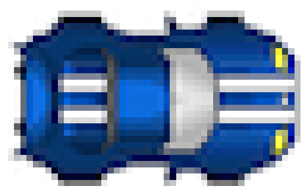
Racecar

A Programming Language for Kids



May 12, 2013

Team 19

Project Manager	Samuel Kohn
Language Guru	Alexander Fields
System Architect	Jeremy Spencer
System Integrator	Mason Silber
Verification and Validation	Colfax Selby
Mentor	Melanie Kambadur
Professor	Alfred Aho

Contents

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	`def myMethod(var1, var2):`	`define myMethod using var1 (number) and var2 (word)`	Uses full words instead of abbreviations and punctuation
Iteration	`for i in range(5):`	`repeat 5 times`	Uses simple, clear, intuitive keywords
Method invocation	`car.drive(direction.FORWARD, 10)`	`drive forward 5 steps`	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 form 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. However, 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 elseIf statements after an if statement, like the following:

```
:) Print out a message describing any obstacles if canMove left
{
    print "You can move left!
}
elseIf 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 elseIf immediately preceding it, and it is also a problem if there is an elseIf 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 do 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 paramter, 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>
}
elseIf <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_bloc

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:

Messager 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 is 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-

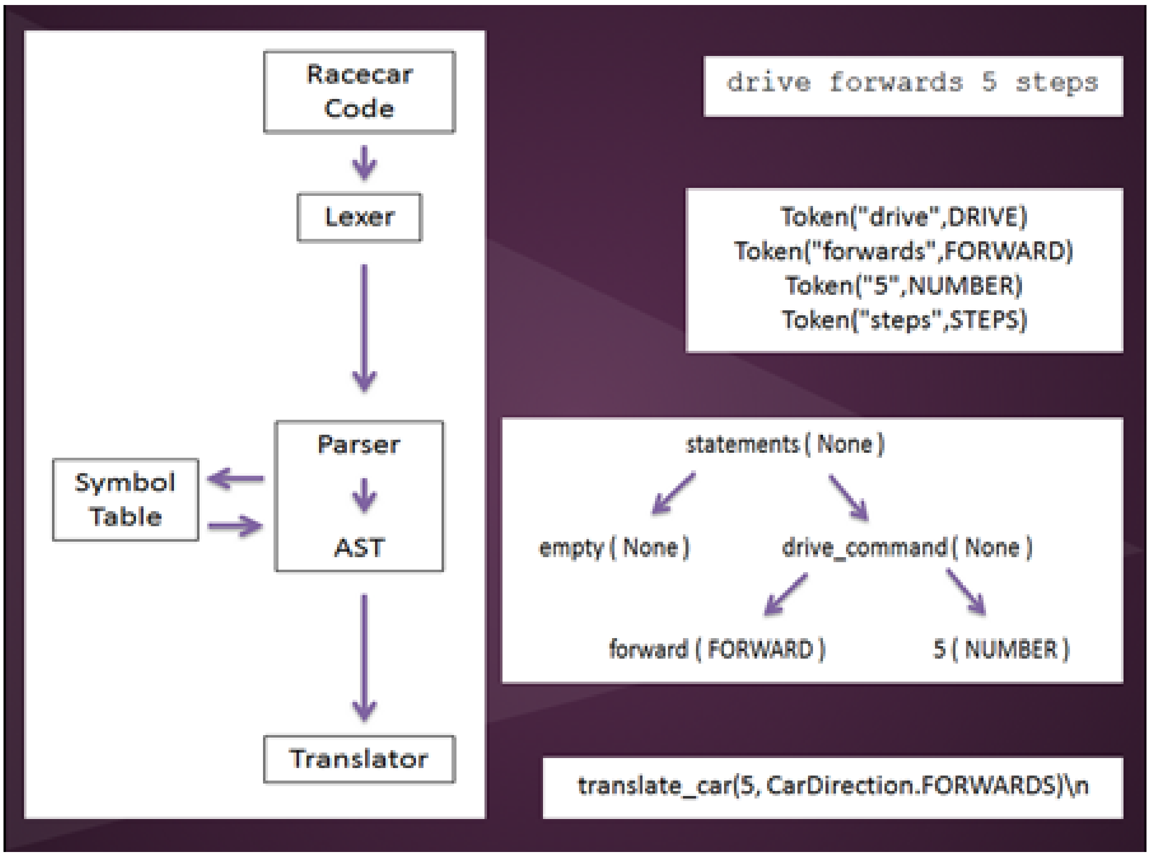


Figure 1: System Architecture

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 turnLeftThenDriveStraight using numStepsTurn \ (number) and numStepsDrive (number) { turn left drive forward numStepsTurn steps turn right drive forward numStepsDrive steps } turnLeftThenDriveStraight 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) turnLeftThenDriveStraight(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 check 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 auxilary 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`

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

10.2 Graphical Interface

Listing 2: `racecar.py`

```
# Written by Mason Silber and Sam Kohn

from Tkinter import *
from PIL import Image
from PIL import ImageTk
```

```
import tkFileDialog
import tkMessageBox
import re
import time
import Racecar.Tree
import Racecar.Compiler
import random
import pdb
import math

random.seed()

#current_program ised used to store the current file open in order to save back
#to that file
current_program = None

#Variable that serves as an interrupt to stop the program
should_stop = False
collision_occurred = False

#List of obstacles on the course at any given time
obstacles = []

#list of walls on the course at any given time
walls = []

#grid ticks
grid_ticks = []


class Obstacle:
    def __init__(self, x, y, width, height):
        self.obstacle_object = canvas.create_oval(
            x-width/2,
            y-height/2,
            x+width/2,
            y+height/2,
```

```
            fill="#000")
        self.width = width
        self.height = height
        self.center = (x, y)
        self.radius = width/2


#Wall class: can only be vertical or horizontal
class Wall:
    def __init__(self, start_x, start_y, length, is_horizontal):
        if is_horizontal:
            self.wall_object = canvas.create_line(
                start_x,
                start_y,
                start_x+length,
                start_y)
            self.start = start_x
            self.end = start_x+length
            self.constant_coord = start_y
        else:
            self.wall_object = canvas.create_line(
                start_x,
                start_y,
                start_x,
                start_y+length)
            self.start = start_y
            self.end = start_y+length
            self.constant_coord = start_x
        self.is_horizontal = is_horizontal


class Program:
    def __init__(self):
        self.name = ''
        self.file_obj = None


```

```
#Static variables for turning the car
class WheelDirection:
    LEFT = 1
    RIGHT = -1


#Car direction object
#X and Y can be 1,0,-1 respectively. The only invalid combination is when x = 0
#and y = 0. Positive axes point right and up respectively
class CarDirection:
    FORWARDS = 1
    BACKWARDS = -1

    def __init__(self):
        self.direction = 0

    DIRECTIONS = [(1, 0),
                  (1, -1),
                  (0, -1),
                  (-1, -1),
                  (-1, 0),
                  (-1, 1),
                  (0, 1),
                  (1, 1)]

    def get_direction(self):
        return CarDirection.DIRECTIONS[self.direction]

    def turn_right(self):
        self.direction = (self.direction - 1) % len(CarDirection.DIRECTIONS)

    def turn_left(self):
        self.direction = (self.direction + 1) % len(CarDirection.DIRECTIONS)

    def opposite_direction(self):
        return DIRECTIONS[
            (self.direction + len(CarDirection.DIRECTIONS)/2) %
```

```
            len(CarDirection.DIRECTIONS)]


class Car:
    def __init__(self):
        self.position_x = 0
        self.position_y = 0
        #Car direction starts facing right
        self.car_direction = CarDirection()
        self.image = None
        self.image_tk = None
        self.car_object = None
        self.width = 27
        self.height = 27
        self.radius = 25

