ast.errors), 0)
588         self.assertEqual(result[0], correct_translation)
589
590     def test_if_elseif_else_statement(self):
591         test_string = \
592             """if 1
593 {
594     print "yay"
595 }
596 elif 2
597 {
598     print "no"
599 }
600 else
601 {
602     print "done"
603 }
604 """
605         correct_translation = \
606             """if 1:
607     print_to_console("yay")
608 elif 2:
609     print_to_console("no")
610 else:
611     print_to_console("done")
612 """
613         result = Compiler.getPythonCode(test_string)
614         ast = Parser.parseString(test_string)
615
616         saErrors = SemanticAnalyzer.analyzeStart(ast)

```

```

617         self.assertEqual(len(saErrors), 0)
618
619         self.assertEqual(len(ast.errors), 0)
620         self.assertEqual(result[0], correct_translation)
621
622     def test_if_elseif_else_statement_nested(self):
623         test_string = \
624             """if 1
625 {
626     print "yay"
627     if 1
628     {
629         print "yahoo"
630     }
631     elif 2
632     {
633         print "oh no"
634     }
635     else
636     {
637         print "here"
638     }
639 }
640 elif 2
641 {
642     print "no"
643 }
644 else
645 {
646     print "done"
647 }
648 """
649         correct_translation = \
650             """if 1:
651     print_to_console("yay")
652     if 1:
653         print_to_console("yahoo")

```

```

654     elif 2:
655         print_to_console("oh no")
656     else:
657         print_to_console("here")
658 elif 2:
659     print_to_console("no")
660 else:
661     print_to_console("done")
662 """
663     result = Compiler.getPythonCode(test_string)
664     ast = Parser.parseString(test_string)
665
666     saErrors = SemanticAnalyzer.analyzeStart(ast)
667     self.assertEqual(len(saErrors), 0)
668
669     self.assertEqual(len(ast.errors), 0)
670     self.assertEqual(result[0], correct_translation)
671
672     def test_if_if_else_complicated(self):
673         test_string = \
674             """if 1
675 {
676     print "yay"
677     if 1
678     {
679         print "yahoo"
680     }
681     else
682     {
683         print "oh no"
684         if 1
685         {
686             print "good"
687         }
688         if 1
689         {
690             print "yay"

```

```

691         }
692     elif 2
693     {
694         print "no"
695         if 1
696         {
697             print "hi"
698         }
699     }
700     elif 3
701     {
702         print "yes"
703     }
704     else
705     {
706         if 5
707         {
708             print "works"
709         }
710         print "end"
711     }
712 }
713 }
714 else
715 {
716     print "no"
717 }
718 """
719     correct_translation = \
720         """if 1:
721     print_to_console("yay")
722     if 1:
723         print_to_console("yahoo")
724     else:
725         print_to_console("oh no")
726         if 1:
727             print_to_console("good")

```



```

728         if 1:
729             print_to_console("yay")
730         elif 2:
731             print_to_console("no")
732             if 1:
733                 print_to_console("hi")
734         elif 3:
735             print_to_console("yes")
736         else:
737             if 5:
738                 print_to_console("works")
739                 print_to_console("end")
740     else:
741         print_to_console("no")
742     """
743     result = Compiler.getPythonCode(test_string)
744     ast = Parser.parseString(test_string)
745
746     saErrors = SemanticAnalyzer.analyzeStart(ast)
747     self.assertEqual(len(saErrors), 0)
748
749     self.assertEqual(len(ast.errors), 0)
750     self.assertEqual(result[0], correct_translation)
751
752     def test_comment_singleline(self):
753         test_string = \
754             """:) this is a single line comment
755     """
756         correct_translation = \
757             """
758         result = Compiler.getPythonCode(test_string)
759         ast = Parser.parseString(test_string)
760
761         saErrors = SemanticAnalyzer.analyzeStart(ast)
762         self.assertEqual(len(saErrors), 0)
763
764         self.assertEqual(len(ast.errors), 0)

```

