MRRoboto:

Macro Record Robot Language
Final Report

Authors:

Adam Marczyk (alm2126@columbia.edu)
Hema Krishnan (hk2230@columbia.edu)
Jason Kopylec (jkk2106@columbia.edu)

Table of Contents

Whitepaper

Macro Record Robot Language (MRRoboto) places in the hands of the programmer a powerful tool for controlling GUI based operating systems. By simulating the actions of a user, one can test interactive applications and automate repetitive tasks with ease. MRRoboto provides a simple script-style interface, while leveraging the strengths of the Java programming environment. This paper proposes a language that:

1. Contains a simple interface where users can easily create scripts that mimic keyboard and mouse input to automate repetitive tasks
2. Serves to model virtual user input using common computer vision, artificial intelligence and machine learning techniques
3. Interfaces with Java to utilize the rich language libraries without restricting the applications to be Java-specific.

Background

The first programs to allow real-time user interaction came shortly after the invention of the Video Display Terminal in the mid-1960's which incorporated a television style monitor with an electric typewriter [Bellis]. In the years following, real-time interactive operating systems were being developed, culminating in the introduction of UNIX in 1970 [Milo]. Along with the ability to interact with the user via the command line came tools for automating user input. Shell scripting and IO redirection provide the groundwork for application testing and automation of repetitive tasks and have become invaluable programmer tools.

The Graphical User Interface (GUI) has become a staple of the modern, mainstream operating system since its inception in the 1979 Macintosh [Tuck]. With the proliferation of personal computing and the World Wide Web, user interaction based on keyboard and mouse (or similar pointing devices) input is a main function of modern operating systems. Unfortunately, there is a lack of simple scripting tools for automating this type of interaction. There is no widely used batch/script type language to easily automate user functions as with command line operating systems. Usually knowledge of large and complex operating system API is required to manipulate individual screen elements which are bulky and counterintuitive. User interaction is much easier when thought of in terms of mouse and keyboard commands.

Java provides an interface, albeit a clumsy one, for modeling and performing user input in a more natural way. By streamlining the Java model and providing a simple script-style interface, programmers would have an ideal tool for testing their interactive graphical applications and be able to automate tasks in a way analogous to UNIX shell scripts or MSDOS batch files.

Existing Technologies

Current programming languages, applications and operating systems do provide mechanisms for manipulating the user desktop and applications, but each comes with inherent restrictions that MRRoboto serves to correct. To outline a few examples:

Microsoft Windows API

Microsoft has built into its latest generation operating systems (Win 2k, XP) a multitude of API to control and handle every aspect of the kernel and windowing system. Windows allows scripts and applications to manipulate individual components of the user desktop.

The first problem with this API is its sheer size. The Windows API supports literally thousands of procedures to control the desktop environment. It is quite easy to get overwhelmed and lost in the Microsoft Developers Network website which holds the OS and language specifications [MSDN.com].

Secondly, these APIs do not reflect accurately the actions of interactive users. For example, to create a script that closes the currently open window, the script would require you to know the name of that window, get a handle to it and then evoke a method that closes the window and kills the process. What a user would do is simply move their mouse over the "X" in the corner and click.

Traditional Batch Files

The traditional batch file is comprised of lists of commands that can are executed sequentially (with some limited flow control) as if the user had typed them one after another. This is powerful because it is easy to automate repetitive tasks and the script can use the same language as the user to communicate with the operating system.

The problem with traditional batch files is that they do not support multiple windows or mouse actions. The only way that batch files can send input to the applications they execute is through the command line or by redirecting data from standard input. This limits mouse-driven applications or programs with specialized, non-sequential input controls, such as web pages.

JUNIT

As large scale open source project, the sole focus of JUNIT is to automate application and code testing. In particular, the module JFCUnit was built to test graphical applications by providing input from a virtual test user. It serves to "start the application and then interact with it, typing in some text and maybe pressing some buttons." [Hammell]

JUNIT supplies the Java applications programmer with a plethora of tools for running tests on code, but this is also its limitation. It was developed to test only Java applications, and runs in the same thread as the application, so if the application fails, so does the user interactivity [Clark]. This does not at all supply portability to general problems of testing application, modeling user artificial intelligence, or automating regular windowing tasks.

Stand Alone Applications

Searching for "Windows Macro" within *Download.com* displays a number of applications such as Macro Express, Workspace Macro and Phantom Sidekick to name a few. The main goal of these applications is to automate repetitive tasks. Often this is done by pressing a "record" key and then typing and clicking the procedure and pressing "stop" when completed.

These applications have their share of problems though. The first is the representation of the macro once it has been recorded. Some applications represent the macro in terms of Windows API, which suffers from the same hindrances of the API itself, namely its huge size and not modeling user input as a sequence of keystrokes and mouse actions.

Applications that do not translate into Windows API either compile directly into machine code or a proprietary intermediary. This is problematic because if a small part of a large repetitive task changes, then the entire macro must be recorded again from the beginning.

In addition to these limitations, stand-alone apps are more difficult, if not impossible to utilize as virtual user interfaces to generate input on the fly, or incorporate into larger Java applications.

Java.awt.Robot

Java API provides a class for manipulating the user interface in a much more user oriented fashion. It includes methods for moving the mouse to a specific coordinate on the screen, mouse clicks, keyboard input and some rudimentary methods for retrieving the screen contents ["Robot"]. The java.awt.Robot class can work either within or outside the scope of java applications. The Java Robot requires no user knowledge of the underlying operating system specifics or API.

The downside to the Java implementation is that is a bit clumsy. There are a number of places where in addition to needing to know about Object-Oriented and basic Java programming, there are special constants and API necessary for forming the expected output results. For example, each keyboard character must be pressed and released in separate calls and is indexed by a special constant [Baldwin]. This is quite a hassle if you want to automatically type the text "http://www.google.com" into a web browser.

Putting the Pieces Together

Macro Record Robot Language (MRRoboto) is a programming language that leverages the assets from the following sources:

1. The ease of making batch files
2. The power of Java (and particularly the Java.awt.Robot class)
3. User ability to interact with environment via keyboard, mouse and monitor

Each of these tools on its own has limitations, but by modeling a language that puts them together, a complete array of applications both trivial and complex can be developed with flexibility and ease. The audience for this language spans from the casual user who wants to check email faster without a lot of typing and clicking to the machine learning or computer vision researcher who wants to model user interaction with Windows to programmers who want automate the testing of their GUIs.

What it Looks Like – Example 1

A simple application that automates a repeated user function would typically look like a number of mouse clicks and keystrokes. For example, let's say we wanted to write a script that checks Yahoo! email. Code would probably look something like the following:

CheckYahooMail.mrr

```
moveAndClick(20, 755)         **click Internet Explorer icon
wait(10000)              **wait 10,000ms (10 sec) for the page to load
type("username")         **type Yahoo mail username
type("|TAB| mypassword")**Hit tab-key and my Yahoo password
type("|TAB||ENTER|")     **Hit tab-key again and hit enter
```

After compiling and running the code, one should be taken directly to their Yahoo! Mail inbox.

Artificial Intelligence – Example 2

Applications that act as intelligent agents controlling a desktop can become quite complex, but we can also show a simple application that give their human counterparts some real competition.

As an example, lets create a script that plays a simple game that imitates a shooting range (see Fig. 1) [“Shooting”]. The user scores a point for each bull’s-eye that is “shot” by pressing the mouse button on it. A point is deducted for every misfire. The difficult part of the game is that it contains virtual human movement, so the gun sight shifts back and forth and shooting causes recoil that loses aim as a human would. The user must try to shoot as fast as possible once the crosshairs are over the bull’s-eye.

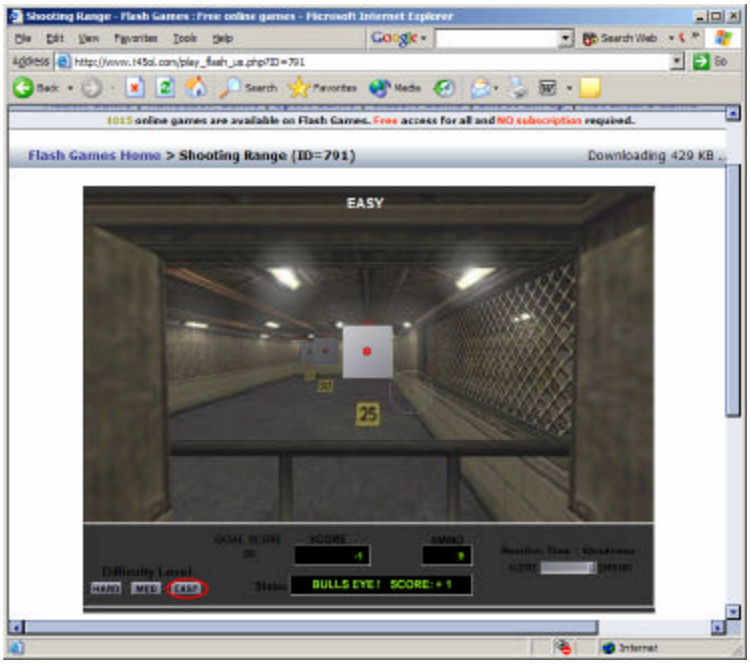


Fig. 1 Shooting Range Game.

We want to generate a program that plays the game with a decent level of skill which can be done with the following MRRoboto program:

ShooterGame.mrr

```
**Assume that the shooter game window has been opened and started
int numShots
numShots=22 **User gets 22 shots at the target

**Try to click on the bull’s-eye for each shot
int i
for i, 1 to numShots
      click(350, 500)    ** click in the middle of the bullseye
      delay(2000)        **2sec delay between shots to allow for recoil
end
```

We see here the program executes the same actions that an ideal user playing the game would. Someone new to the game could definitely increase their high scores by letting MRRoboto play for them.

Language Tools

To make the language more expressive than simply "click here, click here, type here, etc" a number of programming language primitives and structures will be incorporated to facilitate solving a broad array of programming problems. These include:

Data Types

Support for data types such as *integers* and *strings* will be included as basic programming tools. In addition, there will be primitives for dealing with display elements, such as retrieving a bitmap of the screen which can be compared to other bitmaps. This provides the groundwork for designing reactive components and run-time error checking.

Flow Control

Flow control structures such as *for-loops* and *if-then* statements will be in place to work with the primitive data types for conditional flow control. Procedures and functions with syntax similar to java will also be supported to facilitate program segmentation.

Screen Output

Taking feedback from the display opens up a complex set of computer vision issues that may not be solved in a project this size, but the aim is to incorporate some primitive methods for retrieving pixel colors to illustrate the possibilities of using display feedback to drive automation and/or artificial intelligence applications.

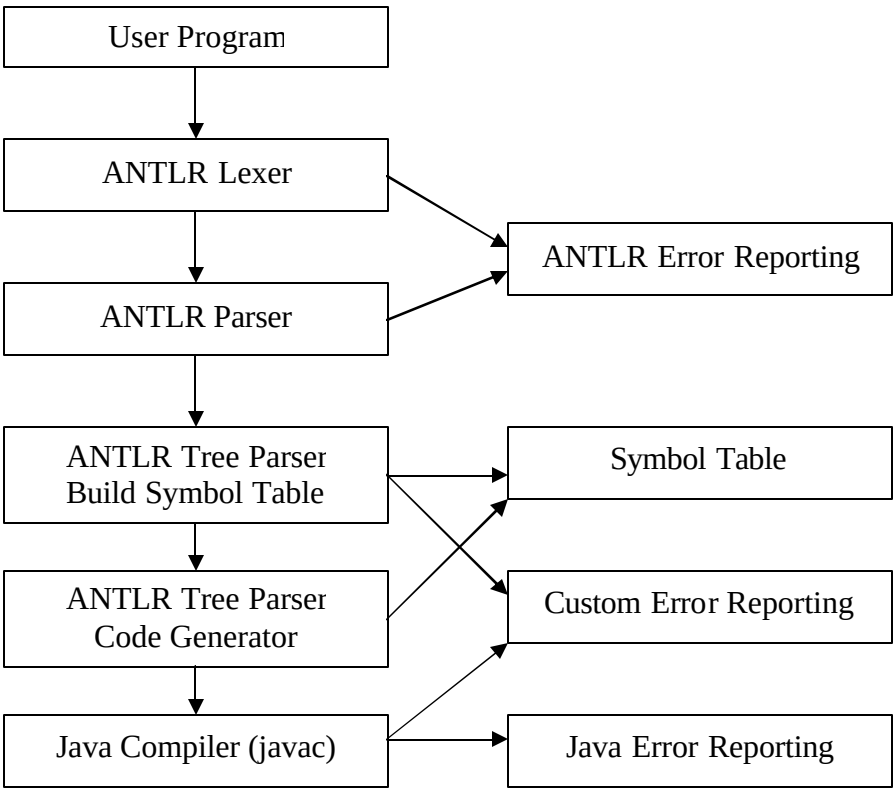
Embedded Java

To utilize the far reaching power of Java without muddying the syntax of the language, native Java code can be placed within brackets anywhere in the code. This opens the door for advanced programmers to utilize this simple scripting language as a core part of larger Java development projects.

Compiler

The MRRoboto programs are compiled into native Java code that can then be compiled and run as normal Java programs. Many similar languages are interpreted, but by compiling into Java (and then into Java byte code) the scripts can be fully incorporated seamlessly into larger Java programs. The MRRoboto program is parsed and translated by a custom procedure developed using ANTLR and Java. See Figure 2 for the compilation process.

Fig. 2 MRRoboto Compilation Process



ANTLR Lexer/Parser

Providing custom programming tools requires defining a syntactic grammar and semantic rules. ANTLR provides the tools to easily create lexical analyzers, parsers and other compiler components based on defined grammars [Parr]. MRRoboto will use ANTLR to develop the front-end syntax checking required prior to the conversion to native Java. ANTLR will also be used to report custom errors back to the user about syntactic or semantic errors. Any native Java in the source code is ignored.

Translation to Native Java

Once the code is deemed "valid" by the ANTLR parser, a module will create a Java template class that initializes a Java.awt.Robot instance. The MRRoboto commands are translated into java code and procedures statement-by-statement. This class can then be compiled into a working java program. Any errors in Java from the source code would be indicated during compilation.

Summary

MRRoboto takes advantages of a script style interface while maintaining the full features of a large scale programming language like Java to provide what other tools have fallen short of in terms of providing a virtual user interface. The language can be used to easily model and solve a multitude of problems from process automation to artificial intelligence.