    #Drive method that updates the car's position (in the model, not on the UI)
    #UI animation will need to be done moving x and y simultaneously
    def update_position(self, steps, movement_direction):
        self.position_x += (
            self.car_direction.get_direction()[0]
            * steps
            * movement_direction)

        self.position_y += (
            self.car_direction.get_direction()[1]
            * steps
            * movement_direction)


#Function to get a unique position of object, in order to detect for collisions
def get_position(x, y):
    return 1000 * int(x) + int(y)


def getCurrentPosition():
    global car
```

```python
    return get_position(car.position_x, car.position_y)


#Checks if there is going to be a collision on the upcoming path
#drive_direction has to be CarDirection.FORWARDS or CarDirection.BACKWARDS
def can_move(num_steps, drive_direction):
    global car
    curr_x = int(car.position_x)
    curr_y = int(car.position_y)
    direction = car.car_direction.get_direction()
    path = []

    #If the direction is backwards, just reverse the direction
    if drive_direction == CarDirection.BACKWARDS:
        direction = car.car_direction.opposite_direction()

    #Create path coordinates
    for i in range(0, steps_to_pixels(num_steps)):
        pos = (curr_x + i * direction[0], curr_y + i * direction[1])
        path.append(pos)

    #Check each point in the path to see if it collides with any of the
    #obstacles
    for pos in path:
        if is_collision(pos[0], pos[1]):
            return False

    return True


#Number of steps on screen is proportional to screen size
def steps_to_pixels(steps):
    return canvas_frame.winfo_reqwidth()/110*steps


#Function to find the distance between two points
def distance_between_points(x_1, y_1, x_2, y_2):
```

```
    return math.sqrt(math.pow((x_2-x_1), 2) + math.pow((y_2-y_1), 2))


#API Functions
#direction must be either CarDirection.FORWARDS or CarDirection.BACKWARDS
def translate_car(steps, direction):
    global car
    global should_stop
    global collision_occurred

    steps = int(steps)
    direction = int(direction)

    curr_x = car.position_x
    curr_y = car.position_y

    one_step = steps_to_pixels(1)

    for i in range(0, steps_to_pixels(int(steps))):
        #Check interrupt variable
        if should_stop and i % one_step == 0:
            return

        time.sleep(0.01)
        #car_direction is FORWARDS or BACKWARDS (1 and -1 respectively)

        if is_collision(curr_x, curr_y):
            print_to_console("COLLISION")
            #Stop execution of program
            #TODO Deal with delay on collision
            should_stop = True
            collision_occurred = True
            return
        else:
            canvas.move(
                car.car_object,
                direction * car.car_direction.get_direction()[0],
```

```
                direction * car.car_direction.get_direction()[1])

            curr_x = curr_x + direction * car.car_direction.get_direction()[0]
            curr_y = curr_y + direction * car.car_direction.get_direction()[1]
            canvas.update()

        car.update_position(1, direction)


#direction must be WheelDirection.LEFT or WheelDirection.RIGHT
#Note: only check interrupt variable at the beginning, because
#we shouldn't allow partial rotations
def rotate_car(direction):
    global car
    global should_stop

    #Check interrupt variable
    if should_stop:
        return

    #This is current index in DIRECTIONS array
    current_direction_deg = car.car_direction.direction*45

    if direction == WheelDirection.LEFT:
        car.car_direction.turn_left()
    elif direction == WheelDirection.RIGHT:
        car.car_direction.turn_right()
    else:
        return

    for i in range(0, 45):
        time.sleep(0.01)
        canvas.delete(car.car_object)

        if direction == WheelDirection.LEFT:
            car.image_tk = ImageTk.PhotoImage(
                car.image.rotate(current_direction_deg + i))
```

```python
        elif direction == WheelDirection.RIGHT:
            car.image_tk = ImageTk.PhotoImage(
                car.image.rotate(current_direction_deg - i))
        else:
            return

        car.car_object = canvas.create_image(
            car.position_x,
            car.position_y,
            image=car.image_tk)
        canvas.update()


#Check for collision with walls of the maze and a finish line
def collision_with_internal_walls(pos_x, pos_y):
    for wall in walls:
        #horizontal wall
        if wall.is_horizontal:
            #in rance of wall
            if wall.start <= pos_x <= wall.end:
                #Current direction of the car
                direction = car.car_direction.get_direction()
                #distance from wall
                dist_to_wall = math.fabs(pos_y-wall.constant_coord)
                #Car is horizontally oriented
                if direction == (1, 0) or direction == (-1, 0):
                    if dist_to_wall < car.radius/2:
                        return True
                    #Check for collision with car
                #Car is not horizontally oriented
                else:
                    if dist_to_wall < car.radius:
                        return True
        #vertical wall
        else:
            #in range of wall
            if wall.start <= pos_y <= wall.end:
```

```
                direction = car.car_direction.get_direction()
                #distance from wall
                dist_to_wall = math.fabs(pos_x-wall.constant_coord)
                #Car is vertically oriented
                if direction == (0, 1) or direction == (0, -1):
                    if dist_to_wall < car.radius/2:
                        return True
                #Car is not vertically oriented
                else:
                    if dist_to_wall < car.radius:
                        return True

    return False


def is_collision(curr_x, curr_y):
    #Check for collisions with obstacles and walls

    #Check obstacles
    for obstacle in obstacles:
        distance = distance_between_points(
            curr_x,
            curr_y,
            obstacle.center[0],
            obstacle.center[1])
        if distance < (car.radius + obstacle.radius):
            return True
    #check internal walls for collision
    if collision_with_internal_walls(curr_x, curr_y):
        return True
    #Check boundary walls
    elif not (origin[0] <= curr_x <= anti_origin[0]):
        return True
    elif not (origin[1] <= curr_y <= anti_origin[1]):
        return True
    else:
        return False
```

```python


def print_to_console(message):
#Should console be cleared each time the program is restart?
#Or should there be a button?
    console.config(state=NORMAL)
    console.insert(END, str(message) + '\n')
    console.config(state=DISABLED)


#Course generation functions

#Course one is a slalom of blocks
def course_one():
    clear_course()
    obstacle_coord_x = 123
    obstacle_coord_y = int(canvas.winfo_reqheight())/2
    while obstacle_coord_x < anti_origin[0]:
        obstacle = Obstacle(obstacle_coord_x, obstacle_coord_y, 30, 30)
        obstacles.append(obstacle)
        obstacle_coord_x = obstacle_coord_x + 150


#TODO -- Fill in the rest of the courses
#Course two is a simple maze
finish_line = None


def course_two():
    global finish_line

    clear_console()

    message = "Try to navigate through the maze and cross the finish line!"
    print_to_console(message)

    clear_course()
```

```
    wall_coord_x = 123
    wall_length = 4*int(canvas.winfo_reqheight())/5

    #used to toggle position of line
    put_wall_on_top = True

    #walls
    while wall_coord_x < anti_origin[0]:
        if put_wall_on_top:
            wall = Wall(
                wall_coord_x,
                0,
                wall_length,
                False)
            walls.append(wall)
        else:
            wall = Wall(
                wall_coord_x,
                int(canvas.winfo_reqheight())/5+23,
                wall_length,
                False)
            walls.append(wall)
        put_wall_on_top = not put_wall_on_top
        wall_coord_x = wall_coord_x+100

