Racecar Language

Lesson Plan

Team 19

Samuel Kohn	Project Manager
Alexander Fields	Language Guru
Colfax Selby	Verification and Validation
Jeremy Spencer	System Architect
Mason Silber	System Integrator

Due: 27 March 2013
COMS W4115: Programming Languages and Translators
Professor: Alfred Aho
Mentor: Melanie Kambadur

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 big GO button is what you click when you want to run the program sitting in the program window. The STOP 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 `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.

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. This way, you 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.

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 explode. See if the students can figure out how many steps it takes to just barely reach the edge without losing the car.

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 comment beginning (`:)`) 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 page.

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.

Lesson 3: Turning the Wheel

Lesson Summary:

In this lesson, you will learn how to turn the wheels on the car. The car turns just like a real car—by turning its wheels and then moving forwards. The keywords covered are `steer`, `left`, and `right`.

The Program:

```
:-( This program drives the car forwards
    for a bit, and then turns the car to the
    left by turning the wheels and then moving
    one additional step.
:-)

drive forward 10 steps

:) Here we steer to the left but do not move the car
steer left

:) And now we move the car with the wheels turned
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 its wheels to the left (the other possibility is `steer right`). The wheels automatically turn to point 45 degrees to the left. The last line takes one step forwards with the wheels turned, turning the car 45 degrees and moving it diagonally forwards and to the left.

Further advice:

Your students might have played video games in which “turning” means that the entire object turns in place, but this is not how cars work in real life and in Racecar. Similarly, when the car’s wheels are turned in one direction, they stay that way until you tell the wheels to straighten out (covered in the next lesson). For each step taken while the car's wheels are turned, the car will turn 45 degrees in that direction and then move one step. For example, consider the following program:

```
:) This program will turn the car halfway around

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

steer right

:-( Each step results in another 45 degree turn
    so 4 * 45 = 180 degrees
:-)
drive forward 4 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's wheels turn to the right. Each step moves the car at a 45 degree angle to the direction it is facing. Moving the car 4 steps with the wheels turned will turn the car 180 degrees (and move it around a bit, too), and the car will be facing the opposite of its original direction, although it will not be located in the same place it started. Again, this is similar to how in real life you cannot just turn a car around 180 degrees in place. Note that after 8 steps with the wheels turned, the car will return to its original position and direction after driving in a complete circle (really an octagon, but that's not so important!).

Lesson 4: A Complete Turn

Lesson Summary:

In this lesson, you will learn how to perform a complete turn and then resume driving in a straight line.

The Program:

```
:-( We turn the car and then straighten out the wheels
    so that the car will resume driving in a straight line
:-)

turn wheels left
```

```
:) 1 step will angle the car at 45 degrees
drive forward 1 step

:) Straighten the wheels and try driving straight forwards
turn wheels straight
drive forward 10 steps
```

In this program we learn a new direction to turn the wheels: `straight`. This command is necessary because (like a real car) the car "remembers" which direction its wheels are pointing. Consequently, to drive in a straight line again, we must straighten the wheels, done on line 3. The last line tells the car to drive forward 10 steps in the new direction (now considered "straight").

Further advice:

In Racecar, the only options to go after `turn wheels` are `left`, `straight`, and `right`. It is worth noting that if the wheels are already pointed left, `turn wheels right` will make them turn all the way to the right—they will not be straight. Similarly, if the wheels are already pointed left, then `turn wheels left` will not do anything to the car, but it will print a notice to the screen so that you are aware that the command did not do anything.

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 `word` you give it on the computer screen.