Language Tutorial

Macro Record Robot is a script-like language that can allow for the automation of repetitive tasks. It is fairly easy to write a program using MRRoboto. Before we start to show you, here's how you use the compiler and execute a program. Enter the following at any command prompt.

> java MRRoboto <filename.mrr>

The filename has to have the .mrr extension in order for the compiler to work. This command translates the MRRoboto code into java bytecode and outputs the file into one of the same name. All that is left to do is to compile this code using the java compiler and then running the executable.

> javac <filename.java>
> java <filename>

Developing with MRRoboto

Now that you are familiar with how the compiler works, let us progress to learning how to write a file using the MRRoboto language.

Hello World

The following is the canonical first program and prints "Hello World" to the console.

helloworld.mrr

```
** This is a simple MRRoboto Program

print("Hello World!")
```

Checking Yahoo Mail

If you had a Yahoo! Mail account and wanted to automate the process of checking your email, you can create a file to do so using any available text editor. In fact, let us put the following in a file called CheckMail.mrr and see how the process works. You may have to modify the screen coordinates to match your screen resolution and icon placement, but this can be done easily by using the RobotoXY tool (see Appendix D).

CheckMail.mrr

```
** Sample program to check yahoo mail
moveAndDoubleClick(31, 240)   **click Internet Explorer icon
wait(10000)                   **wait 10 secs for page to load
moveAndClick(43, 316)         **click Yahoo! Mail link
wait(10000)                   **wait 10 secs for the page to load
type(" your username")        **type Yahoo mail username
type("|TAB|  your password")  **Hit tab-key and my Yahoo password
```

```
type("|ENTER|")                    **Hit tab-key again and hit enter
```

The first line is your basic comment to indicate what the program below does. As you may have noticed, comments start with two asterix symbols and they are single line comments only. Lines 2-8 illustrate the usage of library functions that perform the actions necessary to start a browser window and log into Yahoo! Mail. The numbers that have been passed into the moveAndDoubleClick() and moveAndClick() methods are the coordinates of the Internet Explorer icon on the desktop and the link to Yahoo! Mail in the browser respectively.

These coordinates were determined by using a program called RobotoXY, a script that returns the position of the mouse, at any given time, on the screen. Using these coordinates you can create any number of files that will open or take care of some of the most mundane tasks.

If you follow the instructions above regarding how to use to the compiler, then the .mrr file above will be translated into one called CheckMail.java. Run the java compiler on this file and execute the resulting program.

The following example extends our first Hello World program by showing off the keyboard control capabilities by opening a Notepad window and typing “Hello World” into it ten times. It will show you how loop constructs and functions work in MRRoboto, along with an important library function. The language is not hard to understand even if you are a novice at programming. It provides you with a better grasp on the constructs used in higher level languages. Try entering the following in a file named HelloWorld.mrr.

HelloWorld.mrr

```
1.    **Opens a text editor and prints Hello, World for a set number of times
2.    openNotepad()                              **procedure call
3.    procedure openNotepad()                    **procedure declaration
4.        exec("notepad.exe")                    **library function used to open notepad
5.        wait(1000)                             **wait for 1 sec
6.        int i
8.        for i, i to 20 step 2
9.            type("Hello, World")               **types Hello, World in open file
10.           type("|ENTER|")                    **Hit the [Enter] key
11.       end                                    **end for loop
12.   end                                        **end procedure
```

Line 1 marks a comment, as seen before. Line 2 indicates how a procedure should be called. This call can be made either before or after the procedure declaration. Lines 3-12 illustrates a procedure declaration. Line 4 contains a library call that should be quite useful. The exec(cmd) method starts any program that you like without having to start command prompt first. In this case, Notepad is the program that the exec function starts. Lines 6-7 indicate the necessary variable declarations for the ‘for loop’.

Unlike Java, variable declarations cannot be made as one writes the program. These variables need to be declared and initialized separately. Lines 8-10 indicate the workings of the ‘for loop’. If you look at the for statement, you will notice two keywords – to and step. The word ‘to’ indicates the range that needs to be covered and the word ‘step’ indicates how much the variable i will be incremented. This word is optional. If it is not included, then the default increment is 1. This can also be implemented using a while loop. The reserved word ‘end’ is required to finish both the loop construct and the procedure declaration. For additional information on loops and functions, see the Language Reference Manual.

Language Reference Manual

The Macro Record Robot Language (MRRoboto) places in the hands of the programmer a powerful tool for controlling GUI-based operating systems. By simulating the actions of a user, one can test interactive applications, automate repetitive tasks with ease, and set one's computer to perform tasks at scheduled times when no user can be at the keyboard. MRRoboto provides a simple script-style API to mimic keyboard and mouse input, and utilizes Java's rich language libraries without requiring specific skill in programming Java syntax.

This manual outlines the language syntax and semantics for creating MRRoboto programs and compilers. Wherever possible, context-free grammars have been included to precisely define the syntactic constructs. The overall flavor of the language is similar to scripting languages such as Unix shell scripts, with additional power and flexibility added through the use of Java-like control constructs.

File Name Conventions

MRRoboto programs should be named with the extension *.mmr*. The file name will become the name of the Java class that is created, so it should take the form of an *identifier*. For example, the file *TestProgram.mmr* compiles into a Java file *TestProgram.java.*

The Main Block

A program in the MRRoboto language has one main block of code, consisting of multiple statements each on its own line, that is run each time that program runs and that is the first thing to be executed each time that program runs (analogous to the main() function in a program in C or Java). The main block is followed by one blank line and then one or more function definitions that may be invoked by the main block during its execution.

Main Block Grammar

```
program
      : main_block (procedure)* EOF!

main_block
      : (stmt)+
```

Statements

A block of code in the MRRoboto language is made up of one or more statements, each followed by a new line character. A statement may be any of the following: a comment, native java code, a variable declaration or assignment, a conditional statement, a loop, or an expression incorporating one or more operators.

Expression Grammar

```
stmt
      : JAVA
      | COMMENT
      | var_declaration
      | assignment
      | procedure_call
      | conditional
      | for_loop
      | while_loop
      | directive
      | NEW_LINE!
```

Comments

Comments are statements inserted by the programmer to more clearly explain the operation of the program. They are considered part of the program source code and are maintained in the generated java code to aid in ease of understanding the compiled java code. Comments are marked by a double asterisk ("**") at the beginning of a line and extend through the end of that line. Multiple-line comments may be created by starting multiple successive lines with this delimiter.

Comment Grammar

```
comment
      : "**" (.)*
```

Identifiers

Identifiers are defined as the names of functions, variables, or class files. In this language, an identifier must start with an alphabetic character followed by any number (up to Java's maximum) of letters, digits or the underscore character, '_'. Reserved words are illegal as identifiers.

Examples of Legal Identifiers: `abc123, x, i_count, tmp`
Examples of Illegal Identifiers: `abc&c, _ac2, 1stP, end`

Identifier Grammar

```
identifier
     : letter (alphanumeric | "_")*

alphanumeric
     : letter | digit

letter
     : [A-Za-z]

digit
     : [0-9]
```

Reserved Words

The following are reserved as keywords in the language. They may not be used as identifiers. Keywords in Java are also illegal identifiers, but they are not listed here.

MRRoboto Reserved Words

```
mouseMove                 click
moveAndClick              rightClick
moveAndRightClick         wait
setWait                   type
press                     hold
release                   releaseAll
exec                      print
printInt                  substring
length                    pc
to                        step
end                       procedure
error                     mrRobotoLibrary
break                     continue
return                    for
while
```

These reserved words represent either built-in library functions, data types or control flow keywords; their exact functions will be discussed in detail below.

Variables

Variables are mutable, dynamically valued blocks of memory that store values used by the program during its execution. All variables must be declared and assigned a value before being used; variable names are identifiers and must be distinct. MRRoboto

supports two built-in data types for variables: *string*, and *int*. Each of these is described in detail below.

String Literals

A string is a sequence of printing and non-printing characters. Strings in the MRRoboto language are the same as those in Java, with the addition of escape codes for keys that do not produce printing characters. Escape sequences are indicated by the key name (see below chart), surrounded by backslashes. Strings are delimited with double quotes. To include literal double quotes or backslashes, pair them, as in Java.

Example Strings: `"abc", "How are you today?", "|TAB|", "\\"`

Key Escape Codes

`|F1| - |F12|`	F1 – F12 keys
`|ALT|`	[ALT]
`|CTRL|`	[CTRL]
`|CAPS_LOCK|`	[CAPS LOCK]
`|DELETE|`	[DELETE]
`|BACKSPACE|`	[BACKSPACE]
`|TAB|`	[TAB]
`|ENTER|`	[RETURN]/[ENTER]
`|DOWN|`	Down Arrow Key
`|UP|`	Up Arrow Key
`|LEFT|`	Left Arrow Key
`|RIGHT|`	Right Arrow Key
`|ESCAPE|`	[ESC]
`|HOME|`	[HOME]
`|END|`	[END]
`|INSERT|`	[INSERT]
`|PRINTSCREEN|`	[PRINTSCREEN]
`|SHIFT|`	[SHIFT]
`|PGDOWN|`	[PAGE DOWN]
`|PGUP|`	[PAGE UP]
`|NUM_LOCK|`	[NUM LOCK]
`|DOUBLE_QUOTE|`	[“]

String Grammar

```
string
     : """ (.)* """
```

Integers

An integer is either 0 or a positive whole number composed of a sequence of digits. The minimum and maximum values are machine-specific. MRRoboto does not distinguish between signed and unsigned integers and does not allow for negative values.

Example Integers: 0, 1, 2, 13, 725

Integer Grammar

```
integer
      : [0-9]+
```

Declaration

Variables must be declared before use in order to determine their type. A declaration statement consists of the type being declared, followed by the variable name identifier. Variables are declared one per line. String and integer variables are automatically initialized to “”, and 0, respectively.

Declaration Grammar

```
declaration
      : var_type identifier

var_type
      : ("int" | "string")
```

Assignment

Once a variable has been declared, it must be assigned a value before it can be used. An assignment statement consists of the name of the variable, followed by an equals-sign character, followed by the value the programmer wishes to assign to it (which must be of the same type as the variable was initially declared to have). The value can be either an atomic value (e.g., 1, "a"), or an expression that evaluates to a value of the correct type.

Operators

There are five types of operators in the MRRoboto language: string, mathematical, equality, logical, and parenthetical. All operators in this language are infix.

There is only one string operator, the binary concatenation operator "++", which takes two strings as operands and combines them into one.

There are four mathematical operators: addition ("+"), multiplication ("*"), division ("/"),and subtraction ("-"), each of which takes two integers as operands. If the result of an integer operation would otherwise be a decimal number, the decimal part is ignored and a whole number is returned as if the result had been processed by the "floor" operation (e.g., 5/2 = 2, which is the same as floor(2.5)).

There are seven equality operators, "=", "==", "<", ">", "<>", "<=" and ">=". All of them take two operands, which can be of any type but must both be of the same type. The first operator assigns the value of the right operand (which may be either a variable or literal value) to the left operand (which must be a variable). The other equality operators return either 0 for true or 1 for false, depending on whether the first operand is exactly equal to, less than, greater than, not equal to, less than or equal to, or greater than or equal to the second operand, respectively.

There are two logical operators, "&&" and "||", which take two operands of integer type. The "&&" operator returns 1 if both its operands are non-zero; otherwise it returns 0. The "||" operator returns 1 if either of its operands are non-zero; otherwise it returns 0.

The parenthetical operators are "(" and ")", which must always be matched. These operators do nothing by themselves, but any statement inside them is "promoted" to the highest level of precedence and evaluated as an atomic unit.

Precedence

Precedence for the operators follows roughly the same rules as in the C and Java programming languages. The operators are listed below in order of increasing precedence (operators on the same line have equal precedence):

```
=
||
&&
< > <= >= == <>
+ - ++
* /
( )
```

Flow Control

MRRoboto allows the programmer to control the flow of program execution through two types of constructs: conditional statements, which allow a block of code to be skipped, and loop statements, which allow a block of code to be executed multiple times.

MRRoboto supports one type of conditional statement, namely the standard if-else type conditional common to many programming languages.

If-Else Statements

An if-else conditional statement consists of the keyword "if" followed by a boolean expression (i.e., one that evaluates either to 0 or to non-zero), followed by a block of code, followed by an optional "else" keyword followed by a block of code, followed by an "end" keyword. If the boolean expression evaluates to non-zero, the block of code following the "if" will be executed; otherwise, if there is an "else" statement, the block of code following it will be executed. In either case, program execution then continues starting at the next statement following the "end". Conditional statements can be nested; in such a case, the "end" aligns with the most recent "if".

Conditional Statement Grammar

```
conditional
      : "if " expr newline block ("else" newline block)? "end"
```

Loop Statements

MRRoboto supports two types of loop statements, the for loop which allows a block of code to be executed a certain number of times, and the while loop, which allows a block of code to be executed repeatedly until a certain condition is met.

Loop Statement Grammar

```
loop
      : while_loop | for_loop
```

For Loops

Similar to the "for" statement in common languages such as C++ and Java, the MRRoboto "for" loop sets an initial variable and repeatedly executes a block of code, altering the initial variable's value in a predetermined way at each execution until it reaches some predefined termination condition. Specifically, the MRRoboto "for" loop takes a variable (which must be of integer format), sets its value to the first value given, and by default, at each iteration increments its value by 1 until it reaches the second value given, which is the termination condition. If the optional "step" keyword is supplied, the variable's value will be incremented by the supplied value rather than by 1.

For Loop Statement Grammar

```
for_loop
      : "for " identifier "," value " to " value ("step" value)?
newline block "end"
```

While Loops

The MRRoboto "while" loop takes a boolean expression and repeatedly executes a block of code until that expression evaluates to 0. (Obviously, the code should alter the value of that expression in some way at each iteration, to prevent the loop from running indefinitely.) Like the for loop, a while loop is closed by the keyword "end" to indicate that the following code is no longer considered part of the loop.

While Loop Statement Grammar

```
while_loop
      : "while " expr newline block "end"
```

User-Defined Procedures

In addition to the main block of code, the MRRoboto language allows programmers to avoid needless repetition by encapsulating code that may be called multiple times during the execution of a program into one or more user-defined functions, which when invoked will run the code they contain.