```

765         self.assertEqual(result[0], correct_translation)
766
767     def test_comment_multiline(self):
768         test_string = \
769             """:-( this is
770 a multiline
771 comment
772 :-)
773 """
774         correct_translation = \
775             """
776
777 result = Compiler.getPythonCode(test_string)
778 ast = Parser.parseString(test_string)
779
780 saErrors = SemanticAnalyzer.analyzeStart(ast)
781 self.assertEqual(len(saErrors), 0)
782
783 self.assertEqual(len(ast.errors), 0)
784 self.assertEqual(result[0], correct_translation)
785
786     def test_loop_for(self):
787         test_string = \
788             """myCounter is a number
789 set myCounter to 10
790 repeat myCounter times
791 {
792     drive forward 1 step
793     print myCounter
794 }
795 """
796         correct_translation = \
797             """myCounter = None
798 myCounter = 10
799 for x in range(myCounter):
800     translate_car(1, CarDirection.FORWARDS)
801     print_to_console(myCounter)
802 """

```

```

802         result = Compiler.getPythonCode(test_string)
803         ast = Parser.parseString(test_string)
804
805         saErrors = SemanticAnalyzer.analyzeStart(ast)
806         self.assertEqual(len(saErrors), 0)
807
808         self.assertEqual(len(ast.errors), 0)
809         self.assertEqual(result[0], correct_translation)
810
811     def test_loop_for_nested(self):
812         test_string = \
813             """myCounter is a number
814 set myCounter to 10
815 myCounter2 is a number
816 set myCounter2 to 10
817 repeat myCounter times
818 {
819     drive forward 1 step
820     repeat myCounter2 times
821     {
822         drive forward 1 step
823     }
824 }
825 """
826         correct_translation = \
827             """myCounter = None
828 myCounter = 10
829 myCounter2 = None
830 myCounter2 = 10
831 for x in range(myCounter):
832     translate_car(1, CarDirection.FORWARDS)
833     for x in range(myCounter2):
834         translate_car(1, CarDirection.FORWARDS)
835 """
836         result = Compiler.getPythonCode(test_string)
837         ast = Parser.parseString(test_string)
838

```

```

839         saErrors = SemanticAnalyzer.analyzeStart(ast)
840         self.assertEqual(len(saErrors), 0)
841
842         self.assertEqual(len(ast.errors), 0)
843         self.assertEqual(result[0], correct_translation)
844
845     def test_loop_while(self):
846         test_string = \
847             """myCounter is a number
848 set myCounter to 1
849 repeat if myCounter is not 5
850 {
851     drive forward 1 step
852     set myCounter to myCounter + 1
853 }
854 """
855         correct_translation = \
856             """myCounter = None
857 myCounter = 1
858 while myCounter != 5:
859     translate_car(1, CarDirection.FORWARDS)
860     myCounter = ((myCounter) + (1))
861 """
862         result = Compiler.getPythonCode(test_string)
863         ast = Parser.parseString(test_string)
864
865         saErrors = SemanticAnalyzer.analyzeStart(ast)
866         self.assertEqual(len(saErrors), 0)
867
868         self.assertEqual(len(ast.errors), 0)
869         self.assertEqual(result[0], correct_translation)
870
871     def test_loop_while_nested(self):
872         test_string = \
873             """myCounter is a number
874 set myCounter to 1
875 myCounter2 is a number

```

```

876 repeat if myCounter is not 5
877 {
878     drive forward 1 step
879     set myCounter2 to 0
880     repeat if myCounter2 is not 5
881     {
882         drive forward 1 step
883         set myCounter2 to myCounter2 + 1
884     }
885     set myCounter to myCounter + 1
886 }
887 """
888     correct_translation = \
889         """myCounter = None
890 myCounter = 1
891 myCounter2 = None
892 while myCounter != 5:
893     translate_car(1, CarDirection.FORWARDS)
894     myCounter2 = 0
895     while myCounter2 != 5:
896         translate_car(1, CarDirection.FORWARDS)
897         myCounter2 = ((myCounter2) + (1))
898     myCounter = ((myCounter) + (1))
899 """
900     result = Compiler.getPythonCode(test_string)
901     ast = Parser.parseString(test_string)
902
903     saErrors = SemanticAnalyzer.analyzeStart(ast)
904     self.assertEqual(len(saErrors), 0)
905
906     self.assertEqual(len(ast.errors), 0)
907     self.assertEqual(result[0], correct_translation)
908
909 def test_boolean_operators(self):
910     test_string = \
911         """if 1 < 2
912 {

```