    #finish line
    wall_coord_x = wall_coord_x-100
    finish_line = canvas.create_line(
        wall_coord_x,
        wall_length,
        wall_coord_x,
        canvas.winfo_reqheight()+23,
        fill="black",
        dash=(4, 4))


def course_three():
```

```
    clear_course()

    max_x = canvas.winfo_reqwidth()
    max_y = canvas.winfo_reqheight()

    while len(obstacles) < 30:
        pos_x = random.randrange(0, max_x, 1)
        pos_y = random.randrange(0, max_y, 1)
        radius = random.randrange(10, 50, 1)

        if is_collision(pos_x, pos_y):
            continue
        #Check for collision with car
        elif distance_between_points(
                pos_x,
                pos_y,
                car.position_x,
                car.position_y) < (car.radius + radius):
            continue
        else:
            obstacle = Obstacle(pos_x, pos_y, radius, radius)
            obstacles.append(obstacle)


def course_four():
    clear_course()
    obstacle_coord_x = 123
    obstacle_coord_y = 60
    while obstacle_coord_y < anti_origin[1]:
        while obstacle_coord_x < anti_origin[0]:
            obstacle = Obstacle(obstacle_coord_x, obstacle_coord_y, 30, 30)
            obstacles.append(obstacle)
            obstacle_coord_x = obstacle_coord_x + 150
        obstacle_coord_y = obstacle_coord_y + 80
        obstacle_coord_x = 123

    for obstacle in obstacles:
```

```
            print obstacle.center


def course_five():
    clear_course()


def clear_course():
    global obstacles
    global walls
    global finish_line
    #remove obstacles from the course
    for obstacle in obstacles:
        canvas.delete(obstacle.obstacle_object)

    #Needs to take care of finish line too, which isn't a wall object
    for wall in walls:
        canvas.delete(wall.wall_object)

    if finish_line is not None:
        canvas.delete(finish_line)
    #clear the obstacles and walls array
    obstacles = []
    walls = []
    finish_line = None


#Menu functions
def open_file():
    global current_program

    #Keep returning to the file dialog if they didn't select a .race file
    while True:
        file_name = tkFileDialog.askopenfilename(defaultextension=".race")
        if file_name == '':
            return

```

```
        #Check validity of file being opened
        file_regex = re.compile("\w*\.race$")
        if len(file_regex.findall(file_name)) == 0:
            tkMessageBox.showwarning(
                "Open File Error",
                "You must open a .race file")
        else:
            break

    file_object = open(file_name, 'r')
    current_program = Program()
    current_program.name = file_name
    current_program.file_obj = file_object
    code.delete(1.0, END)
    code.insert(1.0, file_object.read())
    current_program.file_obj.close()


def save():
    global current_program
    if current_program is None:
        save_file_as()
    else:
        save_file()


def save_file():
    global current_program
    if not current_program.file_obj.closed:
        current_program.file_obj.close()
    #Open file for writing (will clear it)
    current_program.file_obj = open(current_program.name, 'w')
    current_program.file_obj.truncate()
    current_program.file_obj.write(code.get(1.0, END))
    current_program.file_obj.close()
```

```
def save_file_as():
    global current_program
    file_name = tkFileDialog.asksaveasfilename(defaultextension=".race")

    #Defaults to saving on the desktop
    if file_name == '':
        file_name = '~/Desktop/racecar_program.race'

    current_program = Program()
    current_program.name = file_name
    current_program.file_obj = open(file_name, 'w')
    current_program.file_obj.write(code.get(1.0, END))
    current_program.file_obj.close()


def clear():
    if code.get(1.0, END) == '':
        return

    if tkMessageBox.askyesno(
            "Clear code",
            "Are you sure you want to delete all of your code?"):
        code.delete(1.0, END)


def clear_console():
    console.config(state=NORMAL)
    console.delete(1.0, END)
    console.config(state=DISABLED)


#Triggers interrupt
def stop_program():
    global should_stop
    should_stop = True
```

```
#Code generation and compilation
#Runs code
def generate_program(code):
    global should_stop
    global collision_occurred

    #Set the interrupt variable whenever a program is run
    should_stop = False
    collision_occurred = False
    if len(code) > 1:
        #print code[:-1]
        #demo(code)
        python_code, errors, correct = verify_program(code)
        if(correct):
            #Print message to console saying program is executing
            print_to_console("Program executing")
            console.tag_add("Correct", "1.0", "1.end")
            console.tag_config("Correct", foreground="Green")

            #Toggle the buttons on the bottom and run program
            toggle_buttons(True)
            tempGlobal = globals().copy()
            exec(python_code, tempGlobal)
            toggle_buttons(False)

            #If collision occurred
            if should_stop:
                if collision_occurred:
                    if tkMessageBox.showwarning(
                                "Oops!", "You crashed! Try again"):
                        reset_car_position()
            else:
                #Print message to console saying program is finished executing
                print_to_console("Done running program")
                console.tag_add("End", "end -2 l", END)
                console.tag_config("End", foreground="Green")
                print "OUTPUT: " + console.index("end -1 l")
```

```
        else:
            #Print message to console saying program has errors
            print_to_console(
                "You have " +
                str(len(errors)) +
                " error(s) in your program")
            console.tag_add("Error", "1.0", "1.end")
            console.tag_config("Error", foreground="Red")

            for error in errors:
                print_to_console(error)
    else:
        print "Blank"


#Checks if program is a valid Racecar program and returns corresponding python
#code if necessary
def verify_program(code):
    clear_console()
    if len(code) < 2:
        return ("BLANK", False)
    code, errors = Racecar.Compiler.getPythonCode(code)
    if errors:
        return (code, errors, False)
    else:
        return (code, errors, True)


#Called when verify program is called
def verify_program_callback(code):
    verification = verify_program(code)
    if verification[2]:
        print_to_console("Program syntax correct")
        console.tag_add("Correct", "1.0", "1.end")
        console.tag_config("Correct", foreground="Green")
    else:
        errors = verification[1]
```

```
        print_to_console(
            "You have " +
            str(len(errors)) +
            " error(s) in your program")
        console.tag_add("Error", "1.0", "1.end")
        console.tag_config("Error", foreground="Red")

        for error in errors:
            print_to_console(error)


#Resets car's position and orientation to original
def reset_car_position():
        global car
        canvas.delete(car.car_object)
        car.image_tk = ImageTk.PhotoImage(car.image)
        car_height = int(canvas.winfo_reqheight())/2
        car.car_object = canvas.create_image(
            23,
            car_height,
            image=car.image_tk)
        car.position_x = 23
        car.position_y = car_height
        car.car_direction = CarDirection()

#car object
car = Car()


#User interface
#Toggle enabled and disabled buttons when program is run and stopped
def toggle_buttons(stop_button_should_be_enabled):
    if stop_button_should_be_enabled:
        run_button.config(state=DISABLED)
        stop_button.config(state=NORMAL)
        reset_car_position_button.config(state=DISABLED)
        clear_button.config(state=DISABLED)
```

```
    else:
        run_button.config(state=NORMAL)
        stop_button.config(state=DISABLED)
        reset_car_position_button.config(state=NORMAL)
        clear_button.config(state=NORMAL)

root = Tk()
root.title('Racecar')
#Height is always three fourths the width of the window
window_width = root.winfo_screenwidth() - 100
window_height = 9*window_width/16
root.geometry("%dx%d" % (window_width, window_height))
root.resizable(width=FALSE, height=FALSE)

menu_bar = Menu(root)

menu = Menu(menu_bar, tearoff=0)
menu.add_command(label="Open", command=open_file)
menu.add_command(label="Save", command=save)
menu.add_command(label="Save As", command=save_file_as)
menu.add_separator()
menu.add_command(label="Quit", command=exit)
menu_bar.add_cascade(label="File", menu=menu)

menu = Menu(menu_bar, tearoff=0)

command = lambda: verify_program_callback(code.get(1.0, END))
menu.add_command(label="Verify Code", command=command)

command = lambda: generate_program(code.get(1.0, END))
menu.add_command(label="Run Code", command=command)

menu.add_command(label="Clear Code", command=clear)
menu.add_command(label="Clear Console", command=clear_console)
menu_bar.add_cascade(label="Code", menu=menu)

menu = Menu(menu_bar, tearoff=0)
```

```
menu.add_command(label="Course 1", command=course_one)
menu.add_command(label="Course 2", command=course_two)
menu.add_command(label="Course 3", command=course_three)
menu.add_command(label="Course 4", command=course_four)
menu.add_command(label="Course 5", command=course_five)
menu.add_separator()
menu.add_command(label="Clear course", command=clear_course)
menu_bar.add_cascade(label="Courses", menu=menu)

root.config(menu=menu_bar)