Functions are defined separately from the main block of code, separated from it and by each other by blank lines. They begin with the keyword "function" and end with the keyword "end". Each function has a name, which must be a valid identifier; a function may have the same name as a variable, but not the same name as another function. After its name, a function must have a set of parentheses which may contain one or more arguments. In the body of a function is a block of code ending with one "return" statement.

Function Grammar

```
procedure
      : procedure_declaration (stmt)* "end"! (NEW_LINE!)*

procedure_declaration
      : "procedure"! (ID)? ID^ LEFT_PAREN! (argument_list)?
         RIGHT_PAREN!
```

Arguments

A function is defined with one or more arguments, which are variables associated with that function whose value is set when the function is invoked. Arguments passed to a function must be of the same number and types as the function defines them. All arguments are passed by value, meaning that any change made to a variable passed as an

argument to a function within that function does not affect the value of that variable in the invoking block of code.

Return Values

Each user-defined procedure that returns a value must end with a line containing the keyword "return" and a value, which can be either a numeric literal or an integer variable. When a function executes, its value as a statement in the invoking block of code is considered to be the same as its return value.

Invocation

Procedures can be called either from the main block of code or from within another function via a statement that consists of the function's name, an opening parenthesis, an optional list of arguments of the correct number and types, and a closing parenthesis.

Procedure Call Grammar

```
function_call
      : identifier "(" (value (", " value)*)?")"
```

Scope

MRRoboto incorporates a series of scope rules to determine the "visibility" of variables and user-defined functions. All scoping in this language is static, meaning that the scope of a variable or function can be completely determined at compile time.

Variable Scope

MRRoboto has no global variables. Each variable can be "seen" only by other statements in the same block of code; a variable declared in the main block of code can be affected only by other statements in that main block, while a variable declared in the body of a function can be affected only by other statements in that function. The one exception to this rule is function arguments, which allow the value of variables declared in one block of code to be visible to another block of code.

Function Scope

All functions in this language have global scope, meaning they can be invoked from anywhere: within the main block, within another function, or even within their own body (i.e., recursive function calls are allowed).

Library Functions

The MRRoboto language has a variety of built-in library functions which the programmer can always invoke from any point in a program to perform certain specialized tasks. These functions are the heart of the virtual user interface for controlling the keyboard and mouse.

Library Functions

`mouseMove(int, int)`	move the mouse to the screen coordinate (x,y)
`click()`	single click of the left mouse button at the current mouse position
`moveAndClick(int, int)`	move the mouse to the screen coordinate (x,y) and single slick the left mouse button
`rightClick()`	single click the right mouse button
`moveAndRightClick(int, int)`	move the mouse to the screen coordinate (x,y) and single click the right mouse button
`doubleClick()`	click the left mouse button twice at its current position
`moveAndDoubleClick(int, int)`	move the mouse to the screen coordinate (x,y) and double click
`wait(int)`	pause the program for the given number of milliseconds
`setWait(int)`	set the automatic wait between keyboard and mouse events to the given value
`type(string)`	create keyboard events for the given string
`press(string)`	press and release the given key
`hold(string)`	press and hold the given key
`release(string)`	release the given key
`exec(string)`	execute the given string command
`print(string)`	print the given string to the console
`printInt(int)`	print the given integer to the console
`substring(string, int, int)`	returns the substring of the given string between the the two given integer values
`length(string)`	return the integer length of the given string
`pc(int, int)`	return a string containing the hexadecimal pixel value at the given coordinate (x,y)
`mouseHold(int)`	hold the left mouse button down for given amount of time

Project Plan

Our regular group meetings provided a basis to understanding the goals of our project and accomplishing each task both as an individual and as a group. At each meeting, we would lay out the scope of the language and determine how to best to implement each part. During development and testing, our meetings were even more helpful as this was the only way that everyone could communicate regarding their individual tasks. Each person was responsible for performing unit testing on their respective pieces before it was sent to the other members. This helped cut down the number of errors that could have come up otherwise.

Team Responsibilities

Since the start of the project, we have held regular group meetings after class for about an hour to discuss the items that have been completed and what areas need more focus in order to reach our goals. In addition to the meetings, we also set tasks for each member of the group. Listed below is what each team member contributed to the MRRoboto project.

Jason Kopylec	Front-end, CodeGenerator, Library, testing, documentation
Adam Marczyk	Treewalker, SymbolTable, SemanticAnalyzer, testing, documentation
Hema Krishnan	Lexer, parser, testing, documentation

Programming Style Guide

Our object was to create a language that could be understood by anyone easily. In following through with this objective, we set some basic guidelines regarding the style in which to write the language. The coding conventions that we adhered to aided us in making modifications to the existing functionality without waiting for the original author(s) to to explain everything. This, in turn, allowed for faster development. In addition to this, all code was to be commented as clearly as possible.

ANTLR Coding Style

All rule expansions will start on the line after the rule name in the lexer and parser. The above shall be followed unless, the rule is defining an operator. Since these are usually short, it did not seem logical to write both the name and definition on separate lines.

The colon ":" will always start on the next line one tab key away. The ";" will finish the rule by being placed under the colon on the next line, unless an operator is being defined. In this case, the ";" is placed at the end of the line.

Token names, defined in the parser, are written in upper-case, while the syntactic terms are written in lower-case. All other code and comments follow the Java coding style.

JAVA Coding Style
All necessary indentation, under function definitions or otherwise, will be one tab key away.

The left brace will occupy the same line as the name of the method, and will be placed after one space.

The corresponding right brace will occupy one full line at the end, being placed in the same column as the start of the method declaration.

The else-part of an if-statement will follow in the same line as the right brace for that if-statement.

Naming conventions will be almost the same that Java follows. Class names shall be devised such that every word starts with an upper-case letter and all names start with MRR. Variable names will start with a lower-case letter, with every word that follows starting with a capital letter. Method names follow the same style as well.

Comments follow the Javadoc standards.

Project Timeline

September 28th	Language white paper
Last week of October	Language grammar
October 21st	Language reference manual
Last week of October	Front-end
Last week of November	Treewalker, Symbol Table, Semantic Analyzer
First week of December	Library
Second week of December	Testing and Error Checking
December 20th	Entire project and final report

Software Development Environment

The primary programming language is Java and therefore most of the major files are written in it. The lexer and the parser are written using Antlr grammar syntax. These grammar files, in turn, are translated into Java code.

Operating Systems:
As Java is a platform-independent language, it can be used as a development tool anywhere. Even though this is true, we needed a GUI based system to showcase the full capabilities of our language. For this reason, we chose to use Windows to develop and

test our tool. Also, all of us had access to this system, which made it easier to collaborate on this project. Each of developed on our respective machines and shared the source files for further implementation and testing.

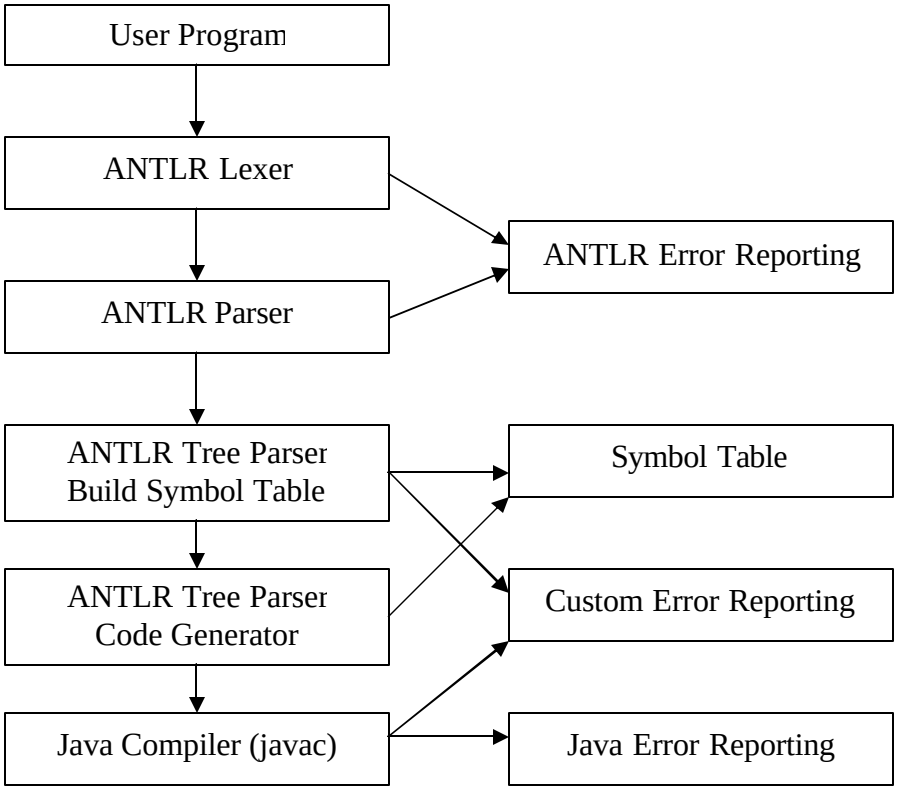
Java 1.4:
Java is a simple, easy-to-use object-oriented language that is portable and secure. Our primary interest in the language lay in the Robot class, which has been in existence since Java 1.3. This along with the rest of he classes available helped set up the symbol table, sematic analyzer and code generator.

Antlr:
The language lexer and parser are both written in Antlr. Antlr is a language tool that provides a framework for constructing recognizers, compilers, and translators from grammatical descriptions containing Java or C++ actions. ANTLR provides excellent support for tree construction, tree walking, and translation. Our purpose was to translate our language into Java, for which Antlr was the perfect tool to use.

Project Log

September 28th	Language white paper
Last week of October	First version of language grammar
October 21st	Language reference manual
October 25th	Front-end
Last week of November	Final version of grammar
November 20th	Symbol Table
Third week of November	First version of Treewalker
Second week of December	Final version of Treewalker & Semantic Analyzer
First week of December	Library
Second week of December	Testing and Error Checking
December 20th	Entire project and final report

Architectural Design



User Program - The MRRoboto code generated by the user to be converted and run as a java program

ANTLR Lexer/ANTLR Parser – Check syntax and build the AST. There is build i **ANTLR Error Reporting** that is relied on by the MRRoboto compiler to detect and alert the user to syntax errors.

Build Symbol Table – This is the first of a two pass compiler. It scans through the user file looking only for variable declarations and procedure declarations. These declarations are used to build the symbol table that is referred to in the second pass of the compiler to do reference and type checking.

ANTLR Tree Parser/Code Generator – This is the second walking of the AST checking static semantics, generating the destination java code or reporting any errors.

MRRoboto.java – This overall picture is managed by this module, it is the main procedure run which initializes the ANTLR pieces, holds the symbol tables and guides the code generator and error reporters. It is through this module that all the other components communicate with one another.

Test Plan

The testing procedures grew alongside the development of the language itself. At the beginning as the project began to be coded, different group members were working on separate parts of the language, so each had different issues to look at when it came to testing.

The first stage of development began on two fronts. Hema and Adam worked on generating the ANTLR lexer and parser while Jason implemented the library functions and the preliminaries for the tree walker and code generator. The parser and generator were tested on an extremely fine grained and incremental basis. Any modifications that were made to parts already deemed as previously working, would be rerun with tested to ensure that there was no loss of functionality.

The opposite work, building the code generator, focused initially on only two pieces of functionality in the language, comments and native java code. The syntax and semantics for these language primitives are straight forward, and the focus instead was to get a complete and running compiler which performed parsing and generating ASTs, walking those ASTs and generating code and/or error messages. Next variable and procedure declarations were added in order to facilitate the implementation of the first pass of the compiler which built the symbol table and handled scope for the second pass.

Once these were implemented, there were two goals of testing, designing programs that should run, and crafting programs that should fail. By separating these two goals, we were able to ensure that there was no loss of functionality as more features were added and also to assert that our language was allowing only programs that we deemed to legal programs of the language.

Here we show three examples of such programs. The first two were to ensure that properly formed programs were seen as such, and they test the library functions and flow control, respectively. The final test program is built out of purposefully malformed programs. They are syntactically correct, but semantically invalid because of type checking or scope errors, in order to test the quality of the error checking procedures. Following each of the properly formed programs is the code generated by the MRRoboto compiler.

testlibraryfunctions.mrr

```
** MRRoboto Test Suite
** Coded by Jason Kopylec
** Each library function is tested for correctness using this file

mouseMove(50, 100)
click()
moveAndClick(50, 500)
rightClick()
moveAndRightClick(50, 500)
```

```
doubleClick()
moveAndDoubleClick(50, 500)
exec("notepad.exe")
wait(2000)
setWait(1000)
type("Hello")
hold("|ENTER|")
release("|ENTER|")
print("Print works!")
printInt(5)
print(substring("Jason", 1, 2))
printInt(length("Jason"))
print(pc(50, 50))
print("All the library functions Work!!")
```

testlibraryfunctions.java

```
//Generated by MRRoboto

public class testlibraryfunctions {

    public static MRRobotoLibrary mrRobotoLibrary;

    public static void main(String[] args) {
        mrRobotoLibrary = new MRRobotoLibrary();
        usermain();
    }

    public static void usermain() {

        //  MRRoboto Test Suite
        //  Coded by Jason Kopylec
        //  Each module is tested correctness using this file
        //  Test direct java code
        // BEGIN USER CODE
         System.out.println("Direct Java Code Works");
        // END USER CODE
        //  Test Library Functions
        mrRobotoLibrary.mouseMove(50, 100);
        mrRobotoLibrary.click();
        mrRobotoLibrary.moveAndClick(50, 500);
        mrRobotoLibrary.rightClick();
        mrRobotoLibrary.moveAndRightClick(50, 500);
        mrRobotoLibrary.doubleClick();
        mrRobotoLibrary.moveAndDoubleClick(50, 500);
        mrRobotoLibrary.exec("notepad.exe");
        mrRobotoLibrary.wait(2000);
        mrRobotoLibrary.setWait(1000);
        mrRobotoLibrary.type("Hello");
        mrRobotoLibrary.hold("|ENTER|");
        mrRobotoLibrary.release("|ENTER|");
        mrRobotoLibrary.print("Print works!");
        mrRobotoLibrary.printInt(5);
        mrRobotoLibrary.print(mrRobotoLibrary.substring("Jason", 1,
2));
        mrRobotoLibrary.printInt(mrRobotoLibrary.length("Jason"));
```

```
        mrRobotoLibrary.print(mrRobotoLibrary.pc(50, 50));
        mrRobotoLibrary.print("All the library functions Work!!");
    }

}
```