```

913     print "yes"
914 }
915 elif 1 is 2
916 {
917     print "yes"
918 }
919 if 1 >= 2
920 {
921     print "no"
922 }
923 elif 1 is not 2
924 {
925     print "yes"
926 }
927 elif 1 > 2
928 {
929     print "yes"
930 }
931 elif 1 < 2
932 {
933     print "yes"
934 }
935 """
936     correct_translation = \
937         """if 1 < 2:
938     print_to_console("yes")
939 elif 1 == 2:
940     print_to_console("yes")
941 if 1 >= 2:
942     print_to_console("no")
943 elif 1 != 2:
944     print_to_console("yes")
945 elif 1 > 2:
946     print_to_console("yes")
947 elif 1 < 2:
948     print_to_console("yes")
949 """

```

```

950         result = Compiler.getPythonCode(test_string)
951         ast = Parser.parseString(test_string)
952
953         self.assertEqual(len(ast.errors), 0)
954         self.assertEqual(result[0], correct_translation)
955
956     def test_string_concatenation(self):
957         test_string = \
958             """myWord is a word
959 set myWord to "hi "
960 print "hey" ++ myWord
961 """
962         correct_translation = \
963             """myWord = None
964 myWord = "hi "
965 print_to_console((str("hey") + str(myWord)))
966 """
967         result = Compiler.getPythonCode(test_string)
968         ast = Parser.parseString(test_string)
969
970         self.assertEqual(len(ast.errors), 0)
971         self.assertEqual(result[0], correct_translation)
972
973     def test_string_concatenation_complicated(self):
974         test_string = \
975             """myWord is a word
976 set myWord to "hi "
977 print "hey" ++ myWord ++ "now"
978 """
979         correct_translation = \
980             """myWord = None
981 myWord = "hi "
982 print_to_console((str((str("hey") + str(myWord))) + str("now")))
983 """
984         result = Compiler.getPythonCode(test_string)
985         ast = Parser.parseString(test_string)
986

```

```

987         self.assertEqual(len(ast.errors), 0)
988         self.assertEqual(result[0], correct_translation)
989
990     def test_get_car_position(self):
991         test_string = \
992             """print getCarPosition
993 """
994         correct_translation = \
995             """print_to_console(getCurrentPosition())
996 """
997         result = Compiler.getPythonCode(test_string)
998         ast = Parser.parseString(test_string)
999
1000        self.assertEqual(len(ast.errors), 0)
1001        self.assertEqual(result[0], correct_translation)
1002
1003    def test_can_move(self):
1004        test_string = \
1005            """if canDrive forward 5 steps
1006 {
1007     drive forward 5 steps
1008 }
1009 """
1010        correct_translation = \
1011            """if can_move(5, CarDirection.FORWARDS):
1012    translate_car(5, CarDirection.FORWARDS)
1013 """
1014        result = Compiler.getPythonCode(test_string)
1015        ast = Parser.parseString(test_string)
1016
1017        self.assertEqual(len(ast.errors), 0)
1018        self.assertEqual(result[0], correct_translation)
1019
1020    def test_if_plus(self):
1021        test_string = \
1022            """if 2 + 5
1023 {

```



```

1024     print "yay"
1025 }
1026 """
1027     correct_translation = \
1028         """if ((2) + (5)):
1029     print_to_console("yay")
1030 """
1031     result = Compiler.getPythonCode(test_string)
1032     ast = Parser.parseString(test_string)
1033
1034     self.assertEqual(len(ast.errors), 0)
1035     self.assertEqual(result[0], correct_translation)
1036
1037     def test_template(self):
1038         test_string = \
1039             """
1040 """
1041         correct_translation = \
1042             """
1043         result = Compiler.getPythonCode(test_string)
1044         ast = Parser.parseString(test_string)
1045
1046         self.assertEqual(len(ast.errors), 0)
1047         self.assertEqual(result[0], correct_translation)
1048
1049
1050 class SymbolTableTests(unittest.TestCase):
1051
1052     def test_symbol_table_add_entry(self):
1053         '''Tests the SymbolTableEntry.addEntry() function.'''
1054
1055         table = SymbolTable.SymbolLookupTable()
1056
1057         entry1 = SymbolTable.SymbolTableEntry("name1", "word", [0], None, [])
1058         entry2 = SymbolTable.SymbolTableEntry("name1", "word", [0], None, [])
1059
1060         table.addEntry(entry1)

```