#frame for left side of window
left_frame = Frame(root)

#label for code window
code_label = Label(left_frame, text="Enter code here", anchor=W, pady=5)

#frame for code window to hold textbox and scrollbar
code_frame = Frame(
    left_frame,
    width=int(0.3*window_width),
    height=9*window_height/10)
code_frame.grid_propagate(False)

#scrollbar for code window
code_scrollbar = Scrollbar(code_frame)
code_scrollbar.pack(side=RIGHT, fill=Y)

#code is the window in which the code is written
code = Text(
    code_frame,
    width=50,
    #height=window_height/16-8,
    wrap=WORD,
    yscrollcommand=code_scrollbar.set)

#Frame for buttons
```

```
button_frame = Frame(left_frame)

#run_button passes code into a run program method
command = lambda: generate_program(code.get(1.0, END))
run_button = Button(
    button_frame,
    text="Run Code",
    pady=5,
    padx=5,
    command=command)

#Stop execution of running program
stop_button = Button(
    button_frame,
    text="Stop Program",
    padx=5,
    pady=5,
    command=stop_program)
stop_button.config(state=DISABLED)

#reset car position button puts the car back in its original position and
#orientation
reset_car_position_button = Button(
    button_frame,
    text="Reset Car Position",
    pady=5,
    padx=5,
    command=reset_car_position)

#clear_button clears the code in the text box
clear_button = Button(
    button_frame,
    text="Clear Code",
    command=clear)

#canvas is where the car will go
canvas_frame = Frame(
```

```
    root,
    width=window_width/1.5,
    height=window_height/1.5,
    padx=2,
    pady=2)

canvas_frame.configure(borderwidth=1.5, background='black')
canvas = Canvas(
    canvas_frame,
    width=window_width/1.5,
    height=window_height/1.5)

car.image = Image.open('Racecar/RacecarGUI/images/racecar.png')
car.image_tk = ImageTk.PhotoImage(car.image)

car.car_object = canvas.create_image(
    23,
    int(canvas.winfo_reqheight())/2,
    image=car.image_tk)

car.position_x = 23
car.position_y = int(canvas.winfo_reqheight())/2

#label above the console
console_label = Label(root, text="Console", anchor=W, pady=5)

#frame for the console to hold the textbox and the scrollbar
console_frame = Frame(root)

#scrollbar for the console
console_scrollbar = Scrollbar(console_frame)
console_scrollbar.pack(side=RIGHT, fill=Y)

#console to print to
console = Text(
    console_frame,
    width=int(window_width/1.5),
```

```
    height=8,
    padx=2,
    pady=2,
    wrap=WORD,
    yscrollcommand=console_scrollbar.set)

console.config(state=DISABLED)

#add them to GUI Window
#These are grouped logically in order to better see what's going on
left_frame.pack(side=LEFT, fill=BOTH)

code_label.pack()

code_frame.pack(expand=1, fill=BOTH)
code.pack(expand=1, fill=BOTH)

button_frame.pack(fill=BOTH)
run_button.grid(row=1, column=1)
stop_button.grid(row=1, column=2)
reset_car_position_button.grid(row=1, column=3)
clear_button.grid(row=1, column=4)

canvas_frame.pack(expand=1, fill=BOTH)
canvas.pack(expand=1, fill=BOTH)

console_label.pack()

console_frame.pack(expand=1, fill=BOTH, pady=(0, 10))
console.pack(expand=1, fill=BOTH)

code_scrollbar.config(command=code.yview)
console_scrollbar.config(command=console.yview)

root.update_idletasks()

#Origin and antiorigin are limits on the canvas where the car moves
```

```
origin = (23, 26)
anti_origin = (
    23+106*canvas_frame.winfo_width()/110,
    26+56*canvas_frame.winfo_width()/110)

#horizontal grid lines
position = 0
while position < anti_origin[0]:
    tick = canvas.create_line(
        position,
        anti_origin[1]-5+35,
        position,
        anti_origin[1]+35,
        fill="#000",
        width=2)
    grid_ticks.append(tick)
    position += steps_to_pixels(5)

#vertical grid lines
position = 0
while position < anti_origin[1]:
    tick = canvas.create_line(
        0,
        position,
        5,
        position,
        fill="#000",
        width=2)
    grid_ticks.append(tick)
    position += steps_to_pixels(5)

print anti_origin
#Run the GUI
root.mainloop()
```

10.3 Lexer and Parser

Listing 3: `Parser.py`

```python
# Written by Sam Kohn and Jeremy Spencer

import ply.lex as lex
import ply.yacc as yacc
from Tree import *

reserved = {
    'drive': 'DRIVE',
    'forward': 'FORWARD',
    'forwards': 'FORWARD',
    'backward': 'BACKWARD',
    'backwards': 'BACKWARD',
    'number': 'NUMBER_TYPE',
    'word': 'WORD_TYPE',
    'step': 'STEP',
    'steps': 'STEP',
    'turn': 'TURN',
    'left': 'LEFT',
    'right': 'RIGHT',
    'canDrive': 'CAN_DRIVE',
    'getCarPosition': 'GET_CAR_POSITION',
    'define': 'DEFINE',
    'using': 'USING',
    'and': 'AND',
    'print': 'PRINT',
    'elseIf': 'ELSE_IF',
    'if': 'IF',
    'else': 'ELSE',
    'repeat': 'REPEAT',
    'times': 'TIMES',
    'a': 'A',
    'is': 'IS',
    'not': 'NOT',
    'set': 'SET',
    'to': 'TO',
}
```

```python
tokens = [
    "NUMBER",
    "WORD",
    "ID",
    "GT",
    "LT",
    "GEQ",
    "LEQ",
    "CONCAT",
    "NEWLINE",
    "SINGLE_LINE_COMMENT",
] + list(set(reserved.values()))

literals = "{}()+-*/"

#t_NUMBER = r'[0-9]+'
#t_WORD = r'".*?"'
t_GT = r'>'
t_LT = r'<'
t_GEQ = r'>='
t_LEQ = r'<='
t_CONCAT = r'\+\+'
t_SINGLE_LINE_COMMENT = r':\).*$'
t_ignore = ' \t'


def t_ID(t):
    r'[A-Za-z][A-Za-z0-9]*'
    t.type = reserved.get(t.value, 'ID')
    t.value = (t.value, t.type, t.lexer.lineno)
    return t


def t_NUMBER(t):
    r'[0-9]+'
```

```
    t.value = (t.value, t.type, t.lexer.lineno)
    return t


def t_WORD(t):
    r'".*?"'
    t.value = (t.value, t.type, t.lexer.lineno)
    return t


def t_NEWLINE(t):
    r'\n|;|:-\((.|\n)*?:-\)'
    # \n is for actual newlines
    # ; is for debugging use
    # the next expression is for multiline comments. it is an adaptation of
    # hw1, problem 2.
    # the last expression :\).* matches single-line comments
    t.lexer.lineno += 1
    return t


def t_error(t):
    print "Illegal character '%s' at line '%s'" % (t.value[0], t.lexer.lineno)
    t.lexer.skip(1)
    t.value = (t.value, "ERROR", t.lexer.lineno)
    return t


def p_error(p):
    if p is None:
        raise SyntaxError("Reached end of file unexpectedly!")
    elif p.value[0] is None:
        print "Lexing Error with character ", p.value[1]
        p.value = p.value[1]
    else:
        print "Syntax error at token ", p.type

```