testprimitives.mrr

```
** MRRoboto Test Program
** Tests Primitives
** Coded by Jason Kopylec

**Test if statement
if (2<3)
      print("If looks good!")
else
      print("If is wrong if this prints")
end

**Test for statement
int i
for i, 1 to 5
      printInt(i)
end

**Test for with step
print("")
for i, 2 to 8 step 2
      printInt(i)
end
print("Who do we appreciate?  For loops work!")

** Test while statement
int x
x = 1
while (x<>3)
      print("While is working * 2!")
      x=x+1
end

**Test procedure call
testProcedure()

** Test function call
string f
f = testFunction()
print(f)

procedure testProcedure()
      print("Procedure works!")
end

procedure string testFunction()
      return "Function works!"
end
```

testprimitives.java

```
//Generated by MRRoboto

public class testprimitives {

    public static MRRobotoLibrary mrRobotoLibrary;

    public static void main(String[] args) {
        mrRobotoLibrary = new MRRobotoLibrary();
        usermain();
    }

    public static void usermain() {

        // Test if statement
        if((2 < 3)) {
            mrRobotoLibrary.print("If looks good!");
        } else {
            mrRobotoLibrary.print("If is wrong if this prints");
        }
        // Test for statement
        int i = 0;
        for(i = 1; i <= 5; i += 1) {
            mrRobotoLibrary.printInt(i);
        }
        // Test for with step
        mrRobotoLibrary.print("");
        for(i = 2; i <= 8; i += 2) {
            mrRobotoLibrary.printInt(i);
        }
        mrRobotoLibrary.print("Who do we appreciate?  For loops
work!");
        //  Test while statement
        int x = 0;
        x = 1;
        while((x != 3)) {
            mrRobotoLibrary.print("While is working * 2!");
            x = (x + 1);
        }
        // Test procedure call
        testProcedure();
        //  Test function call
        String f = "";
        f = testFunction();
        mrRobotoLibrary.print(f);
    }

    public static void testProcedure() {
        mrRobotoLibrary.print("Procedure works!");
    }

    public static String testFunction() {
        return "Function works!";
    }}
```

badtest.mrr

```
** MRRoboto Test Suite
** Coded by Jason Kopylec
** Syntax errors are picked up by ANTLR
** This file tests semantic errors


** Test Library Functions
mouseMove("This procedure takes integers")
click("This procedure takes no arguments")
exec(9999)

**Assign an int to a string
string x
x=5


**Assign a string to an int
int y
y="This should error"

**Try doing math on strings
string m
m = "Jason" - "bad program"
```

Lessons Learned

Jason Kopylec

Communication and teamwork are key. There are times during the semester when each group member needed to focus on other work and keeping close tabs on the work being done makes balancing the workload much easier. I definitely learned that even making a programming language that looks much like other languages that we are used to is quite challenging and full of subtle difficulties and theoretical and time limitations. As the team leader, I also learned to put the extra time in needed to see that due dates were met and that team members were keeping on task. We had a motto in our group that was, "it doesn't matter who does it, it's just gotta get done."

My advice to future groups is to familiarize yourselves with the ANTLR tool as early as possible. The class discussions and documentation is thorough, but until you dig in and get your hands dirty, ANTLR will remain foreign. Another important issue to keep in mind with ANTLR is to keep in mind the entire project. It does not make much sense to spend a lot of time on the lexer and parser, just to have to go back and modify large portions of it in order to incorporate the tree walker. Learn to like and work well with your teammates, you will need them and they will need you. Look at old projects when you get stuck or unsure how to implement something.

Adam Marczyk

Keep your language design simple and clean! We figured out very early on (within a week or two of putting our group together) exactly what we wanted our language to do and what capabilities we wanted it to have, which turned out to be a valuable asset later on. Don't clutter your design with unnecessarily complicated syntax rules and don't fall victim to feature creep. Languages narrowly tailored to perform specific tasks are better than general-purpose ones.

We found that version control software is not necessary as long as you maintain a clear understanding of who is responsible for what tasks. We kept backup directories corresponding to each day of work and noted in the source code what had been added since the last backup was made and what remained to be done, which worked well at keeping everything under control.

Writing code in ANTLR is not the kind of task that can be easily parallelized, so group meetings were of limited advantage (what they were best for was coordinating who would do what and working together to figure out how to get started - see the below point on learning curves). Still, you'll probably want to have at least one per week to maintain communication among your group members.

ANTLR is not that difficult to use, and once you know what you're doing with it, you'll be able to put your language together relatively quickly. However, it has somewhat of a learning curve right at the start. Be sure to leave yourself plenty of time to figure out how to use it. (See the last tip.)

We started coding the tree-walker about three and a half weeks in advance. That was enough time, but only just. If you want a comfortable safety margin, I recommend starting closer to four to five weeks in advance (more if your language is more complex). Another month for the parser and two to three weeks for the lexer should be about right. (In fact, the actual coding will take less time than this, but these time estimates factor in the work for the other courses you'll probably be taking.)

The ANTLR homepage at www.antlr.org has example grammar files that correspond to languages such as C++, Java, and so on. These are helpful.

Hema Krishnan

In starting this project, the first and foremost thing that I learned was that communication was the key. It greatly helped having the group meetings and finding out how everyone else was faring. The other thing was value was organization. It helped that we knew who was doing what and therefore learned early on who had expertise in what area. These two things aided our venture in devising our own language. It also did not hurt that all of us in the group knew about the subject matter and were enthusiatic about the project.

It is also important to know that if one gets stuck while coding a critical piece of the puzzle, they should not be hesitant in asking for help, be it from the other group members, the professor or the TA. I found this out the hard way when writing the lexer and parser. You never know who has ideas they can impart to you and solve the problems you are having.

Bibliography

Baldwin, Richard. "Introduction to the Java Robot Class in Java" Developer.com 27 May 2003 <http://www.developer.com/java/other/print.php/2212401>

Bellis, Mary. "The History of the Computer Keyboard." About.com 2004 <http://inventors.about.com/library/inventors/blcomputer_keyboard.htm>

Clark, Mike. "JUnit FAQ." JUnit.org 23 Sept. 2004 <http://junit.sourceforge.net/doc/faq/faq.htm>

Hammell, Thomas. "Extreme Java GUI Testing." Developer.com 26 Apr. 2002 <http://www.developer.com/java/other/print.php/1016841>

Milo. "History of Operating Systems." OSData.com 26 Sept. 2000 < http://www.osdata.com/kind/history.htm>

MSDN.com 2004 <http://www.msdn.com>

Parr, Terrence. "ANTLR Reference Manual" ANTLR 9 May 2004 <http://www.antlr.org/doc/index.html>

"Robot (Java 2 Platform SE v1.4.2)" Sun.com <http://java.sun.com/j2se/1.4.2/docs/api/java/awt/Robot.html>

"Shooting Range" Animated Game. Flash Games <http://www.t45ol.com/play_flash_us.php?ID=791>

Tuck, Mike. "The Real History of the GUI" SitePoint.com 13 Aug. 2001 <http://www.sitepoint.com/print/real-history-gui>

"Windows Macro." Website Search. Download.com 26 Sept. 2004 <http://www.download.com/3120-20-0.html?qt=windows+macro&tg=dl-2001>

Appendix A: Test Code

goodtest.mrr

```
** MRRoboto Test Suite
** Coded by Jason Kopylec
** Each module is tested correctness using this file

** Test direct java code
<% System.out.println("Direct Java Code Works"); %>

** Test Library Functions
mouseMove(50, 100)
click()
moveAndClick(50, 500)
rightClick()
moveAndRightClick(50, 500)
doubleClick()
moveAndDoubleClick(50, 500)
exec("notepad.exe")
wait(2000)
setWait(1000)
type("Hello")
hold("|ENTER|")
release("|ENTER|")
print("Print works!")
printInt(5)
print(substring("Jason", 1, 2))
printInt(length("Jason"))
print(pc(50, 50))

print("All the library functions Work!!")

**Test if statement
if (2<3)
       print("If looks good!")
else
       print("If is wrong if this prints")
end

**Test for statement
int i
for i, 1 to 5
       printInt(i)
end

**Test for with step
print("")
for i, 2 to 8 step 2
       printInt(i)
end
print("Who do we appreciate?  For loops work!")

** Test while statement
int x
x = 1
while (x<>3)
       print("While is working * 2!")
       x=x+1
end

**Test procedure call
testProcedure()

** Test function call
string f
f = testFunction()
print(f)
```

```
procedure testProcedure()
	print("Procedure works!")
end

procedure string testFunction()
	return "Function works!"
end
```

BadTest.mrr

```
** MRRoboto Test Suite
** Coded by Jason Kopylec
** Syntax errors are picked up by ANTLR
** This file tests semantic errors

** Test direct java code
<% System.out.println("Direct Java Code Works"); %>

** Test Library Functions
mouseMove("This procedure takes integers")
click("This procedure takes no arguments")
exec(9999)

**Assign an int to a string
string x
x=5


**Assign a string to an int
int y
y="This should error"

**Try doing math on strings
string m
m = "Jason" - "bad program"
```