```

1061
1062         self.assertFalse(table.addEntry(entry2))
1063
1064         entry3 = SymbolTable.SymbolTableEntry("name2", "word", [0], None, [])
1065
1066         self.assertTrue(entry3.validateWithTableEntry(entry3))
1067
1068     def test_symbol_table_verify(self):
1069         '''Tests the SymbolLookupTable.verifyEntry() function.'''
1070         table = SymbolTable.SymbolLookupTable()
1071
1072         entry1 = SymbolTable.SymbolTableEntry("name1", "word", [0], None, [])
1073         entry2 = SymbolTable.SymbolTableEntry("name1", "word", [0], None, [])
1074         entry3 = SymbolTable.SymbolTableEntry("name2", "word", [0], None, [])
1075
1076         table.addEntry(entry1)
1077         table.addEntry(entry3)
1078
1079         self.assertTrue(table.verifyEntry(entry2))
1080
1081     def test_symbol_table_get_entry(self):
1082         '''Tests the SymbolLookupTable.getEntry() function.'''
1083         table = SymbolTable.SymbolLookupTable()
1084
1085         entry1 = SymbolTable.SymbolTableEntry("name1", "word", [0], None, [])
1086
1087         table.addEntry(entry1)
1088
1089         self.assertEqual(
1090             table.getEntry(
1091                 SymbolTable.SymbolTableEntry(
1092                     "name1", None, [0], None, [])), entry1)
1093
1094
1095     class SemanticAnalyzerTests(unittest.TestCase):
1096         def test_basic(self):
1097             test_string = \

```

```

1098         """drive forward 5 steps
1099     """
1100
1101     ast = Parser.parseString(test_string)
1102     self.assertEqual(len(ast.errors), 0, "Test failed at parser.")
1103
1104     saErrors = SemanticAnalyzer.analyzeStart(ast)
1105     self.assertEqual(len(saErrors), 0)
1106
1107     def test_print_var(self):
1108         test_string = \
1109             """myNum is a number
1110 set myNum to 10
1111 print myNum
1112 """
1113
1114         ast = Parser.parseString(test_string)
1115         self.assertEqual(len(ast.errors), 0, "Test failed at parser.")
1116
1117         saErrors = SemanticAnalyzer.analyzeStart(ast)
1118         self.assertEqual(len(saErrors), 0)
1119
1120     def test_print_undeclared_var(self):
1121         test_string = \
1122             """print myNum
1123 """
1124
1125         ast = Parser.parseString(test_string)
1126         self.assertEqual(len(ast.errors), 0, "Test failed at parser.")
1127
1128         saErrors = SemanticAnalyzer.analyzeStart(ast)
1129         self.assertEqual(len(saErrors), 1)
1130
1131     def test_set_undeclared_var(self):
1132         test_string = \
1133             """set myNum to 10
1134 """

```

```

1135
1136     ast = Parser.parseString(test_string)
1137     self.assertEqual(len(ast.errors), 0, "Test failed at parser.")
1138
1139     saErrors = SemanticAnalyzer.analyzeStart(ast)
1140     #should have an error saying mySecondNum doesnt exist
1141     self.assertEqual(len(saErrors), 1)
1142
1143     def test_general_access_undeclared_var(self):
1144         test_string = \
1145             """myNum is a number
1146 set myNum to mySecondNum
1147 """
1148
1149         ast = Parser.parseString(test_string)
1150         self.assertEqual(len(ast.errors), 0, "Test failed at parser.")
1151
1152         saErrors = SemanticAnalyzer.analyzeStart(ast)
1153         #should have an error saying mySecondNum doesnt exist
1154         self.assertEqual(len(saErrors), 1)
1155
1156     def test_var_declared_in_if(self):
1157         test_string = \
1158             """if 1
1159 {
1160     myNum is a number
1161     set myNum to 10
1162     print myNum
1163 }
1164 """
1165
1166         ast = Parser.parseString(test_string)
1167         self.assertEqual(len(ast.errors), 0, "Test failed at parser.")
1168
1169         saErrors = SemanticAnalyzer.analyzeStart(ast)
1170         self.assertEqual(len(saErrors), 0)
1171

```