```python

def makeParseTreeNode(p, value):
    '''Returns a Tree object containing
         as children p[1:] and a value of value'''
    toReturn = Tree()
    for element in p[1:]:
        if type(element) == type(toReturn):
            toReturn.children.append(element)
            toReturn.errors += element.errors
        else:
            # the element is not a tree. wrap it in a tree
            newElement = Tree()
            if isinstance(element, tuple):
                newElement.value = element[0]
                newElement.type = element[1]
            else:
                newElement.value = element
            toReturn.children.append(newElement)

    if isinstance(value, tuple):
        toReturn.value = value[0]
        toReturn.type = value[1]
    else:
        toReturn.value = value
    if value == "error":
        errorMessage = str(p[1][2]) + ": " + p[1][0]
        toReturn.errors.append(errorMessage)

    return toReturn


def p_statements(p):
    '''statements : statements statement'''
    p[0] = makeParseTreeNode(p, "statements")


def p_error_statement(p):
```

```
    '''statement : error NEWLINE'''
    if not isinstance(p[1], tuple):
        p[1] = p[1].value
    p[0] = makeParseTreeNode(p, "error")


def p_statements_empty(p):
    '''statements : empty'''
    p[0] = p[1]


def p_statement_block(p):
    """statement_block : '{' statements '}' newline_opt_comment"""
    p[0] = makeParseTreeNode([p[0], p[2]], "statement_block")


def p_empty(p):
    '''empty :'''
    p[0] = Tree()
    p[0].value = "empty"


def p_newline_opt_comment(p):
    '''newline_opt_comment : opt_comment NEWLINE'''
    p[0] = p[2]


def p_opt_comment(p):
    '''opt_comment : SINGLE_LINE_COMMENT
        | empty'''
    p[0] = p[1]


def p_statement_simple_compound(p):
    '''statement : simple_statement
                 | compound_statement'''
    p[0] = p[1]
```

```python


def p_simple_statement_command(p):
    '''simple_statement : statement_contents newline_opt_comment'''
    p[0] = p[1]


def p_statement_newline(p):
    '''simple_statement : newline_opt_comment'''
    p[0] = Tree()
    p[0].value = "empty"


def p_statement_contents_drive(p):
    '''statement_contents : drive_command'''
    p[0] = p[1]


def p_statement_contents_turn(p):
    '''statement_contents : turn_command'''
    p[0] = p[1]


def p_compound_statement_define(p):
    '''compound_statement : define_command'''
    p[0] = p[1]


def p_compound_statement_repeat_if(p):
    '''compound_statement : repeat_if_command'''
    p[0] = p[1]


def p_compound_statement_repeat_times(p):
    '''compound_statement : repeat_times_command'''
    p[0] = p[1]

```

```
def p_compound_statement_if(p):
    '''compound_statement : if_command'''
    p[0] = p[1]


def p_statement_contents_print(p):
    '''statement_contents : print_command'''
    p[0] = p[1]


def p_statement_contents_assignment(p):
    '''statement_contents : assignment_command'''
    p[0] = p[1]


def p_statement_contents_declaration(p):
    '''statement_contents : declaration_command'''
    p[0] = p[1]


def p_statement_contents_function(p):
    '''statement_contents : function_command'''
    p[0] = p[1]


def p_expression_can_drive(p):
    '''expression : can_drive_expression'''
    p[0] = p[1]


def p_expression_comparison(p):
    '''expression : comparison'''
    p[0] = p[1]


def p_can_drive(p):
```

```
    '''can_drive_expression : CAN_DRIVE drive_direction \
    primary_expression opt_steps'''
    p[0] = makeParseTreeNode([p[0], p[2], p[3]], "can_drive_expression")


def p_comparison_with_operator(p):
    '''comparison : plus_expression comparison_operator plus_expression'''
    p[0] = makeParseTreeNode(p, "comparison")


def p_comparison_plus(p):
    '''comparison : plus_expression'''
    p[0] = p[1]


def p_comparison_operator(p):
    '''comparison_operator : IS
               | IS NOT
               | GT
               | LT
               | GEQ
               | LEQ'''
    if len(p) == 3:  # i.e. token is IS NOT
        p[0] = (p[1][0] + " " + p[2][0], p[1][1] + " " + p[2][1])
    else:  # any other token
        p[0] = p[1]


def p_plus_expression_plus_minus(p):
    '''plus_expression : plus_expression '+' times_expression
          | plus_expression '-' times_expression'''
    p[0] = makeParseTreeNode(p, "plus_expression")


def p_plus_expression_times_expression(p):
    '''plus_expression : times_expression'''
    p[0] = p[1]
```

```
def p_times_expression_times_divide(p):
    '''times_expression : times_expression '*' word_expression
       | times_expression '/' word_expression'''
    p[0] = makeParseTreeNode(p, "times_expression")


def p_times_expression_word_expression(p):
    '''times_expression : word_expression'''
    p[0] = p[1]


def p_word_expression_concat(p):
    '''word_expression : word_expression CONCAT primary_expression'''
    p[0] = makeParseTreeNode(p, "word_expression")


def p_word_expression_primary_expression(p):
    '''word_expression : primary_expression'''
    p[0] = p[1]


def p_primary_expression_parens(p):
    """primary_expression : '(' expression ')'"""
    p[0] = p[2]


def p_primary_expression_token(p):
    '''primary_expression : NUMBER
       | WORD
       | GET_CAR_POSITION
       | ID'''
    p[0] = p[1]


def p_function_command(p):
```

```
    '''function_command : ID opt_parameters'''
    if p[2].value == "empty":
        p[0] = makeParseTreeNode([p[0], p[1]], "function_command")
    else:
        p[0] = makeParseTreeNode(p, "function_command")


def p_opt_parameters(p):
    '''opt_parameters : opt_parameters primary_expression'''
    if p[1].value == "empty":
        p[0] = makeParseTreeNode([p[0], p[2]], "opt_parameters")
    else:
        p[0] = makeParseTreeNode(p, "opt_parameters")


def p_opt_parameters_empty(p):
    '''opt_parameters : empty'''
    p[0] = p[1]


def p_drive_command(p):
    '''drive_command : DRIVE drive_direction plus_expression opt_steps'''
    p[0] = makeParseTreeNode([p[0], p[2], p[3]], "drive_command")


def p_drive_direction(p):
    '''drive_direction : FORWARD
       | BACKWARD'''
    p[0] = p[1]


def p_opt_steps(p):
    '''opt_steps : STEP
       | empty'''
    p[0] = p[1]


```

```python
def p_turn_command(p):
    '''turn_command : TURN turn_direction'''
    p[0] = makeParseTreeNode(p, "turn_command")


def p_turn_direction(p):
    '''turn_direction : LEFT
       | RIGHT'''
    p[0] = p[1]


def p_define_command(p):
    """define_command : DEFINE ID opt_param_list \
    newline_opt_comment statement_block"""
    p[0] = makeParseTreeNode([p[0], p[2], p[3], p[5]], "define_command")


def p_opt_param_list(p):
    '''opt_param_list : USING ID '(' type_enum ')' opt_extra_params'''
    p[0] = makeParseTreeNode(p, "opt_param_list")


def p_opt_param_list_empty(p):
    '''opt_param_list : empty'''
    p[0] = p[1]


def p_opt_extra_params(p):
    '''opt_extra_params : AND ID '(' type_enum ')' opt_extra_params'''
    p[0] = makeParseTreeNode(p, "opt_extra_params")


def p_opt_extra_params_empty(p):
    '''opt_extra_params : empty'''
    p[0] = p[1]
```