Appendix B: ANTLR Code

mrroboto.g

```
/************************************************************
 *  Programming Languages, Fall 2004
 *  Authors:   Jason Kopylec - jkk2106
 *             Hema Krishnan - hk2230
 *             Adam Marczyk  - alm2126
 *
 *  MRRoboto ANTLR Language Lexer, Parser and TreeWalker
 *
 ************************************************************/

//////////////////////////////////////////////////////////////////////

class MRRobotoLexer extends Lexer;

options {
       k = 2;
       testLiterals = false;
       charVocabulary = '\3'..'\377';
       exportVocab = MRRoboto;
}

SPACE
       : ( ' '
       | '\t'
       ) { $setType(Token.SKIP); }
       ;

NEW_LINE
       : '\n' { newline(); }
       | '\r'
       | '\n' '\r'
       { newline(); }
       ;

protected
LETTER
       : 'a'..'z'
       | 'A'..'Z'
       ;

protected
DIGIT
       : '0'..'9'
       ;

ID
       options { testLiterals=true; }
       : LETTER (LETTER | DIGIT | '_')*
       ;

/* Denotes a block of Java code to copy to the output file unparsed. */
JAVA
       : "<%"! ( options { greedy=false; } : (.))* "%>"! NEW_LINE!
       ;

/* Comments begin at a ** token and extend to the end of the line. */
COMMENT
       : "**"! ( options { greedy=true; } : ~( '\n' | '\r' ))*
       ;

/* Note that the first quotation mark is kept. This is deliberate. Ensuring
   that all string literals start with the " character allows us to
   distinguish literal strings from other data types in the AST. */
STRING_LITERAL
       : "\"" ( options { greedy=false; } : (.))* "\""!
```

```
        ;

INT_LITERAL
        : (DIGIT)+
        ;

/* Operators in MRRoboto obey the following precedence rules,
   from lowest to highest:
   =
   ||
   &&
   < > <= >= == <>
   + - ++
   * /
   ( )
 */
EQUALS          : '=';
LEFT_PAREN      : '(';
RIGHT_PAREN     : ')';
COMMA           : ',';
PPLUS           : "++"; //string concatenation
PLUSOP          : '+' | '-';
MULTOP          : '*' | '/';
GTE             : ">=";
LTE             : "<=";
GT              : '>';
LT              : '<';
EQ              : "==";
NEQ             : "<>";
AND             : "&&";
OR              : "||";

////////////////////////////////////////////////////////////////////////

class MRRobotoParser extends Parser;

options {
        buildAST = true;
        k = 2;
}

tokens {
        PROGRAM;
        MAIN_BLOCK;
        PROCEDURE;
        PROCEDURE_CALL;
        VAR_DECL;
        ARGUMENT_LIST;
}

/* A program in MRRoboto consists of a main block (one or more statements)
   followed by zero or more procedure definitions. */
program
        : main_block (procedure)* EOF!
        { #program = #([PROGRAM, "PROGRAM"], program); }
        ;

main_block
        : (stmt)+
        { #main_block = #([MAIN_BLOCK, "MAIN_BLOCK"], main_block); }
        ;

/* A procedure definition consists of the procedure declaration
   (defines the name of the procedure and its arguments, followed
   by zero or more statements, terminated by the "end" keyword. */
procedure
        : procedure_declaration (stmt)* "end"! (NEW_LINE!)*
        { #procedure = #([PROCEDURE, "PROCEDURE"], procedure); }
        ;

/* A statement can be any of the following: a block of unparsed Java code,
```

```
   a comment, a variable declaration, an assignment, a procedure call,
   an if statement, a for loop, a while loop, a directive (break,
   continue or return), or whitespace. */
stmt
        : JAVA { #stmt = #([JAVA], stmt); }
        | COMMENT { #stmt = #([COMMENT], stmt); }
        | var_declaration
        | assignment
        | procedure_call
        | conditional
        | for_loop
        | while_loop
        | directive
        | NEW_LINE!
        ;

/* A procedure declaration, found at the beginning of every procedure
   definition (prototyping is neither necessary nor allowed), consists
   of the keyword "procedure", an optional return type (assumed to be void
   if not provided), a unique name, and an optional list of arguments. */
procedure_declaration
        : "procedure"! (ID)? ID^ LEFT_PAREN! (argument_list)? RIGHT_PAREN!
        ;

/* Argument definitions consist of a comma-separated list of variable definitions. */
argument_list
        : var_declaration (COMMA! var_declaration)*
        ;

directive
        : "break"^
        | "continue"^
        | "return"^ boolexpr
        ;

conditional
        : "if"^ LEFT_PAREN! boolexpr RIGHT_PAREN! NEW_LINE!
        (stmt)*
        ("else" (stmt)*)?
        "end"! NEW_LINE!
        ;

/* The 'for' construct, similar to for loops in C and Java, is used
   to iterate over a block of code a given number of times.
   An example:
       int x
       for x, 1 to 10 step 2
          ....
       end
   (Notice that the variable used as the index must be of
   integer type and must be declared beforehand.)
   The for loop sets the index variable to the starting value
   and increments it by the step amount before each iteration
   until it is equal to or greater than the ending value, at
   which point the loop terminates without iterating again.
   The step amount is optional and assumed to be 1 if not provided.
   Note that termination is not guaranteed if the body of the
   for loop modifies the index variable unpredictably or if
   the ending value is less than the starting value. */
for_loop
        : "for"^ ID COMMA! boolexpr "to"! boolexpr
        ("step" boolexpr)?
        (stmt)* "end"! NEW_LINE!
        ;

while_loop
        : "while"^ LEFT_PAREN! boolexpr RIGHT_PAREN!
        (stmt)* "end"! NEW_LINE!
        ;

procedure_call
```

```
        : ID LEFT_PAREN! (arguments)? RIGHT_PAREN!
        { #procedure_call = #([PROCEDURE_CALL, "PROCEDURE_CALL"], procedure_call); }
        ;

arguments
        : boolexpr (COMMA! boolexpr)*
        { #arguments = #([ARGUMENTS, "ARGUMENTS"], arguments); }
        ;

var_declaration
        : ID ID
        { #var_declaration = #([VAR_DECL, "VAR_DECL"], var_declaration); }
        ;

assignment
        : ID EQUALS^ boolexpr
        ;

boolexpr
        : boolexpr1 (OR^ boolexpr1)*
        ;

boolexpr1
        : bool (AND^ bool)*
        ;

bool
        : expr ((GTE^ | LTE^ | GT^ | LT^ | EQ^ | NEQ^) expr)*
        ;

expr
        : expr1 ((PLUSOP^ | PPLUS^) expr1)*
        ;

expr1
        : value (MULTOP^ value)*
        ;

value
        : ID
        | STRING_LITERAL
        | INT_LITERAL
        | procedure_call
        | LEFT_PAREN! boolexpr RIGHT_PAREN!
        ;

////////////////////////////////////////////////////////////////////////

/* The tree-walker makes two passes. On the first pass, it captures
   all procedure names (along with their types and the types of their
   arguments) and stores them as globally scoped symbols; on the
   second pass, it does everything else. */

class MRRobotoWalker extends TreeParser;

{
    MRRobotoCodeGenerator cg = new MRRobotoCodeGenerator();
    MRRSemanticAnalyzer semantics = new MRRSemanticAnalyzer();
    String error = "";
    String typeError = "";
}

/* This is used to create a Java class file with the same name
   as the .mrr file it was created from. */
createfile : #(fn:ID { cg.createClass(fn.getText()); } );

////////////////// begin first pass rules /////////////////

/* On the first pass, take only procedures - ignore the main block. */
firstpass : #(PROGRAM ignore_main_block! (store_procedure_names)* );
```

```
ignore_main_block : #(MAIN_BLOCK {});

store_procedure_names : #(PROCEDURE procedure_name {});

/* For each procedure definition, store the name, the return type,
   and the type of its arguments. Procedure types are optional in
   this language, and assumed to be void if none is provided. */
procedure_name : #(procname:ID (proctype:ID)? (procedure_argument)*
       {
         String type = (proctype == null ? "void" : proctype.getText());
         if(!type.equals("void") && !semantics.isValidType(type))
              error += "Bad return type: "+type+"\n";
         semantics.stopStoringArguments(procname.getText(), type);
       } );

procedure_argument : #(VAR_DECL type:ID ID { semantics.addArgument(type.getText()); } );

////////////////// end first pass rules //////////////////

/* As we walk the tree, we translate MRRoboto code into Java code,
   all of which is buffered in the code generator. We also buffer
   any semantic errors we may encounter. When we're done, if the code
   translated cleanly, tell the code generator to write it out to a file;
   otherwise, print a listing of errors to the display and quit. */
program : #(PROGRAM ( main_block (procedure)* { cg.endCode(error); }));

main_block : { semantics.enterScope(); }
          #(MAIN_BLOCK (stmt)*
         { semantics.leaveScope(); }
         );

procedure: #(pdec:PROCEDURE
       { semantics.enterScope(); }
       procedure_declaration
       (stmt)*
       {
         if(semantics.getNeedsReturn() && !semantics.hasReturnStmt())
           error += "Procedure "+semantics.inFunction()+" requires a return statement\n";
         semantics.leaveScope();
         semantics.exitFunction();
       }
       );

stmt : (
       /* Non-comment statements following a break, continue or return
          statement at the same level of scope are unreachable. */
       {
         if(semantics.unreachable())
           error += "Unreachable statement\n";
         cg.writeIndent();
        /* This is for functions that need a return statement.
           We could be clever about checking for this; we won't be.
           Instead, we simply assert our godlike powers as language
           designers to force the last statement of any typed function
           to be the return statement. */
          semantics.clearReturn();
       }
       (java
       | var_declaration { cg.endline(); }
       | assignment { cg.endline(); }
       | conditional
       | procedure_call { cg.endline(); }
       | for_loop
       | while_loop
       | directive { cg.endline(); } )
       )
       | comment
       ;

/* Directives, unlike all other statements, can't go just anywhere. Make
   sure their placement in the code is valid, and note that any statements
```

```
   immediately following them at the same scope level are unreachable. */
directive :
        #("break"
        {
            if(!semantics.inLoop())
               error += "'break' statement outside loop\n";
            semantics.setUnreachable();
            cg.generateDirective("break");
        } )
        | #("continue"
        {
            if(!semantics.inLoop())
               error += "'continue' statement outside loop\n";
            semantics.setUnreachable();
            cg.generateDirective("continue");
        } )
        | #("return"
        {
            semantics.setUnreachable();
            semantics.setReturn();
            cg.generateDirective("return ");
        }
          ret:tree_expr
        {
            if(semantics.inFunction() == null)
               error += "'return' statement outside function\n";
            else if(!semantics.verifyType(ret,
semantics.lookupType(semantics.inFunction(), true)))
               error += "Bad return type: expected
"+semantics.lookupType(semantics.inFunction(), true)+", got
"+semantics.inferType(ret)+"\n";
        } )
        ;

/* When beginning a loop, tell the semantic checker that we've entered
   a new level of scope, and also let it know that we're inside a loop
   (i.e., break and continue directives will be valid here). */
for_loop :
        {
          semantics.enterScope();
          semantics.beginLoop();
        }
        #("for" index:ID
        {
          if(!semantics.isDefined(index.getText()))
               error += "Undefined variable "+index.getText()+"\n";
          if(!semantics.verifyType(index, "int"))
               error += "Index variable "+index.getText()+" in 'for' loop must be int\n";
          cg.generateForLoop(index.getText());
        }
        start:tree_expr
        {
          cg.continueForLoop(index.getText());
          if(!semantics.verifyType(start, "int"))
               error += "Start condition in 'for' loop must be int\n";
        }
        end:tree_expr
        {
          cg.completeForLoop(index.getText());
          if(!semantics.verifyType(end, "int"))
               error += "End condition in 'for' loop must be int\n";
        }
        ("step" { cg.generateStep(); } step:tree_expr
          {
            if(!semantics.verifyType(start, "int"))
               error += "Step expression in 'for' loop must be int\n";
          } )?
        { cg.closeForLoop(); }
        (stmt)*
        {
          cg.closeBlock();
```

```
          semantics.exitLoop();
          semantics.leaveScope();
        }
        );

while_loop :
        {
          semantics.enterScope();
          semantics.beginLoop();
        }
        #("while" { cg.generateWhileLoop(); }
        cond:tree_expr
        {
          if(!semantics.verifyType(cond, "boolean") &&
             !semantics.verifyType(cond, "comparator"))
               error += "Condition in 'while' loop must be boolean\n";
          cg.closeWhileLoop();
        }
        (stmt)*
        { cg.closeBlock(); semantics.leaveScope(); semantics.exitLoop(); }
        );

/* Conditional statements, by contrast, do constitute a new scope level,
   but aren't loops. Note that an "if" block and the associated "else"
   block are different scopes - this is the same way it is in Java. */
conditional : { semantics.enterScope(); }
        #("if" { cg.generateConditional(); }
        cond:tree_expr
        {
          if(!semantics.verifyType(cond, "boolean") &&
             !semantics.verifyType(cond, "comparator"))
               error += "Condition in 'if' statement must be boolean\n";
          cg.closeConditional();
        }
        (stmt
        | { semantics.leaveScope(); semantics.enterScope(); } "else" { cg.generateElse();
}
        )*
        { cg.closeBlock(); semantics.leaveScope(); } );

procedure_declaration : #(procname:ID
        {
            semantics.enterFunction(procname.getText());
        }
        (proctype:ID { semantics.setNeedsReturn(); })?
        {
            cg.generateProcedureDeclaration(procname.getText(),
semantics.lookupType(procname.getText(), true));
            cg.startArgumentList();
        }
        (var_declaration ( { cg.writeComma(); } var_declaration)*)?
        {
             cg.endArgumentList();
             cg.closeProcedureDeclaration();
        }
        );

var_declaration : #(vint:VAR_DECL type:ID name:ID
        {
        if(!semantics.isValidType(type.getText()))
            error += "Unknown type: "+type.getText()+"\n";
        cg.generateVarDeclaration(type.getText(), name.getText());
        if(semantics.addSymbol(name.getText(), type.getText()) == -1)
            error += "Redefinition error: "+name.getText()+"\n";
        }
        );

comment : #(c:COMMENT { cg.generateComment(c.getText()); } );

java : #(j:JAVA { cg.generateUserJava(j.getText()); } );
```

```
assignment : #(assgn:EQUALS
        { cg.generateAssignment(assgn.getFirstChild().getText()); }
        id:ID rvalue:tree_expr
        {
            if(!semantics.isDefined(id.getText()))
               error += "Undefined variable "+id.getText()+"\n";
            else if(semantics.symbolIsFunc(id.getText()))
               error += "Lvalue in assignment must be a defined variable\n";
            else if((typeError = semantics.checkTypes(assgn, id, rvalue)) != null)
               error += typeError+"\n";
        }
        );

/* Since precedence levels are handled in the parser, we don't need
   to handle them here. All arithmetic and boolean operators can be handled
   as aspects of a single rule; semantic analysis will tell us if
   any types don't match. We do need to handle comparators somewhat
   differently for strings and ints, though. */
tree_expr :
        #(aop:AND
            { cg.lparen(); }
            t:tree_expr
            { cg.generateOp(aop.getText()); }
            t0:tree_expr
            {
              if((typeError = semantics.checkTypes(aop, t, t0)) != null)
               error += typeError+"\n";
            }
            { cg.rparen(); }
        ) |
        #(oop:OR
            { cg.lparen(); }
            tA:tree_expr
            { cg.generateOp(oop.getText()); }
            tB:tree_expr
            {
              if((typeError = semantics.checkTypes(oop, tA, tB)) != null)
               error += typeError+"\n";
            }
            { cg.rparen(); }
        ) |
        #(g1:GT
            { cg.lparen(); }
            t1:tree_expr
            {
              if(semantics.inferType(t1).equals("string")) cg.stringComp();
               else cg.generateOp(g1.getText());
            }
            t2:tree_expr
            {
              if(semantics.inferType(t2).equals("string"))
                cg.stringComp2("== 1");
              if((typeError = semantics.checkTypes(g1, t1, t2)) != null)
               error += typeError+"\n";
            }
            { cg.rparen(); }
        ) |
        #(l:LT
            { cg.lparen(); }
            t3:tree_expr
            {
              if(semantics.inferType(t3).equals("string")) cg.stringComp();
              else cg.generateOp(l.getText());
            }
            t4:tree_expr
            {
              if(semantics.inferType(t4).equals("string"))
                cg.stringComp2("== -1");
              if((typeError = semantics.checkTypes(l, t3, t4)) != null)
               error += typeError+"\n";
            }
```

```
            { cg.rparen(); }
        ) |
        #(g2:GTE
            { cg.lparen(); }
            t5:tree_expr
            {
              if(semantics.inferType(t5).equals("string")) cg.stringComp();
              else cg.generateOp(g2.getText());
            }
            t6:tree_expr
            {
              if(semantics.inferType(t6).equals("string"))
                cg.stringComp2("> -1");
              if((typeError = semantics.checkTypes(g2, t5, t6)) != null)
               error += typeError+"\n";
            }
            { cg.rparen(); }
        ) |
        #(l2:LTE
            { cg.lparen(); }
            t7:tree_expr
            {
              if(semantics.inferType(t7).equals("string")) cg.stringComp();
              else cg.generateOp(l2.getText());
            }
            t8:tree_expr
            {
              if(semantics.inferType(t8).equals("string"))
                cg.stringComp2("< 1");
              if((typeError = semantics.checkTypes(l2, t7, t8)) != null)
               error += typeError+"\n";
            }
            { cg.rparen(); }
        ) |
        #(e:EQ
            { cg.lparen(); }
            t9:tree_expr
            {
              if(semantics.inferType(t9).equals("string")) cg.stringComp();
              else cg.generateOp(e.getText());
            }
            t10:tree_expr
            {
              if(semantics.inferType(t10).equals("string"))
                cg.stringComp2("== 0");
              if((typeError = semantics.checkTypes(e, t9, t10)) != null)
               error += typeError+"\n";
            }
            { cg.rparen(); }
        ) |
        #(n:NEQ
            { cg.lparen(); }
            t11:tree_expr
            {
              if(semantics.inferType(t11).equals("string")) cg.stringComp();
              else cg.generateOp("!=");
            }
            t12:tree_expr
            {
              if(semantics.inferType(t12).equals("string"))
                cg.stringComp2("!= 0");
              if((typeError = semantics.checkTypes(n, t11, t12)) != null)
               error += typeError+"\n";
            }
            { cg.rparen(); }
        ) |
        #(pop:PLUSOP
            { cg.lparen(); }
            t13:tree_expr
            { cg.generateOp(pop.getText()); }
            t14:tree_expr
```

```
            { cg.rparen(); }
            {
              if((typeError = semantics.checkTypes(pop, t13, t14)) != null)
               error += typeError+"\n";
            }
        ) |
        #(mop:MULTOP
            { cg.lparen(); }
            t15:tree_expr
            { cg.generateOp(mop.getText()); }
            t16:tree_expr
            { cg.rparen(); }
            {
              if((typeError = semantics.checkTypes(mop, t15, t16)) != null)
               error += typeError+"\n";
            }
        ) |
        #(sop:PPLUS
            { cg.lparen(); }
            t17:tree_expr
            { cg.generateOp("+"); }
            t18:tree_expr
            { cg.rparen(); }
            {
              if((typeError = semantics.checkTypes(sop, t17, t18)) != null)
               error += typeError+"\n";
            }
        ) |
        #(num:INT_LITERAL { cg.generateConstExpr(num.getText()); })
        | #(str:STRING_LITERAL { cg.generateConstExpr(str.getText()); })
        | procedure_call
        | #(var:ID
            {
              if(!semantics.isDefined(var.getText()) ||
semantics.symbolIsFunc(var.getText()))
               error += "Undefined variable "+var.getText()+"\n";
              cg.generateConstExpr(var.getText(), true);
            }
        )
        ;

procedure_call : #(pcall:PROCEDURE_CALL
        pname:ID
        {
          boolean isLibFunc = semantics.symbolIsLibFunc(pname.getText());
          cg.generateProcedureCall(pname.getText(), isLibFunc);
        }
        (arguments)?
        {
          String[] argList = semantics.getStoredArguments();
          if(!semantics.isDefined(pname.getText()) ||
!semantics.symbolIsFunc(pname.getText()))
               error += "Undefined function "+pname.getText()+"\n";
          else if(!semantics.matchFuncArguments(pname.getText(), argList))
               error += "Bad function arguments supplied to "+pname.getText()+"\n";
          cg.rparen();
        }
        );

arguments: #(args:ARGUMENTS
        {
          String[] argList = new String[args.getNumberOfChildren()];
          int ptr = 0;
        }
        v:tree_expr
        {
          argList[ptr] = semantics.inferType(v);
          ptr++;
        }
        ( { cg.writeComma(); } v1:tree_expr { argList[ptr] = semantics.inferType(v1);
ptr++; } )*
```

```
        {
          semantics.storeArguments(argList);
        }
        );
```