```

1172     def test_var_declared_in_if_accessed_outside(self):
1173         test_string = \
1174             """if 1
1175 {
1176     myNum is a number
1177     set myNum to 10
1178     print myNum
1179 }
1180 print myNum
1181 """
1182
1183     ast = Parser.parseString(test_string)
1184     self.assertEqual(len(ast.errors), 0, "Test failed at parser.")
1185     saErrors = SemanticAnalyzer.analyzeStart(ast)
1186     #should have an error about printing myNum after the if
1187     self.assertEqual(len(saErrors), 1)
1188
1189     def test_var_declared_in_func(self):
1190         test_string = \
1191             """define moveForwardFive
1192 {
1193     myNum is a number
1194     set myNum to 5
1195     drive forward myNum steps
1196 }
1197 """
1198
1199     ast = Parser.parseString(test_string)
1200
1201     saErrors = SemanticAnalyzer.analyzeStart(ast)
1202     self.assertEqual(len(saErrors), 0)
1203
1204     def test_var_declared_in_func_accessed_outside(self):
1205         test_string = \
1206             """define moveForwardFive
1207 {
1208     myNum is a number

```

```

1209     set myNum to 5
1210     drive forward myNum steps
1211 }
1212 moveForwardFive
1213 drive forward myNum steps
1214 """
1215
1216     ast = Parser.parseString(test_string)
1217     self.assertEqual(len(ast.errors), 0, "Test failed at parser.")
1218
1219     saErrors = SemanticAnalyzer.analyzeStart(ast)
1220     self.assertEqual(len(saErrors), 1)
1221
1222     def test_access_passed_in_var_in_func(self):
1223         test_string = \
1224             """define moveForward using numSteps (number)
1225 {
1226     drive forward numSteps steps
1227 }
1228 """
1229
1230     ast = Parser.parseString(test_string)
1231     self.assertEqual(len(ast.errors), 0, "Test failed at parser.")
1232
1233     saErrors = SemanticAnalyzer.analyzeStart(ast)
1234     self.assertEqual(len(saErrors), 0)
1235
1236     def test_access_passed_in_var_in_if_in_func(self):
1237         test_string = \
1238             """define moveForwardFive using numSteps (number)
1239 {
1240     drive forward numSteps steps
1241     if 1
1242     {
1243         myCounter is a number
1244         set myCounter to 10
1245         drive forward 10 steps

```

```

1246     }
1247 }
1248 """
1249
1250     ast = Parser.parseString(test_string)
1251     self.assertEqual(len(ast.errors), 0, "Test failed at parser.")
1252
1253     saErrors = SemanticAnalyzer.analyzeStart(ast)
1254     self.assertEqual(len(saErrors), 0)
1255
1256     def test_call_func_anywhere(self):
1257         test_string = \
1258             """moveForwardFive
1259 define moveForwardFive
1260 {
1261     drive forward 5 steps
1262 }
1263 moveForwardFive
1264 """
1265
1266     ast = Parser.parseString(test_string)
1267     self.assertEqual(len(ast.errors), 0, "Test failed at parser.")
1268
1269     saErrors = SemanticAnalyzer.analyzeStart(ast)
1270     self.assertEqual(len(saErrors), 0)
1271
1272     def test_declare_func_in_func(self):
1273         test_string = \
1274             """define moveForwardFive
1275 {
1276     drive forward 5 steps
1277     define moveForwardTen
1278     {
1279         drive forward 10 steps
1280     }
1281 }
1282 """

```

```

1283
1284         ast = Parser.parseString(test_string)
1285         self.assertEqual(len(ast.errors), 0, "Test failed at parser.")
1286
1287         saErrors = SemanticAnalyzer.analyzeStart(ast)
1288         self.assertEqual(len(saErrors), 1)
1289
1290     def test_call_funciton_with_param(self):
1291         test_string = \
1292             """define moveForwardFive using numSteps (number)
1293 {
1294     drive forward numSteps steps
1295 }
1296 myNum is a number
1297 set myNum to 10
1298 moveForwardFive myNum
1299 """
1300
1301         ast = Parser.parseString(test_string)
1302         self.assertEqual(len(ast.errors), 0, "Test failed at parser.")
1303
1304         saErrors = SemanticAnalyzer.analyzeStart(ast)
1305         self.assertEqual(len(saErrors), 0)
1306
1307     def test_access_func_param_outside(self):
1308         test_string = \
1309             """define moveForwardFive using numSteps (number)
1310 {
1311     drive forward numSteps steps
1312 }
1313 myNum is a number
1314 set myNum to 10
1315 moveForwardFive myNum
1316 print numSteps
1317 """
1318
1319         ast = Parser.parseString(test_string)

```