```python
def p_type_enum(p):
    '''type_enum : WORD_TYPE
        | NUMBER_TYPE'''
    p[0] = p[1]


def p_repeat_if_command(p):
    """repeat_if_command : REPEAT IF expression newline_opt_comment \
    statement_block"""
    p[0] = makeParseTreeNode(p, "repeat_if_command")


def p_repeat_times_command(p):
    """repeat_times_command : REPEAT plus_expression \
    TIMES newline_opt_comment statement_block"""
    p[0] = makeParseTreeNode(p, "repeat_times_command")


def p_if_command(p):
    """if_command : IF expression newline_opt_comment statement_block \
    opt_else_if opt_else"""
    p[0] = makeParseTreeNode(p, "if_command")


def p_opt_else_if(p):
    """opt_else_if : ELSE_IF expression newline_opt_comment \
    statement_block opt_else_if
        | empty"""

    if len(p) == 2:
        p[0] = p[1]
    else:
        p[0] = makeParseTreeNode(p, "opt_else_if")


def p_opt_else(p):
    """opt_else : ELSE newline_opt_comment statement_block
```

```
           | empty"""

    if len(p) == 2:
        p[0] = p[1]
    else:
        p[0] = makeParseTreeNode(p, "opt_else")


def p_print_command(p):
    """print_command : PRINT word_expression"""
    p[0] = makeParseTreeNode(p, "print")


def p_declaration_command(p):
    """declaration_command : ID IS A type_enum"""
    p[0] = makeParseTreeNode(p, "declaration_command")


def p_assignment_command(p):
    """assignment_command : SET ID TO expression"""
    p[0] = makeParseTreeNode(p, "assignment_command")


def parseString(stringToParse):
    '''Returns the parse tree for the given string'''
    lexer = lex.lex()
    parser = yacc.yacc()
    return parser.parse(stringToParse)

if __name__ == "__main__":
    lexer = lex.lex()
    parser = yacc.yacc()
    inputString = ''
    while True:

        inputString = raw_input('enter expression > ')

```

```
        if inputString == 'exit':
            break

        else:
            try:
                result = parser.parse(inputString)
            except SyntaxError as e:
                print "Error: ", e
            else:
                result.printTree()
                print
                print "errors: ", result.errors
```

10.4 Semantic Analyzer and Symbol Table

Listing 4: SemanticAnalyzer.py

```
# Written by Alex Fields and Colfax Selby

# Scoping done using a universal count, which is a unique number for
# every single scope

from SymbolTable import *
import Parser

table = None
count = 0
function = None
scopeList = [0]
errorList = []
firstPass = True


def analyzeStart(ast):
    # this block for testing purposes
    global table, count, function, scopeList, errorList, firstPass
    table = SymbolLookupTable()
    count = 0
    function = None
```

```python
    scopeList = [0]
    errorList = []
    firstPass = True

    analyze(ast)
    # TODO uncomment after removing testing code
    # global firstPass, count, function, scopeList
    count = 0
    function = None
    firstPass = False
    scopeList = [0]
    analyze(ast)
    return errorList


def analyze(ast):
    '''Traverse the AST and check for semantic errors.'''

    # potential AST values and their associated analysis functions
    # use astAnalzyers.get() instead of a long chain of else-ifs
    astAnalyzers = {
        "assignment_command": assignmentCommandAnalyzer,
        "comparison": comparisonAnalyzer,
        "declaration_command": declarationCommandAnalyzer,
        "define_command": defineCommandAnalyzer,
        "drive_command": driveCommandAnalyzer,
        "empty": emptyAnalyzer,
        "function_command": functionCommandAnalyzer,
        "if_command": ifCommandAnalyzer,
        "opt_else": optElseAnalyzer,
        "opt_else_if": optElseIfAnalyzer,
        "opt_extra_params": optExtraParamsAnalyzer,
        "opt_param_list": optParamListAnalyzer,
        "plus_expression": plusExpressionAnalyzer,
        "print": printAnalyzer,
        "repeat_if_command": repeatIfAnalyzer,
        "repeat_times_command": repeatTimesAnalyzer,
```

```
        "statement_block": statementBlockAnalyzer,
        "statements": statementsAnalyzer,
        "times_expression": timesExpressionAnalyzer,
        "turn_command": turnCommandAnalyzer,
        "word_expression": wordExpressionAnalyzer,
    }

    # Fetch the appropriate analyzer function from astAnalyzers
    # If there is no analyzer for ast.value, then just let the
    # "analyzer" be ast.value
    analyzer = astAnalyzers.get(ast.value, ast.value)

    # If the "anaylzer" is just a string (inherits from basestring)
    if isinstance(analyzer, basestring):
        # this should only be useful for evaluating the type of an expression
        if (ast.type == "WORD"):
            return "word"
        elif (ast.type == "NUMBER"):
            return "number"
        elif (ast.type == "ID"):
            # do existence and scope checking right here
            # return type if passes
            id = ast.value
            idEntry = table.getEntry(SymbolTableEntry(
                id, None, list(scopeList), function, None))
            if idEntry is None:
                # ID does not exist or exists but the scoping is wrong
                # this is to be returned to binaryOperatorAnalyzer
                return "ERROR"
            if function is None and not idEntry.initialized:
                return "ERROR"
            else:
                return idEntry.type

   # if the translator is a real function, then invoke it
    else:
        return analyzer(ast)
```

```python


def statementsAnalyzer(ast):
    if firstPass:
        if (ast.children[0].value == "define_command" or
                ast.children[0].value == "statements"):
            analyze(ast.children[0])
        if (ast.children[1].value == "define_command" or
                ast.children[1].value == "statements"):
            analyze(ast.children[1])
    else:
        analyze(ast.children[0])
        analyze(ast.children[1])


def driveCommandAnalyzer(ast):
    # for "plus_expression"
    result = analyze(ast.children[1])
    if result != "number":
        errorList.append("Error in drive command: \
                need to use valid variable or number")


def turnCommandAnalyzer(ast):
    # nothing to do here
    return


def emptyAnalyzer(ast):
    return []


def comparisonAnalyzer(ast):
    result = binaryOperatorAnalyzer(ast)
    if result == "number":
        return
    elif result == "word":
```

```
        if ast.children[1].type == "IS" or ast.children[1].type == "IS NOT":
            return
        else:
            errorList.append("Error in comparison: \
                     words must be compared using 'is' or 'is not'")
    else:
        errorList.append("Error in comparison: \
                use only words or only numbers; cannot mix both")


def optElseIfAnalyzer(ast):
    # for "expression"
    if ast.children[1].value != "empty":
        analyze(ast.children[1])
     # for "statement_block"
    if ast.children[3].value != "empty":
        analyze(ast.children[3])
     # for "optional_else_if"
    if ast.children[4].value != "empty":
        analyze(ast.children[4])


def optElseAnalyzer(ast):
    # for "statement_block"
    if ast.children[2].value != "empty":
        analyze(ast.children[2])


def ifCommandAnalyzer(ast):
    # for "expression"
    analyze(ast.children[1])
    # for "statement_block"
    analyze(ast.children[3])

    if ast.children[4].value != "empty":
        analyze(ast.children[4])
```

```
    if ast.children[5].value != "empty":
        analyze(ast.children[5])


def repeatTimesAnalyzer(ast):
    # for "plus_expression"
    if analyze(ast.children[1]) != "number":
        errorList.append("Error in repeat loop: \
                need to use valid variable or number")
    # for "statement_block"
    analyze(ast.children[4])


def repeatIfAnalyzer(ast):
    # for "expression"
    analyze(ast.children[2])
    # for "statement_block"
    analyze(ast.children[4])


def declarationCommandAnalyzer(ast):
    # Note ast.children[3].type is word
    table.addEntry(SymbolTableEntry(
        ast.children[0].value,
        ast.children[3].value,
        list(scopeList),
        function,
        None))


def assignmentCommandAnalyzer(ast):
    # check for the existence of ID - child 1
    # and that it can be accessed in this block
    idNoneBool = False
    id = ast.children[1].value
    idEntry = table.getEntry(SymbolTableEntry(
        id,
```