Appendix C: Java Code

MRRoboto.java

```
/*************************************************************
 * Jason Kopylec
 * Programming Languages, 11/1/04
 *
 * Main Program for compiling MRRoboto files
 * Uses ANTLR generated Lexer, Parser and TreeWalker
 *
 *************************************************************/


import java.io.*;
import antlr.CommonAST;
import antlr.collections.AST;
import antlr.debug.misc.ASTFrame;




public class MRRoboto implements MRRobotoTokenTypes {


//Object variables
private static String sourceFileName;        //.mmr file to be compiled
private static String destinationFileName;   //.java file to be outputted
private static boolean showtree = false;




/*************************************************************
 * MAIN(String[])
 *
 * Takes a .mmr file from the command line and attempts to
 * compile into native Java.
 *
 * @param args - command line argument indicating file
 *               to be parsed
 *
 * @output - <filename>.java with native java code or else
 *           messages explaining why complilation was halted
 *
 *************************************************************/
public static void main(String[] args) {

                                  //Parse arguments passed on the command line
                                  if(!parseCommandLine(args)) return;

                                  //Run the ANTLR created Lexer and Parser and print AST
                                  try {
                                     FileInputStream file = new
FileInputStream(sourceFileName);
                                     MRRobotoLexer l = new MRRobotoLexer(file);
                                     MRRobotoParser p = new MRRobotoParser(l);
                                     p.program();

                                     //Walk parse tree and generate code
                                     MRRobotoWalker w = new MRRobotoWalker();

                                     //Tell the tree-walker's code generator what to name
this file.
                                     CommonAST filename = new CommonAST();
                                     filename.setType(ID);
                                     filename.setText(destinationFileName);
                                     w.createfile(filename);
```

```
                                    w.firstpass((CommonAST) p.getAST());
                                    w.program((CommonAST) p.getAST());

                                    // Open a window in which the AST is displayed
graphically
                                    if(showtree) {
                                        ASTFrame frame = new ASTFrame("MRRoboto Program
Tree", p.getAST());
                                        frame.setVisible(true);
                                    }

                                }
                                catch(Exception e) { error(e.toString());}
}




/**********************************************************
 * PARSECOMMANDLINE(STRING[])
 *
 * Parses command line and places arguments in object variables
 *
 * Command Line should look like:
 *
 * $java [directives]MRRoboto <filename.mmr>
 *       /h - prints compiler usage
 *       /? - prints compiler usage
 *
 * @param String[] args - arguments passed from command line
 *
 **********************************************************/
private static boolean parseCommandLine(String[] args) {

                                //if there are no arguments, print file usage
                                if (args.length==0) {
                                    printUsage();
                                    return false;
                                }

                                //parse directives
                                for(int i=0; i<args.length-1; i++) {

                                   //print usage for /h and /? directives
                                   if (args[i].equals("/showtree")) showtree = true;
                                   else {
                                       error("Unknown argument.");
                                       return false;
                                   }
                                }

                                //Get filename to parse and ensure it ends with .mrr,
error if not
                                sourceFileName = args[args.length-1];
                                if (sourceFileName.endsWith(".mrr")) {
                                   destinationFileName =
sourceFileName.substring(0,sourceFileName.length()-4);
                                }
                                else {
                                   error("Source file must have extension .mrr");
                                   return false;
                                }
                                return true;
}
```

```
/************************************************************
 * PRINTUSAGE()
 *
 * Prints compiler command line and exit. Usage is:
 *
 * $java MRRoboto [<flags>] <filename.mrr>
 *       /h - prints compiler usage
 *       /? - prints compiler usage
 *
 * @output - command line command for invoking the compiler
 *
 ************************************************************/
private static void printUsage() {

                                    String usage = "** MRRoboto Compiler 1.0 **" +
                                              "\n\nUsage:  $java MRRoboto [<flags>]
<filename.mrr>" +
                                              "\nFlags:" +
                                              "\n\t/showtree - displays program syntax
tree";

                                    System.out.println(usage);
                                    System.exit(0);
}



/************************************************************
 * ERROR(STRING)
 *
 * Print an error message and exit
 *
 * @param String msg - Text to display as error
 *
 * @output - error message and program halt
 *
 ************************************************************/
private static void error(String msg) {

                                    System.out.println("MRRoboto Compiler Error: "+msg);
                                    System.exit(1);
}


}
```

MRRobotoCodeGenerator.java

```
/*********************************************************
 * Jason Kopylec
 * Programming Languages, 11/1/04
 *
 * Methods for generating code for MRRoboto Language
 *
 *********************************************************/

import java.io.*;

public class MRRobotoCodeGenerator {

    private final int indentAmt = 4;
    private int curIndent;
    private boolean hasStepAmt;
    private boolean argList;

    private String destFileName;
    private BufferedWriter destFile;
    private StringBuffer fileContents;

    public MRRobotoCodeGenerator() {
       destFileName = "user";
       fileContents = new StringBuffer("");
       curIndent = 0;
    }

    public void createClass(String classname) {
       destFileName = classname;

       write("//Generated by MRRoboto\n\n");
       write("public class "+classname+" { \n\n");
       moreIndent();
       write(indent() + "public static MRRobotoLibrary mrRobotoLibrary;\n\n");
       write(indent() + "public static void main(String[] args) {\n");
       moreIndent();
       write(indent() + "mrRobotoLibrary = new MRRobotoLibrary();\n");
       write(indent() + "usermain();\n");
       lessIndent();
       write(indent() + "}\n\n");
       write(indent() + "public static void usermain() { \n\n");
       moreIndent();
    }

    private void write(String text) { fileContents.append(text); }

    private String indent() {
       String spaces = "";
       for(int i = 0; i < curIndent; i++) spaces += " ";
       return spaces;
    }

    private void moreIndent() { curIndent += indentAmt; }

    private void lessIndent() {
       if(curIndent >= indentAmt) curIndent -= indentAmt;
    }

    public void endCode(String err) {
       lessIndent();
       write(indent() + "}\n\n");
       lessIndent();
       write(indent() + "}\n");
       if(err == null || err.equals("")) {
           try {
              destFile            =            new            BufferedWriter(new
FileWriter(destFileName+".java"));
              destFile.write(fileContents.toString());
              destFile.close();
```

```
            } catch (IOException e) {
                System.out.println("Error writing to '"+destFileName+"!': "+e);
                 System.out.println("Source file creation aborted.");
            }
        } else {
            System.out.println("Errors occurred while creating source file:");
            System.out.println(err.substring(0, err.length()-1));
            System.out.println("Source file creation aborted.");
        }
    }

    public void generateComment(String text) {
        write(indent() +  "// "+text+"\n");
    }

    public void generateUserJava(String text) {
        write("// BEGIN USER CODE\n");
        write(indent() + text + "\n");
        write(indent() + "// END USER CODE\n");
    }

    public void generateVarDeclaration(String type, String id) {
        if(type.equals("string")) type = "String";
        if(argList == false) {
                //Start the java declaration, e.g. String abc =
                write (type + " " + id + " = ");

                //Initialize the variable depending on type
                if (type.equals("String")) write("\"\"");
                if (type.equals("int")) write("0");
        } else {
                write(type + " " + id);
        }
    }

    public void generateAssignment(String id) {
        write(id + " = ");
    }

    public void generateOp(String arg) {
        write(" " + arg + " ");
    }

    public void generateDirective(String arg) {
        write("" + arg + "");
    }

    public void generateElse() {
        lessIndent();
        write(indent() + "} else {\n");
        moreIndent();
    }

    public void generateConstExpr(String arg) {
         generateConstExpr(arg, false);
    }

    public void generateConstExpr(String arg, boolean isVar) {
         if(isVar) write("" + arg + "");
         else if(arg.charAt(0) == '"')
            write("\"" + arg.substring(1) + "\"");
         else
             write("" + arg + "");
    }

    public void lparen() { write("("); }

    public void rparen() { write(")"); }

    public void writeIndent() { write(indent()); }
```

```
            public void endline() { write(";\n"); }

            public void writeComma() { write(", "); }

            public void generateProcedureCall(String procName, boolean isLib) {
               if(isLib) write("mrRobotoLibrary.");
               write(procName + "(");
            }

            public void generateProcedureDeclaration(String procName, String type) {
               //Finish the last procedure
               lessIndent();
               write(indent() + "}\n\n");

               //write java procedure declaration
               if(type.equals("string")) type = "String";
               write(indent() + "public static "+type+" " + procName + "(");
               moreIndent();
            }

            public void closeProcedureDeclaration() {
               write(") {\n");
            }

            public void startArgumentList() { argList = true; }

            public void endArgumentList() { argList = false; }

            public void generateConditional() {
               write("if(");
            }

            public void closeConditional() {
               write(") {\n");
               moreIndent();
            }

            public void closeBlock() {
               lessIndent();
               write(indent() + "}\n");
            }

            public void generateForLoop(String index) {
                write("for("+index+" = ");
                hasStepAmt = false;
            }

            public void continueForLoop(String index) {
                write("; "+index+" <= ");
            }

            public void completeForLoop(String index) {
                write("; "+index+" += ");
            }

            public void generateStep() {
                hasStepAmt = true;
            }

            public void closeForLoop() {
                if(!hasStepAmt) write("1");
                write(") {\n");
                moreIndent();
            }

            public void generateWhileLoop() {
                write("while(");
            }

            public void closeWhileLoop() {
                write(") {\n");
```

```
            moreIndent();
        }

        public void stringComp() {
            write(".compareTo(");
        }

        public void stringComp2(String op) {
            write(") "+op+"");
        }

    }
```