```

1320         self.assertEqual(len(ast.errors), 0, "Test failed at parser.")
1321
1322         saErrors = SemanticAnalyzer.analyzeStart(ast)
1323         #should fail since numSteps is only inside the function
1324         self.assertEqual(len(saErrors), 1)
1325
1326     def test_assignments_var_to_var(self):
1327         test_string = \
1328             """myNum is a number
1329 set myNum to 10
1330 myWord is a word
1331 set myWord to "hello"
1332 set myNum to myWord
1333 set myWord to myNum
1334 """
1335
1336         ast = Parser.parseString(test_string)
1337         self.assertEqual(len(ast.errors), 0, "Test failed at parser.")
1338
1339         saErrors = SemanticAnalyzer.analyzeStart(ast)
1340         self.assertEqual(len(saErrors), 2)
1341
1342     def test_assignments_var_number(self):
1343         test_string = \
1344             """myWord is a word
1345 set myWord to 10
1346 """
1347
1348         ast = Parser.parseString(test_string)
1349         self.assertEqual(len(ast.errors), 0, "Test failed at parser.")
1350
1351         saErrors = SemanticAnalyzer.analyzeStart(ast)
1352         self.assertEqual(len(saErrors), 1)
1353
1354     def test_assignments_var_str_literal(self):
1355         test_string = \
1356             """myNum is a number

```

```

1357 set myNum to "hello"
1358 """
1359
1360         ast = Parser.parseString(test_string)
1361         self.assertEqual(len(ast.errors), 0, "Test failed at parser.")
1362
1363         saErrors = SemanticAnalyzer.analyzeStart(ast)
1364         self.assertEqual(len(saErrors), 1)
1365
1366     def test_compare_vars_num_to_string(self):
1367         test_string = \
1368             """myNum is a number
1369 set myNum to 10
1370 myWord is a word
1371 set myWord to "hello"
1372 if myWord > myNum
1373 {
1374     print "bad"
1375 }
1376 """
1377
1378         ast = Parser.parseString(test_string)
1379         self.assertEqual(len(ast.errors), 0, "Test failed at parser.")
1380
1381         saErrors = SemanticAnalyzer.analyzeStart(ast)
1382         self.assertEqual(len(saErrors), 1)
1383
1384     def test_compare_num_to_stringliteral(self):
1385         test_string = \
1386             """if 10 > "hello"
1387 {
1388     print "bad"
1389 }
1390 """
1391
1392         ast = Parser.parseString(test_string)
1393         self.assertEqual(len(ast.errors), 0, "Test failed at parser.")

```

```

1394
1395         saErrors = SemanticAnalyzer.analyzeStart(ast)
1396         self.assertEqual(len(saErrors), 1)
1397
1398     def test_compare_num_to_num(self):
1399         test_string = \
1400             """if 10 > 5
1401 {
1402     print "good"
1403 }
1404 """
1405
1406         ast = Parser.parseString(test_string)
1407         self.assertEqual(len(ast.errors), 0, "Test failed at parser.")
1408
1409         saErrors = SemanticAnalyzer.analyzeStart(ast)
1410         self.assertEqual(len(saErrors), 0)
1411
1412     def test_compare_strL_to_strL(self):
1413         test_string = \
1414             """if "hello" is "hi"
1415 {
1416     print "good"
1417 }
1418 """
1419
1420         ast = Parser.parseString(test_string)
1421         self.assertEqual(len(ast.errors), 0, "Test failed at parser.")
1422
1423         saErrors = SemanticAnalyzer.analyzeStart(ast)
1424         self.assertEqual(len(saErrors), 0)
1425
1426     def test_compare_strL_to_strL_not(self):
1427         test_string = \
1428             """if "hello" is not "hi"
1429 {
1430     print "good"

```

```

1431 }
1432 """
1433
1434     ast = Parser.parseString(test_string)
1435     self.assertEqual(len(ast.errors), 0, "Test failed at parser.")
1436
1437     saErrors = SemanticAnalyzer.analyzeStart(ast)
1438     self.assertEqual(len(saErrors), 0)
1439
1440     def test_compare_num_str_is(self):
1441         test_string = \
1442             """if 10 is "hello"
1443 {
1444     print "bad"
1445 }
1446 """
1447
1448     ast = Parser.parseString(test_string)
1449     self.assertEqual(len(ast.errors), 0, "Test failed at parser.")
1450
1451     saErrors = SemanticAnalyzer.analyzeStart(ast)
1452     self.assertEqual(len(saErrors), 1)
1453
1454     def test_num_params_passing(self):
1455         test_string = \
1456             """define moveForwardFiveAndTurn using numSteps (number)\
1457 and direction (word)
1458 {
1459
1460 }
1461 moveForwardFiveAndTurn 10 "left"
1462 """
1463
1464     ast = Parser.parseString(test_string)
1465     self.assertEqual(len(ast.errors), 0, "Test failed at parser.")
1466
1467     saErrors = SemanticAnalyzer.analyzeStart(ast)

```