```
            None,
            list(scopeList),
            function,
            None))
    if idEntry is None:
        idNoneBool = True
        # ID does not exist or exists but the scoping is wrong
        errorList.append("Error1 in assignment: \
                variable does not exist or cannot be used here")
    else:
        idEntry.initialized = True

    # do type checking
    # child 3 is an expression - it needs to be evaluated to a type
    child3Evaluation = analyze(ast.children[3])
    if child3Evaluation == "ERROR":
        # type check in expression failed
        errorList.append("Error2 in assignment: \
                use only words or only numbers; cannot mix both")
    else:
        if (not idNoneBool) and idEntry.type != child3Evaluation:
            # type check failed
            errorList.append("Error3 in assignment: \
                    variable and value must have the same type")


def printAnalyzer(ast):
    # for the word or identifier
    if analyze(ast.children[1]) == "ERROR":
        errorList.append("Error in an expression: \
                use only words or only numbers; cannot mix both")
    # check will be done in analyze


def defineCommandAnalyzer(ast):
    global function
    id = ast.children[0].value
```

```
    if scopeList[-1] != 0:
        errorList.append("Error in function creation: \
                functions cannot be created in other \
                functions or a nested block")
    function = id
    if firstPass:
        paramList = []
        if ast.children[1].value != "empty":
            paramList = optParamListAnalyzer(ast.children[1])
            scopeList.pop()
        table.addEntry(SymbolTableEntry(
            id,
            "function",
            list(scopeList),
            None,
            paramList))
        return
    # for "statement_block"
    analyze(ast.children[2])
    function = None


def optParamListAnalyzer(ast):
    scopeList.append(count + 1)
    parameterTypeList = []
    toAdd = SymbolTableEntry(
        ast.children[1].value,
        ast.children[3].value,
        list(scopeList),
        function,
        None)
    toAdd.functionParamBool = True
    table.addEntry(toAdd)
    parameterTypeList.append(ast.children[3].value)
    if ast.children[5].value == "opt_extra_params":
        return optExtraParamsAnalyzer(ast.children[5], parameterTypeList)
    else:
```

```
        return parameterTypeList


def optExtraParamsAnalyzer(ast, parameterTypeList):
    toAdd = SymbolTableEntry(
        ast.children[1].value,
        ast.children[3].value,
        list(scopeList),
        function,
        None)
    toAdd.functionParamBool = True
    table.addEntry(toAdd)
    parameterTypeList.append(ast.children[3].value)
    if ast.children[5].value == "opt_extra_params":
        return optParametersAnalyzer(ast.children[5], parameterTypeList)
    else:
        return list(parameterTypeList)


def statementBlockAnalyzer(ast):
    global count
    count += 1
    scopeList.append(count)
    analyze(ast.children[0])
    scopeList.pop()


def functionCommandAnalyzer(ast):
    # check existence of function ID
    idEntry = table.getEntry(SymbolTableEntry(
        ast.children[0].value,
        "function",
        list(scopeList),
        function,
        None))
    if idEntry is None:
        errorList.append("Error in attempt to use function: \
```

```
                function does not exist")
            return
    elif idEntry.type == "function":
        parameterTypeList = list(idEntry.functionParameterTypes)
    else:
        errorList.append("Error in attempt to use function: \
                function does not exist")
        return
    if len(ast.children) == 2:
        optParametersAnalyzer(ast.children[1], parameterTypeList)


# this is for user-defined function parameters
def optParametersAnalyzer(ast, parameterTypeList):
    if len(ast.children) == 1:
        if (len(parameterTypeList) != 1 or
                analyze(ast.children[0]) != parameterTypeList[0]):
            # Type checking error
            errorList.append("Error in attempt to use function: \
                    wrong type of parameter used")
    else:
        # More parameters left
        if analyze(ast.children[1]) != parameterTypeList.pop():
            # Type checking error
            errorList.append("Error1 in attempt to use function: \
                    wrong type of parameter used")
        else:
            optParametersAnalyzer(ast.children[0], parameterTypeList)


def binaryOperatorAnalyzer(ast):
    result1 = analyze(ast.children[0])
    result3 = analyze(ast.children[2])

    if result1 == "ERROR" or result3 == "ERROR":
        return "ERROR"

```

```
    elif result1 == result3:
        return result1

    elif ast.children[1].type == "CONCAT":
        return "word"

    else:
        return "ERROR"


def plusExpressionAnalyzer(ast):
    return binaryOperatorAnalyzer(ast)


def timesExpressionAnalyzer(ast):
    return binaryOperatorAnalyzer(ast)


def wordExpressionAnalyzer(ast):
    return binaryOperatorAnalyzer(ast)

if __name__ == "__main__":
    inputString = ''
    while True:

        inputString = raw_input('enter expression > ')

        if inputString == 'exit':
            break

        else:
            # first parse the string
            ast = Parser.parseString(inputString)

            ast.printTree()
            print

```

```
            # then check for errors
            if len(ast.errors) > 0:
                print ast.errors
                break

            analyzeStart(ast)
```

10.5 Translator

Listing 5: `Compiler.py`

```
# Written by Sam Kohn and Jeremy Spencer

from Parser import parseString
from SemanticAnalyzer import analyzeStart


def getPythonCode(code):
    '''Convert the given Racecar code into the Python code that will
    run in the GUI.'''

    # first parse the string
    ast = parseString(code)

    # then run the string through the semantic analyzer
    semanticErrors = analyzeStart(ast)

    # then check for errors
    if len(ast.errors) > 0 or len(semanticErrors) > 0:
        return (None, ast.errors + semanticErrors)

    # then generate python code!
    pythonCode = generatePythonCode(ast)

    return (pythonCode, None)


def generatePythonCode(ast):
    '''Traverse the AST and output a string containing the python code
```

```
    to execute in the GUI.'''

    # potential AST values and their associated translation functions
    # use astTranslators.get() instead of a long chain of else-ifs
    astTranslators = {
        "ID": idTranslator,
        "assignment_command": assignmentCommandTranslator,
        "backward": backwardTranslator,
        "backwards": backwardTranslator,
        "comparison": comparisonTranslator,
        "can_drive_expression": canDriveExpressionTranslator,
        "declaration_command": declarationCommandTranslator,
        "define_command": defineCommandTranslator,
        "drive_command": driveCommandTranslator,
        "empty": emptyTranslator,
        "forward": forwardTranslator,
        "forwards": forwardTranslator,
        "function_command": functionCommandTranslator,
        "getCarPosition": getCarPositionTranslator,
        "if_command": ifCommandTranslator,
        "left": leftTranslator,
        "opt_else": optElseTranslator,
        "opt_else_if": optElseIfTranslator,
        "opt_extra_params": optExtraParamsTranslator,
        "opt_param_list": optParamListTranslator,
        "opt_parameters": optParametersTranslator,
        "plus_expression": plusExpressionTranslator,
        "print": printTranslator,
        "repeat_if_command": repeatIfTranslator,
        "repeat_times_command": repeatTimesTranslator,
        "right": rightTranslator,
        "statement_block": statementBlockTranslator,
        "statements": statementsTranslator,
        "times_expression": timesExpressionTranslator,
        "turn_command": turnCommandTranslator,
        "word_expression": wordExpressionTranslator,
    }
```