MRRobotoLibrary.java

```
/************************************************
 * Jason Kopylec
 * CS4115 Programming Languages
 * 10/21/04
 *
 * This object holds all the library functions for the
 * M.R. Roboto Language.  This object is created in the
 * compiled user program and the library calls from the
 * source file are converted into calls from this class
 *
 ************************************************/


import java.awt.*;
import java.awt.event.*;
import java.lang.*;
import java.io.*;
import java.util.*;
import java.util.regex.*;

public class MRRobotoLibrary {

//Object variables
public Robot robot;  //Robot that controls the screen
private HashMap keys;  //holds mapping from keyboard character to keycode



public MRRobotoLibrary() {
//Constructor that initilizes java robot and sets environment variables

                           //initialize variables
                           try {
                              robot = new Robot();
                           }
                           catch(AWTException e) {
                              error("Unable to Initialize Java Robot");
                           }

                           //Load keys with keymap
                           getKeyMap();

}



public void mouseMove(int x, int y) {
//Library function that moves the mouse to the point (x,y)

                           robot.mouseMove(x,y);
}


public void mouseHold(int time) {
//Library function that holds down the left mouse button for a specified time
                           robot.mousePress(InputEvent.BUTTON1_MASK);
                           wait(time);
                           robot.mouseRelease(InputEvent.BUTTON1_MASK);
}


public void click() {
//Library function that clicks the left mouse button once at its current location

                           int autodelay = robot.getAutoDelay();
                           setWait(0);
                           robot.mousePress(InputEvent.BUTTON1_MASK);
                           robot.mouseRelease(InputEvent.BUTTON1_MASK);
                           setWait(autodelay);
```

```
                        wait(autodelay);
}



public void moveAndClick(int x, int y) {
//Move the mouse to the screen position (x,y) and click the left mouse button once

                        mouseMove(x,y);
                        click();
}



public void rightClick() {
//Click the right mouse button at the current mouse position

                        int autodelay = robot.getAutoDelay();
                        setWait(0);
                        robot.mousePress(InputEvent.BUTTON3_MASK);
                        robot.mouseRelease(InputEvent.BUTTON3_MASK);
                        setWait(autodelay);
                        wait(autodelay);
}



public void moveAndRightClick(int x, int y) {
//Click the right mouse button at the point (x,y)

                        mouseMove(x,y);
                        rightClick();
}



public void doubleClick() {
//Double click the left mouse button at the current cursor position

                        click();
                        wait(40);
                        click();
}



public void moveAndDoubleClick(int x, int y) {
//Double click the left mouse button at screen position (x,y)

                        mouseMove(x,y);
                        doubleClick();
}



public void wait(int msecs) {
//delay for msecs miliseconds

                        robot.delay(msecs);
}



public void setWait(int msec) {
//Delay for msec miliseconds after generating robot events

                        robot.setAutoDelay(msec);
}
```

```
public void error(String msg) {
//Displays an error message and exits, used when fatal errors have occured
                        System.out.println("JAVA RUNTIME ERROR: " + msg);
                        System.exit(0);
}



public void type(String text) {
//generate keyboard events for each character in the string
//Invisible keys are put in between pipes, e.g. "|TAB|"

                        //Turn off delay so typing is fast
                        int autodelay = robot.getAutoDelay();
                        setWait(0);

                        String curChar;
                        int nextChar;

                        while(text.length()!=0) {

                           curChar = text.substring(0,1);

                           //If the next string is an escape code, e.g. "|ENTER|",
pass the whole word
                           if (!curChar.equals("|")) {
                                  press(curChar);
                                  text = text.substring(1,text.length());
                           }
                           else {
                                  //Pass all the characters including |'s
                                  nextChar = text.indexOf("|",1);
                                  if (nextChar==-1) error("Invalid Escape Character -
Missing |:\n'"+text+"'");
                                  curChar =  text.substring(0,nextChar+1);

                                  //handle the case when the character to print is '|'
                                  if (curChar.equals("")) curChar = "|";

                                  press(curChar);
                                  text = text.substring(nextChar+1,text.length());
                           }
                        }

                        //Set autowait back to its previous value
                        setWait(autodelay);
                        wait(autodelay);
}



private void press(String text) {
//Hits one key on the keyboard, text

                        hold(text);
                        release(text);
}



public void hold(String text) {
//Press the key, text, down, used with release() and releaseall()
//Takes only one character (or escape code)

                        //error if more than one key is passed
                        if (text.length()==0) error("Error calling hold():  cannot
accept null string \"\"");

                        //If key is an escape code, strip off start and end '|'s
                        if (text.length()>2 && text.substring(0,1).equals("|")) text =
text.substring(1,text.length()-1);
```

```
                            //convert key to its java key code
                            if (needsShift(text)) hold("SHIFT");
                            int kcode = getKeyCode(text);
                            robot.keyPress(kcode);
}



public void release(String text) {
//Releases the keyboard key, text, used with hold()

                            //error if more than one key is passed
                            if (text.length()==0) error("Error calling hold():  cannot
accept null string \"\"");

                            //If key is an escape code, strip off start and end '|'s
                            if (text.length()>2 && text.substring(0,1).equals("|")) text =
text.substring(1,text.length()-1);

                            //convert key to its java key code
                            int kcode = getKeyCode(text);
                            robot.keyRelease(kcode);
                            if (needsShift(text)) release("SHIFT");
}



public void releaseAll() {
//release any keys that are currently held down, works with hold()

}



private int getKeyCode(String text) {
//Text should be one keyboard character, returns the java key code for that character

                            Integer keyindex = (Integer)keys.get(text);
                            if (keyindex == null) error("Keyboard key not found '" + text +
"'");
                            return keyindex.intValue();
}



private void getKeyMap() {
//Called by the constructor to create the table for mapping from keyboard key from string
text

                            keys = new HashMap();

                            //lowercase letters
                            addKey("a", KeyEvent.VK_A);
                            addKey("b", KeyEvent.VK_B);
                            addKey("c", KeyEvent.VK_C);
                            addKey("d", KeyEvent.VK_D);
                            addKey("e", KeyEvent.VK_E);
                            addKey("f", KeyEvent.VK_F);
                            addKey("g", KeyEvent.VK_G);
                            addKey("h", KeyEvent.VK_H);
                            addKey("i", KeyEvent.VK_I);
                            addKey("j", KeyEvent.VK_J);
                            addKey("k", KeyEvent.VK_K);
                            addKey("l", KeyEvent.VK_L);
                            addKey("m", KeyEvent.VK_M);
                            addKey("n", KeyEvent.VK_N);
                            addKey("o", KeyEvent.VK_O);
                            addKey("p", KeyEvent.VK_P);
                            addKey("q", KeyEvent.VK_Q);
                            addKey("r", KeyEvent.VK_R);
```

```
addKey("s", KeyEvent.VK_S);
addKey("t", KeyEvent.VK_T);
addKey("u", KeyEvent.VK_U);
addKey("v", KeyEvent.VK_V);
addKey("w", KeyEvent.VK_W);
addKey("x", KeyEvent.VK_X);
addKey("y", KeyEvent.VK_Y);
addKey("z", KeyEvent.VK_Z);


//uppercase letters
addKey("A", KeyEvent.VK_A);
addKey("B", KeyEvent.VK_B);
addKey("C", KeyEvent.VK_C);
addKey("D", KeyEvent.VK_D);
addKey("E", KeyEvent.VK_E);
addKey("F", KeyEvent.VK_F);
addKey("G", KeyEvent.VK_G);
addKey("H", KeyEvent.VK_H);
addKey("I", KeyEvent.VK_I);
addKey("J", KeyEvent.VK_J);
addKey("K", KeyEvent.VK_K);
addKey("L", KeyEvent.VK_L);
addKey("M", KeyEvent.VK_M);
addKey("N", KeyEvent.VK_N);
addKey("O", KeyEvent.VK_O);
addKey("P", KeyEvent.VK_P);
addKey("Q", KeyEvent.VK_Q);
addKey("R", KeyEvent.VK_R);
addKey("S", KeyEvent.VK_S);
addKey("T", KeyEvent.VK_T);
addKey("U", KeyEvent.VK_U);
addKey("V", KeyEvent.VK_V);
addKey("W", KeyEvent.VK_W);
addKey("X", KeyEvent.VK_X);
addKey("Y", KeyEvent.VK_Y);
addKey("Z", KeyEvent.VK_Z);


//numbers
addKey("1", KeyEvent.VK_1);
addKey("2", KeyEvent.VK_2);
addKey("3", KeyEvent.VK_3);
addKey("4", KeyEvent.VK_4);
addKey("5", KeyEvent.VK_5);
addKey("6", KeyEvent.VK_6);
addKey("7", KeyEvent.VK_7);
addKey("8", KeyEvent.VK_8);
addKey("9", KeyEvent.VK_9);
addKey("0", KeyEvent.VK_0);

//Misc printable chars
addKey(" ", KeyEvent.VK_SPACE);
addKey("!", KeyEvent.VK_1);
addKey("@", KeyEvent.VK_2);
addKey("#", KeyEvent.VK_3);
addKey("$", KeyEvent.VK_4);
addKey("%", KeyEvent.VK_5);
addKey("^", KeyEvent.VK_6);
addKey("&", KeyEvent.VK_7);
addKey("*", KeyEvent.VK_8);
addKey("(", KeyEvent.VK_9);
addKey(")", KeyEvent.VK_0);
addKey("-", KeyEvent.VK_MINUS);
addKey("_", KeyEvent.VK_MINUS);
addKey("+", KeyEvent.VK_EQUALS);
addKey("=", KeyEvent.VK_EQUALS);
addKey("[", KeyEvent.VK_OPEN_BRACKET);
addKey("{", KeyEvent.VK_OPEN_BRACKET);
addKey("]", KeyEvent.VK_CLOSE_BRACKET);
addKey("}", KeyEvent.VK_CLOSE_BRACKET);
```

```
                            addKey("\\", KeyEvent.VK_BACK_SLASH);
                            addKey("|", KeyEvent.VK_BACK_SLASH);
                            addKey(";", KeyEvent.VK_SEMICOLON);
                            addKey(":", KeyEvent.VK_SEMICOLON);
                            addKey("'", KeyEvent.VK_QUOTE);
                            addKey("DOUBLE_QUOTE", KeyEvent.VK_QUOTE);
                            addKey(",", KeyEvent.VK_COMMA);
                            addKey("<", KeyEvent.VK_COMMA);
                            addKey(".", KeyEvent.VK_PERIOD);
                            addKey(">", KeyEvent.VK_PERIOD);
                            addKey("/", KeyEvent.VK_SLASH);
                            addKey("?", KeyEvent.VK_SLASH);


                            //Invisible Characters
                            addKey("F1", KeyEvent.VK_F1);
                            addKey("F2", KeyEvent.VK_F2);
                            addKey("F3", KeyEvent.VK_F3);
                            addKey("F4", KeyEvent.VK_F4);
                            addKey("F5", KeyEvent.VK_F5);
                            addKey("F6", KeyEvent.VK_F6);
                            addKey("F7", KeyEvent.VK_F7);
                            addKey("F8", KeyEvent.VK_F8);
                            addKey("F9", KeyEvent.VK_F9);
                            addKey("F10", KeyEvent.VK_F10);
                            addKey("SHIFT", KeyEvent.VK_SHIFT);
                            addKey("ENTER", KeyEvent.VK_ENTER);
                            addKey("TAB", KeyEvent.VK_TAB);
                            addKey("ESCAPE", KeyEvent.VK_ESCAPE);
                            addKey("ALT", KeyEvent.VK_ALT);
                            addKey("CONTROL", KeyEvent.VK_CONTROL);
                            addKey("CAPS_LOCK", KeyEvent.VK_CAPS_LOCK);
                            addKey("DELETE", KeyEvent.VK_DELETE);
                            addKey("BACK_SPACE", KeyEvent.VK_BACK_SPACE);
                            addKey("DOWN", KeyEvent.VK_DOWN);
                            addKey("UP", KeyEvent.VK_UP);
                            addKey("LEFT", KeyEvent.VK_LEFT);
                            addKey("RIGHT", KeyEvent.VK_RIGHT);
                            addKey("HOME", KeyEvent.VK_HOME);
                            addKey("END", KeyEvent.VK_END);
                            addKey("INSERT", KeyEvent.VK_INSERT);
                            addKey("PRINTSCREEN", KeyEvent.VK_PRINTSCREEN);
                            addKey("PAGE_DOWN", KeyEvent.VK_PAGE_DOWN);
                            addKey("PAGE_UP", KeyEvent.VK_PAGE_UP);
                            addKey("NUM_LOCK", KeyEvent.VK_NUM_LOCK);
}



private void addKey(String key, int val) {
//Place a key to value mapping in the keys HashMap
                            keys.put(key, new Integer(val));
}



private boolean needsShift(String key) {
//Returns true if the shift key is required to print the character
//Java sucks because it needs to use shift to get lower case letters

                            //if key is a double quote, return true
                            if(key.equals("DOUBLE_QUOTE")) return true;

                            String testPattern = "[A-Z\"]";
                            String otherChars = "~!@#$%^&*()_+{}|:<>?";
                            return Pattern.matches(testPattern, key) ||
(otherChars.indexOf(key.charAt(0)) != -1);
}
```

```
public void exec(String cmd) {
//Executes a shell command, cmd
                           try {
                              Runtime rt = Runtime.getRuntime();
                              rt.exec(cmd);
                              wait(robot.getAutoDelay());
                           }
                           catch(Exception e) {error(e.toString());}
}



public void print(String text) {
//Print text to console window

                           System.out.println(text);
}


public void printInt(int text) {
//Print integer to console window
                           print(""+text);
}


public String substring(String x, int start, int finish) {
//return a substring of x from position start to finish
//if finish is greater than string length, whole string is returned

                           int l = x.length();

                           //if start is negative, or
                           //start is greater than the length of x return "" or
                           //or if start > finish
                           if (start<0 || start>l || start > finish) return "";

                           //if start > x.length return upto the end of the string
                           if (finish>l) return substring(x, start, x.length());

                           return x.substring(start, finish);
}



public int length(String x) {
//Return the length of the string x

                           return x.length();
}


public String pc(int x, int y) {
//Return pixel color at (x,y) in the form RRGGBB
//where rgb are hex values
                           Color c = robot.getPixelColor(x,y);
                           String r,g,b;

                           r = Integer.toHexString(c.getRed()).toUpperCase();
                           if (r.length()==1) r = "0"+r;

                           g = Integer.toHexString(c.getGreen()).toUpperCase();
                           if (g.length()==1) g = "0"+g;

                           b = Integer.toHexString(c.getBlue()).toUpperCase();
                           if (b.length()==1) b = "0"+b;

                           return r+g+b;
}

}
```