```

1468         self.assertEqual(len(saErrors), 0)
1469
1470     def test_num_params_passing_2(self):
1471         test_string = \
1472             """define moveForwardFiveAndTurn using numSteps (number)\
1473 and direction (word)
1474 {
1475
1476 }
1477 moveForwardFiveAndTurn 10 "hi" "extra"
1478 """
1479
1480         ast = Parser.parseString(test_string)
1481         self.assertEqual(len(ast.errors), 0, "Test failed at parser.")
1482
1483         saErrors = SemanticAnalyzer.analyzeStart(ast)
1484         self.assertEqual(len(saErrors), 1)
1485
1486     def test_num_params_passing_3(self):
1487         test_string = \
1488             """define moveForwardFiveAndTurn using numSteps (number)\
1489 and direction (word)
1490 {
1491
1492 }
1493 moveForwardFiveAndTurn 10 15
1494 """
1495
1496         ast = Parser.parseString(test_string)
1497         self.assertEqual(len(ast.errors), 0, "Test failed at parser.")
1498
1499         saErrors = SemanticAnalyzer.analyzeStart(ast)
1500         self.assertEqual(len(saErrors), 1)
1501
1502     def test_num_params_passing_4(self):
1503         test_string = \
1504             """define moveForwardFiveAndTurn using numSteps (number)\

```

```

1505     and direction (word)
1506 {
1507     print numSteps
1508     print direction
1509 }
1510 moveForwardFiveAndTurn "hello" "hi"
1511 """
1512
1513     ast = Parser.parseString(test_string)
1514     self.assertEqual(len(ast.errors), 0, "Test failed at parser.")
1515
1516     saErrors = SemanticAnalyzer.analyzeStart(ast)
1517     self.assertEqual(len(saErrors), 1)
1518
1519     def test_num_params_passing_5(self):
1520         test_string = \
1521             """define moveForwardFiveAndTurn using numSteps (number)\
1522 and direction (word)
1523 {
1524     print numSteps
1525     print direction
1526 }
1527 moveForwardFiveAndTurn "left" 10
1528 """
1529
1530     ast = Parser.parseString(test_string)
1531     self.assertEqual(len(ast.errors), 0, "Test failed at parser.")
1532
1533     saErrors = SemanticAnalyzer.analyzeStart(ast)
1534     self.assertEqual(len(saErrors), 1)
1535
1536     def test_built_in_functions_params(self):
1537         test_string = \
1538             """drive forwards five steps
1539 """
1540
1541     ast = Parser.parseString(test_string)

```

```

1542         self.assertEqual(len(ast.errors), 0, "Test failed at parser.")
1543
1544         saErrors = SemanticAnalyzer.analyzeStart(ast)
1545         self.assertEqual(len(saErrors), 1)
1546
1547     def test_while_loop(self):
1548         test_string = \
1549             """repeat 5 times
1550 {
1551     print "hi"
1552 }
1553 """
1554
1555         ast = Parser.parseString(test_string)
1556         self.assertEqual(len(ast.errors), 0, "Test failed at parser.")
1557
1558         saErrors = SemanticAnalyzer.analyzeStart(ast)
1559         self.assertEqual(len(saErrors), 0)
1560
1561     def test_while_loop_bad(self):
1562         test_string = \
1563             """repeat five times
1564 {
1565     print "hi"
1566 }
1567 """
1568
1569         ast = Parser.parseString(test_string)
1570         self.assertEqual(len(ast.errors), 0, "Test failed at parser.")
1571
1572         saErrors = SemanticAnalyzer.analyzeStart(ast)
1573         self.assertEqual(len(saErrors), 1)
1574
1575     def test_calling_nonexistant_function(self):
1576         test_string = \
1577             """fullTurn "left"
1578 """

```