```
    # "declare" pythonCode since otherwise its first use is inside
    # an if statement
    pythonCode = ""

    # Fetch the appropriate translator function from astTranslators
    # If there is no translator for ast.value then just let the
    # "translator" be ast.value
    translator = astTranslators.get(ast.value, ast.value)

    # If the "translator" is just a string (inherits from basestring),
    # then return that translator
    if isinstance(translator, basestring):
        pythonCode = ast.value

    # if the translator is a real function then invoke it
    else:
        pythonCode = translator(ast)

    return pythonCode


def indentLines(unindentedLines):
    '''Insert 4 spaces (i.e. 1 tab) at the beginning of every line'''

    splitCode = unindentedLines.splitlines(True)

    pythonCode = "    " + "    ".join(splitCode)
    return pythonCode


def emptyTranslator(ast):
    return ""


def statementsTranslator(ast):
    pythonCode = generatePythonCode(ast.children[0])
```

```
    pythonCode += generatePythonCode(ast.children[1])
    return pythonCode


def driveCommandTranslator(ast):
    # drive numSteps direction steps -->
    # translate_car(numSteps, direction)\n
    pythonCode = "translate_car("
    pythonCode += generatePythonCode(ast.children[1])
    pythonCode += ", " + generatePythonCode(ast.children[0])
    pythonCode += ")\n"
    return pythonCode


def forwardTranslator(ast):
    pythonCode = "CarDirection.FORWARDS"
    return pythonCode


def backwardTranslator(ast):
    pythonCode = "CarDirection.BACKWARDS"
    return pythonCode


def turnCommandTranslator(ast):
    pythonCode = "rotate_car("
    pythonCode += generatePythonCode(ast.children[1])
    pythonCode += ")\n"
    return pythonCode


def comparisonTranslator(ast):
    pythonCode = generatePythonCode(ast.children[0])
    if ast.children[1].value == "is not":
        pythonCode += " != "
        pythonCode += ast.children[2].value
    elif ast.children[1].value == "is":
```

```
            pythonCode += " == "
            pythonCode += generatePythonCode(ast.children[2])
        else:
            pythonCode += " " + generatePythonCode(ast.children[1])
            pythonCode += " " + generatePythonCode(ast.children[2])

        return pythonCode


def optElseIfTranslator(ast):
    pythonCode = "elif "
    pythonCode += generatePythonCode(ast.children[1]) + ":\n"
    pythonCode += generatePythonCode(ast.children[3])

    if ast.children[4].value != "empty":
        pythonCode += generatePythonCode(ast.children[4])

    return pythonCode


def optElseTranslator(ast):
    pythonCode = "else:\n"
    prelimPythonCode = generatePythonCode(ast.children[2])
    pythonCode += generatePythonCode(ast.children[2])
    return pythonCode


def ifCommandTranslator(ast):
    pythonCode = "if " + generatePythonCode(ast.children[1]) + ":\n"
    pythonCode += generatePythonCode(ast.children[3])

    if ast.children[4].value != "empty":
        pythonCode += generatePythonCode(ast.children[4])

    if ast.children[5].value != "empty":
        pythonCode += generatePythonCode(ast.children[5])
    return pythonCode
```

```
def leftTranslator(ast):
    pythonCode = "WheelDirection.LEFT"
    return pythonCode


def rightTranslator(ast):
    pythonCode = "WheelDirection.RIGHT"
    return pythonCode


def repeatTimesTranslator(ast):
    if ast.children[2].value == "times":
        pythonCode = "for x in range(" + ast.children[1].value + "):\n"
        pythonCode += generatePythonCode(ast.children[4])
    return pythonCode


def repeatIfTranslator(ast):
    pythonCode = "while " + generatePythonCode(ast.children[2]) + ":\n"
    pythonCode += generatePythonCode(ast.children[4])
    return pythonCode


def declarationCommandTranslator(ast):
    # id is a whatever -->
    # id = None
    pythonCode = generatePythonCode(ast.children[0])
    pythonCode += " = None\n"
    return pythonCode


def idTranslator(ast):
    pythonCode = ast.value
    return pythonCode

```

```
def assignmentCommandTranslator(ast):
    pythonCode = generatePythonCode(ast.children[1])
    pythonCode += " = "
    pythonCode += generatePythonCode(ast.children[3])
    pythonCode += "\n"
    return pythonCode


def printTranslator(ast):
    pythonCode = "print_to_console("
    pythonCode += generatePythonCode(ast.children[1])
    pythonCode += ")\n"
    return pythonCode


def defineCommandTranslator(ast):
    pythonCode = "def "
    pythonCode += generatePythonCode(ast.children[0])
    pythonCode += "("
    if ast.children[1].value == "opt_param_list":
        pythonCode += generatePythonCode(ast.children[1])
    pythonCode += "):\n"
    pythonCode += generatePythonCode(ast.children[2])
    return pythonCode


def optParamListTranslator(ast):
    pythonCode = generatePythonCode(ast.children[1])
    if ast.children[5].value == "opt_extra_params":
        pythonCode += generatePythonCode(ast.children[5])
    return pythonCode


def optExtraParamsTranslator(ast):
    pythonCode = ", "
    pythonCode += generatePythonCode(ast.children[1])
```

```
    if ast.children[5].value == "opt_extra_params":
        pythonCode += generatePythonCode(ast.children[5])
    return pythonCode


def statementBlockTranslator(ast):
    prelimPythonCode = generatePythonCode(ast.children[0])

    pythonCode = indentLines(prelimPythonCode)

    return pythonCode


def functionCommandTranslator(ast):
    pythonCode = generatePythonCode(ast.children[0])
    pythonCode += "("
    if len(ast.children) > 1:
        pythonCode += generatePythonCode(ast.children[1])
    pythonCode += ")\n"
    return pythonCode


def optParametersTranslator(ast):
    numChildren = len(ast.children)
    if numChildren > 0:
        pythonCode = generatePythonCode(ast.children[0])
        if numChildren == 2:
            pythonCode += ", "
            pythonCode += generatePythonCode(ast.children[1])
        return pythonCode
    else:
        return ""


def binaryOperatorTranslator(ast):
    pythonCode = "(("
    pythonCode += generatePythonCode(ast.children[0])
```

```
    pythonCode += ") "
    pythonCode += generatePythonCode(ast.children[1])
    pythonCode += " ("
    pythonCode += generatePythonCode(ast.children[2])
    pythonCode += "))"
    return pythonCode


def plusExpressionTranslator(ast):
    return binaryOperatorTranslator(ast)


def timesExpressionTranslator(ast):
    return binaryOperatorTranslator(ast)


def wordExpressionTranslator(ast):
    pythonCode = "(str("
    pythonCode += generatePythonCode(ast.children[0])
    pythonCode += ") + str("
    pythonCode += generatePythonCode(ast.children[2])
    pythonCode += "))"
    return pythonCode


def getCarPositionTranslator(ast):
    return "getCurrentPosition()"


def canDriveExpressionTranslator(ast):
    pythonCode = "can_move("
    pythonCode += generatePythonCode(ast.children[1])
    pythonCode += ", " + generatePythonCode(ast.children[0])
    pythonCode += ")"
    return pythonCode

if __name__ == "__main__":
```

```
    inputString = ''
    while True:

        inputString = raw_input('enter expression > ')

        if inputString == 'exit':
            break

        else:
            print getPythonCode(inputString)
```