The Program:

```
:-( In this program we check if the wheels are
    already turned left, and if not, we turn
    them left. If they are turned left, we print
    out "Wheels are turned left" to the computer screen
:-)

:) Checks if the wheels are turned left
if wheelDirection is "left"
{
    print "Wheels are turned left"
}
else :) If wheels are not turned, it turns them left
{
    steer 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 wheels are turned left by comparing `wheelDirection` to the word `"left"`. `wheelDirection` is automatically replaced by a word containing the current direction that the wheels are pointing. If they are, a message is printed that says "Wheels are turned left" using the `print` function. If they are not, 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 car's wheels are turned 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 `else if` statements after an if statement, like the following:

```
:) Print out a message describing the wheels' direction
if wheelDirection is "left"
```

```
{
    print "Wheel direction is left"
}
else if wheelDirection is "right"
{
    print "Wheel direction is right"
}
else :) the wheels must be straight
{
    print "Wheels are straight"
}
```

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.

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’
    to be 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’
    to be 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`, `steer`, `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!"
}
```

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.

Lesson 7: Loops Part 1

The Program

```
:-( This program creates a variable ‘myCounter’
     to be 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 while loop is similar to a conditional statement in that it checks an expression to see if it is either `true` or `false`, and 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 boolean value of the 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.

Further advice

This 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.

Lesson 8: Loops Part 2

The Program

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

myCounter is a number
set myCounter to 10

:) Instead of driving 10 steps, drive 1 step 10 times!
repeat myCounter times
{
    drive forward 1 step
}
```

In this program, we explore repeat-times loops. Similar to a repeat-if loop, a for loop repeats the enclosed block of code multiple times. Rather than depending on a `true` or `false` expression to decide how 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 a number of times you would like the block to run. See here:

```
repeat 5 times
{
     drive forward 1 step
}
```

Lesson 9: Subroutines

The Program

```
:-( In this program, we create and use
     a subroutine.
:-)

:) Here is the creation of our subroutine.
define turnLeft
{
turn wheels left
drive forward 2 steps
turn wheels straight
}

drive 10 steps
:-(  Here is our first time we actually use or "call" our
     subroutine
:-)
turnLeft
drive 10 steps
```

```
turnLeft
drive 10 steps
turnLeft
drive 10 steps
turnLeft
```

In this program, we write subroutines. Subroutines are like shortcuts or nicknames: they allow us to write code only once for something we will use multiple times in our program. In this program, we make a subroutine called `turnLeft`, which turns the wheels, drives the car along a curve, and straightens the wheels again. Our `turnLeft` subroutine will turn the car 90 degrees to the left (note, this is only true because we drive 2 steps while the wheels are turned, since each step results in a 45 degree rotation). At the end of our program, we drive the car forwards and turn left four times using our subroutine `turnLeft`. Notice how efficient it was to code the program using our subroutine as opposed to writing out all the lines of code required for turning whenever we wanted to turn. The above program will result in the car returning to its original position after driving in a square with "rounded" corners.

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 turn," 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.

Lesson 10: Variables in Subroutines

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 turnLeft using numSteps (number)
{
    turn wheels left
    :) Here is where the variable will be used
    drive numSteps steps
    turn wheels straight
}

:-(  Now we use the subroutine, specifying the variable
     which gets plugged into the code we wrote above
     for our subroutine
:-)
turnLeft 8
drive forward 10 steps
turnLeft 4
```

Like the last program, we write a subroutine for turning left called `turnLeft`, but unlike the last program we now are using a new variable in our subroutine called `numSteps`. 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 function that they are defined with. Now we can have the car turn any number of steps, which we will decide exactly how many when we use the subroutine in our code. In this program, after we make the subroutine, we use it, turning 8 steps (which is a full circle given that each step while the wheels are turned results in a 45 degree rotation). We then have the car drive forward a bit and turn only 4 steps. It is much simpler to put a different number here than to copy the `turnLeft` subroutine from Lesson 9 each time you want to turn a new number of steps!

Further advice

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

```
define turnLeftThenDriveStraight using numStepsTurn (number) and
numStepsDrive (number)
{
turn wheels left
drive numSteps steps
turn wheels straight
drive numStepsDrive
}
```

How to call the function:

```
turnLeftThenDriveStraight 5 10
```