```

1579
1580     ast = Parser.parseString(test_string)
1581     self.assertEqual(len(ast.errors), 0, "Test failed at parser.")
1582
1583     saErrors = SemanticAnalyzer.analyzeStart(ast)
1584     self.assertEqual(len(saErrors), 1)
1585
1586     def test_course2(self):
1587         test_string = \
1588             """define fullTurn using direction (word)
1589 {
1590     if direction is "left"
1591     {
1592         turn left
1593         turn left
1594     }
1595     else
1596     {
1597         turn right
1598         turn right
1599     }
1600 }
1601
1602 define driveThenFullTurn using numSteps (number) and direction (word)
1603 {
1604     drive forward numSteps steps
1605     fullTurn direction
1606 }
1607
1608 driveThenFullTurn 0 "right"
1609 driveThenFullTurn 25 "left"
1610 driveThenFullTurn 18 "left"
1611
1612 repeat 2 times
1613 {
1614
1615     driveThenFullTurn 50 "right"

```



```

1616     driveThenFullTurn 15 "right"
1617     driveThenFullTurn 50 "left"
1618     driveThenFullTurn 16 "left"
1619 }
1620
1621 driveThenFullTurn 50 "right"
1622 driveThenFullTurn 15 "right"
1623 driveThenFullTurn 50 "left"
1624 driveThenFullTurn 10 "left"
1625 """
1626
1627     ast = Parser.parseString(test_string)
1628     self.assertEqual(len(ast.errors), 0, "Test failed at parser.")
1629
1630     saErrors = SemanticAnalyzer.analyzeStart(ast)
1631     self.assertEqual(len(saErrors), 0)
1632
1633     def test_var_uninitialized(self):
1634         test_string = \
1635             """myNum is a number
1636 drive forward myNum steps
1637 """
1638
1639         ast = Parser.parseString(test_string)
1640         self.assertEqual(len(ast.errors), 0, "Test failed at parser.")
1641
1642         saErrors = SemanticAnalyzer.analyzeStart(ast)
1643         self.assertEqual(len(saErrors), 1)
1644
1645     def test_call_function_in_if_and_while(self):
1646         test_string = \
1647             """moveForwardFiveAndTurn 10 "left"
1648 if 1
1649 {
1650     moveForwardFiveAndTurn 10 "left"
1651 }
1652 repeat 2 times

```

```

1653 {
1654     moveForwardFiveAndTurn 10 "left"
1655 }
1656 define moveForwardFiveAndTurn using numSteps (number) and direction (word)
1657 {
1658     print numSteps
1659     print direction
1660     fullTurn direction
1661 }
1662 define fullTurn using direction (word)
1663 {
1664     if direction is "left"
1665     {
1666         turn left
1667         turn left
1668     }
1669     else
1670     {
1671         turn right
1672         turn right
1673     }
1674     moveForwardFiveAndTurn 5 direction
1675 }
1676 """
1677
1678     ast = Parser.parseString(test_string)
1679     self.assertEqual(len(ast.errors), 0, "Test failed at parser.")
1680
1681     saErrors = SemanticAnalyzer.analyzeStart(ast)
1682     self.assertEqual(len(saErrors), 0)
1683
1684     def test_call_function_before_declared(self):
1685         test_string = \
1686             """moveForwardFiveAndTurn 10 "left"
1687 if 1
1688 {
1689     moveForwardFiveAndTurn 10 "left"

```

```

1690 }
1691 repeat 2 times
1692 {
1693     moveForwardFiveAndTurn 10 "left"
1694 }
1695 drive forward 5 steps
1696 define moveForwardFiveAndTurn using numSteps (number) and direction (word)
1697 {
1698     print numSteps
1699     print direction
1700 }
1701 """
1702
1703     ast = Parser.parseString(test_string)
1704     self.assertEqual(len(ast.errors), 0, "Test failed at parser.")
1705
1706     saErrors = SemanticAnalyzer.analyzeStart(ast)
1707     self.assertEqual(len(saErrors), 0)
1708
1709     def test_template(self):
1710         test_string = \
1711             """
1712 """
1713
1714         ast = Parser.parseString(test_string)
1715         self.assertEqual(len(ast.errors), 0, "Test failed at parser.")
1716
1717         saErrors = SemanticAnalyzer.analyzeStart(ast)
1718         self.assertEqual(len(saErrors), 0)
1719
1720 if __name__ == '__main__':
1721     suite = unittest.TestLoader().loadTestsFromTestCase(TranslatorTests)
1722     unittest.TextTestRunner(verbosity=2).run(suite)

```