MRRSemanticAnalyzer.java

```
/* MRRSemanticAnalyzer.java
 * Semantic checks for the MRRoboto language.
 * Coded by Adam Marczyk
 * Columbia University, Fall 2004
 */

import java.util.Stack;
import java.util.Vector;
import antlr.collections.AST;

/* The MRRSemanticAnalyzer class stores symbol tables, does type checking,
   and carries out various other aspects of semantic analysis on
   MRRoboto language code. */
public class MRRSemanticAnalyzer {

    /* A single symbol table holds all symbols of global scope.
       There are no global variables, so this consists solely of function names.
       This table is built up on our first pass through the program source. */
    private MRRSymbolTable global;

    /* A stack of symbol tables holds all symbols of local scope.
       This will consist of variables local to a particular block of code.
       This table is built up on our second pass through the source. */
    private Stack local;

    /* Temporary lists to hold the number and types of arguments
       a function expects. */
    private Vector argumentList;
    private String[] storedArgs;

    /* A list of valid types of variables. */
    private Vector varTypes;

    /* Flags to indicate where we currently are in the program:
        -whether we're in a loop (value = depth of nesting)
        -whether we're in a function (value = function name)
        -whether any further statements we encounter are unreachable
        -whether a function expects a return statement
        -whether a function that expects a return statement has one
     */
    private int inLoop;
    private String inFunction;
    private boolean unreachable;
    private boolean needsReturn;
    private boolean hasReturnStmt;

    public MRRSemanticAnalyzer() {
        global = new MRRSymbolTable();
        local = new Stack();
        argumentList = new Vector();
        varTypes = new Vector();

        inLoop = 0;
        unreachable = false;
        hasReturnStmt = false;
        needsReturn = false;
        inFunction = null;

        varTypes.add("int");
        varTypes.add("string");

        String[] oneInt = {"int"};
        String[] oneString = {"string"};
        String[] twoInts = {"int","int"};
        String[] arg4 = {"string","int","int"};
        String[] empty = new String[0];
```

```
        /* Add all names of library functions as globally scoped symbols. */
        addGlobalSymbol("mouseMove", new MRRSymbol("void", twoInts, true));
        addGlobalSymbol("mouseHold", new MRRSymbol("void", oneInt, true));
        addGlobalSymbol("click", new MRRSymbol("void", empty, true));
        addGlobalSymbol("moveAndClick", new MRRSymbol("void", twoInts, true));
        addGlobalSymbol("rightClick", new MRRSymbol("void", empty, true));
        addGlobalSymbol("moveAndRightClick", new MRRSymbol("void", twoInts, true));
        addGlobalSymbol("doubleClick", new MRRSymbol("void", empty, true));
        addGlobalSymbol("moveAndDoubleClick", new MRRSymbol("void", twoInts, true));
        addGlobalSymbol("wait", new MRRSymbol("void", oneInt, true));
        addGlobalSymbol("setWait", new MRRSymbol("void", oneInt, true));
        addGlobalSymbol("type", new MRRSymbol("void", oneString, true));
        addGlobalSymbol("press", new MRRSymbol("void", oneString, true));
        addGlobalSymbol("hold", new MRRSymbol("void", oneString, true));
        addGlobalSymbol("release", new MRRSymbol("void", oneString, true));
        addGlobalSymbol("releaseAll", new MRRSymbol("void", empty, true));
        addGlobalSymbol("exec", new MRRSymbol("void", oneString, true));
        addGlobalSymbol("print", new MRRSymbol("void", oneString, true));
        addGlobalSymbol("printInt", new MRRSymbol("void", oneInt, true));
        addGlobalSymbol("substring", new MRRSymbol("string", arg4, true));
        addGlobalSymbol("length", new MRRSymbol("int", oneString, true));
        addGlobalSymbol("pc", new MRRSymbol("string", twoInts, true));

        /* Also add all Java reserved words, so they can't be used as variables
           or function names. */
        addGlobalSymbol("abstract", "void");
        addGlobalSymbol("assert", "void");
        addGlobalSymbol("boolean", "void");
        addGlobalSymbol("break", "void");
        addGlobalSymbol("byte", "void");
        addGlobalSymbol("case", "void");
        addGlobalSymbol("catch", "void");
        addGlobalSymbol("char", "void");
        addGlobalSymbol("class", "void");
        addGlobalSymbol("const", "void");
        addGlobalSymbol("continue", "void");
        addGlobalSymbol("default", "void");
        addGlobalSymbol("do", "void");
        addGlobalSymbol("double", "void");
        addGlobalSymbol("else", "void");
        addGlobalSymbol("extends", "void");
        addGlobalSymbol("false", "void");
        addGlobalSymbol("final", "void");
        addGlobalSymbol("finally", "void");
        addGlobalSymbol("float", "void");
        addGlobalSymbol("for", "void");
        addGlobalSymbol("goto", "void");
        addGlobalSymbol("if", "void");
        addGlobalSymbol("implements", "void");
        addGlobalSymbol("import", "void");
        addGlobalSymbol("instanceof", "void");
        addGlobalSymbol("int", "void");
        addGlobalSymbol("interface", "void");
        addGlobalSymbol("long", "void");
        addGlobalSymbol("main", "void");
        addGlobalSymbol("native", "void");
        addGlobalSymbol("new", "void");
        addGlobalSymbol("null", "void");
        addGlobalSymbol("package", "void");
        addGlobalSymbol("private", "void");
        addGlobalSymbol("protected", "void");
        addGlobalSymbol("public", "void");
        addGlobalSymbol("return", "void");
        addGlobalSymbol("short", "void");
        addGlobalSymbol("static", "void");
        addGlobalSymbol("strictfp", "void");
        addGlobalSymbol("super", "void");
        addGlobalSymbol("switch", "void");
        addGlobalSymbol("synchronized", "void");
        addGlobalSymbol("this", "void");
        addGlobalSymbol("throw", "void");
```

```
        addGlobalSymbol("throws", "void");
        addGlobalSymbol("transient", "void");
        addGlobalSymbol("true", "void");
        addGlobalSymbol("try", "void");
        addGlobalSymbol("void", "void");
        addGlobalSymbol("volatile", "void");
        addGlobalSymbol("while", "void");

        /* Also reserve MRRoboto keywords. */
        addGlobalSymbol("to", "void");
        addGlobalSymbol("step", "void");
        addGlobalSymbol("end", "void");
        addGlobalSymbol("procedure", "void");
        addGlobalSymbol("error", "void");
        addGlobalSymbol("mrRobotoLibrary", "void");
    }

    /* Statements beyond this point are unreachable, unless we exit
       a level of scope. Called after break, continue and return directives. */
    public void setUnreachable() { unreachable = true; }

    public boolean unreachable() { return unreachable; }

    /* Keep track of whether we're inside a while or for loop.
       Used to decide whether a break or continue directive is valid. */
    public void beginLoop() { inLoop++; }

    public void exitLoop() { inLoop++; }

    public boolean inLoop() { return (inLoop > 0); }

    /* Keep track of whether we're inside a function, and if so, which one.
       Used to decide whether a return directive is valid. */
    public void enterFunction(String name) { inFunction = name; }

    public void exitFunction() {
        inFunction = null;
        needsReturn = false;
        hasReturnStmt = false;
    }

    public String inFunction() { return inFunction; }

    /* Keep track of whether a function needs a return statement
       (i.e., a non-void function), and if so, whether it has one. */
    public void setNeedsReturn() { needsReturn = true; }

    public boolean getNeedsReturn() { return needsReturn; }

    public void setReturn() { hasReturnStmt = true; }

    public void clearReturn() { hasReturnStmt = false; }

    public boolean hasReturnStmt() { return hasReturnStmt; }

    /* As we walk the subtree containing arguments provided to a
       function, store them in a list. Used to keep track
       of whether the arguments in a function call match those
       in the function definition. */
    public void storeArguments(String[] list) { storedArgs = list; }

    public String[] getStoredArguments() {
        if(storedArgs == null) return new String[0];
        String[] tmp = storedArgs;
        storedArgs = null;
        return tmp;
    }

    /* Functions can only be of global scope, so that's the only
       symbol table we need to check. */
    public boolean matchFuncArguments(String name, String[] args) {
```

```
        return global.matchFuncArguments(name, args);
    }

    /* Determine whether a particular type is defined for this language. */
    public boolean isValidType(String t) {
        if(varTypes.indexOf(t) != -1) return true;
        return false;
    }

    /* Add a symbol to global scope. Return 1 if the add was successful,
       -1 if this symbol was already defined in the global scope. */
    public int addGlobalSymbol(String name, MRRSymbol s) {
        if(global.isDefined(name)) return -1;
        global.addSymbol(name, s);
        return 1;
    }

    /* Polymorphic version of the above function. */
    public int addGlobalSymbol(String name, String s) {
        return addGlobalSymbol(name, new MRRSymbol(s));
    }

    /* Add a symbol to the current local scope (i.e., the symbol table
       on top of the stack). Return 1 if the add was successful,
       -1 if this symbol was already defined in any accessible scope. */
    public int addSymbol(String name, MRRSymbol s) {
        if(isDefined(name)) return -1;
        MRRSymbolTable st = (MRRSymbolTable)local.peek();
        st.addSymbol(name, s);
        return 1;
    }

    /* Polymorphic version of the above function. */
    public int addSymbol(String name, String s) {
        return addSymbol(name, new MRRSymbol(s));
    }

    /* Check whether a symbol is already defined within any accessible scope.
       The first thing we have to check is whether it's defined in the
       global scope. If not, we have to check whether it's defined in the current
       local scope or any accessible scope enclosing the local scope. */
    public boolean isDefined(String name) {
        MRRSymbolTable tmp;
        if(global.isDefined(name)) return true;
        for(int i = 0; i < local.size(); i++) {
            tmp = (MRRSymbolTable)local.get(i);
            if(tmp.isDefined(name)) return true;
        }
        return false;
    }

    /* Check whether a symbol is a function (if not, it's a variable). */
    public boolean symbolIsFunc(String name) {
        if(!global.isDefined(name)) return false;
        return global.symbolIsFunc(name);
    }

    /* Check whether a symbol is a built-in language function. */
    public boolean symbolIsLibFunc(String name) {
        if(!global.isDefined(name)) return false;
        if(global.symbolIsFunc(name) == false) return false;
        return global.symbolIsLibFunc(name);
    }

    /* Enter a new level of scope. */
    public void enterScope() {
        MRRSymbolTable st = new MRRSymbolTable();
        local.push(st);
    }

    /* Leave a level of scope. Symbols defined within it disappear. */
```

```
    public void leaveScope() {
        local.pop();
        unreachable = false;
    }

    /* Determine the type of an accessible symbol. Return null if the
       argument is not a defined symbol. */
    public String lookupType(String name, boolean nodeIsFnCall) {
        MRRSymbolTable tmp;
        if(global.isDefined(name) && global.symbolIsFunc(name) == nodeIsFnCall)
            return global.getSymbolType(name);
        for(int i = 0; i < local.size(); i++) {
            tmp = (MRRSymbolTable)local.get(i);
            if(tmp.isDefined(name) && tmp.symbolIsFunc(name) == nodeIsFnCall)
                return tmp.getSymbolType(name);
        }
        return "void";
    }

    public String lookupType(String name) {
        return lookupType(name, false);
    }

    /* Determine the type of a node in the abstract syntax tree.
       This node may be either an operator, a literal, or a variable.
       Notice that no recursion is necessary here, as all decisions
       can be made locally. */
    public String inferType(AST rootNode, boolean nodeIsFnCall) {
        String text = rootNode.getText();
        String type;
        if(text.equals("+") || text.equals("-") || text.equals("*") || text.equals("/"))
            type = "int";
        else if(text.equals("<") || text.equals(">") || text.equals("<=")
                || text.equals(">=") || text.equals("==") || text.equals("<>"))
            type = "comparator";
        else if(text.equals("&&") || text.equals("||"))
            type = "boolean";
        else if(text.equals("++"))
            type = "string";
        else if(text.equals("="))
            type = inferType(rootNode.getFirstChild());
            //i.e., an assignment operator has the same type as its lvalue

        else if(text.equals("PROCEDURE_CALL"))
            type = inferType(rootNode.getFirstChild(), true);
            //also, a function call has the same type as the function's return type

        /* Otherwise, this node is either a variable or a literal; in either case,
           we can safely determine its type. */
        else if(text.charAt(0) == '"')
            type = "string";
        else if(isDefined(text))
            type = lookupType(text, nodeIsFnCall);
        else {
            try {
                Integer.parseInt(text);
                type = "int";
            } catch(NumberFormatException e) {
                type = "void";
            }
        }
        return type;
    }

    public String inferType(AST rootNode) {
        return inferType(rootNode, false);
    }

    /* Ensure that the actual type of a node matches what we think it is. */
    public boolean verifyType(AST node, String type) {
        return inferType(node).equals(type);
```

```
    }

    /* Verify that the types of symbols in an expression match. This is
       done via an examination of the expression subtree; we must ensure
       that the operator (the root of the tree) and its children (its operands)
       all have the same type. Notice that no recursion is necessary here,
       as all decisions can be made locally. */
    public String checkTypes(AST rootOperator, AST child1, AST child2) {
        String op, c1, c2;
        String opType, c1Type, c2Type;

        op = rootOperator.getText();
        c1 = child1.getText();
        if(c1.equals("PROCEDURE_CALL")) c1 = child1.getFirstChild().getText() + "()";
        c2 = child2.getText();
        if(c2.equals("PROCEDURE_CALL")) c2 = child2.getFirstChild().getText() + "()";

        opType = inferType(rootOperator);
        c1Type = inferType(child1);
        c2Type = inferType(child2);

        //Void type matches nothing, not even itself.
        if(opType.equals("void") || c1Type.equals("void") || c2Type.equals("void"))
        return "Type mismatch between "+op+" ("+opType+"), "+c1+" ("+c1Type+"), "+c2+"
("+c2Type+")";

        /* null return value means our types check out. */
        if(opType.equals("comparator") && c1Type.equals(c2Type) &&
            (c1Type.equals("int") || c1Type.equals("string"))) return null;
        else if(opType.equals("boolean") &&
            (c1Type.equals("comparator") || c1Type.equals("boolean")) &&
            (c2Type.equals("comparator") || c2Type.equals("boolean"))) return null;
        else if(opType.equals(c1Type) && c1Type.equals(c2Type)) return null;

        /* Otherwise return a string with detailed error information. */
        return "Type mismatch between "+op+" ("+opType+"), "+c1+" ("+c1Type+"), "+c2+"
("+c2Type+")";
    }

    /* Store the type of the next argument taken by the current function. */
    public void addArgument(String type) {
        argumentList.add(type);
    }

    /* We've processed all the arguments. Put this function on the symbol table. */
    public void stopStoringArguments(String funcName, String funcType) {
        String[] tmp = new String[argumentList.size()];
        for(int i = 0; i < argumentList.size(); i++)
            tmp[i] = (String)argumentList.get(i);
        addGlobalSymbol(funcName, new MRRSymbol(funcType, tmp));
        argumentList.clear();
    }

    /* Implementation Notes:
     * Building the symbol table requires two passes through the AST.
     * Our grammar looks something like this:
     *  program -> main_block (function_declaration)*
     * and so our AST will look like this:
     *  program
     *      -> main_block
     *      -> function1
     *      -> function2
     *      etc.
     * Our first pass through the tree is trivial: do a one-level breadth-first search
     * from the root, capture all function declarations, and put the names of
     * those functions in the global symbol table.
     * Our second pass through the tree proceeds in normal depth-first fashion:
     * each time we enter a scope, push a new level on the stack, add symbols
     * defined in that scope to the top symbol table on the stack, and pop the stack
     * when we leave that scope.
     * The local symbol table is different for each block; the global symbol table
```

```
     * never changes.
     * A symbol is considered to be defined if it is defined in the global scope
     * or in the local scope; therefore we cannot, for example, have a variable
     * with the same name as a function. We can, however, have two functions
     * each of which has a local variable with the same name.
     */

}
```

MRRSymbol.java

```
/* MRRSymbol.java
 * Part of the semantic checker for the MRRoboto language.
 * Coded by Adam Marczyk
 * Columbia University, Fall 2004
 */

/* The MRRSymbol class represents a single symbol in a symbol table. */
public class MRRSymbol {

    /* The type of this symbol (integer, string, etc). */
    private String type;

    /* If this symbol is a function, args.length is the number of arguments it
        takes, and args[n] is the type of its nth argument. */
    private String[] args;

    /* True if this symbol is a function. */
    private boolean isFunc;

    /* True if this symbol is a predefined library function. */
    private boolean isLibFunc;

    /* Basic constructor for a non-function symbol. */
    public MRRSymbol(String t) {
        type = t;
        isFunc = false;
    }

    /* Basic constructors for a function symbol. */
    public MRRSymbol(String t, String[] a) {
        type = t;
        args = a;
        isFunc = true;
        isLibFunc = false;
    }

    public MRRSymbol(String t, String[] a, boolean x) {
        type = t;
        args = a;
        isFunc = true;
        isLibFunc = x;
    }

    public String getType() { return type; }

    public boolean isFunction() { return isFunc; }

    public boolean isLibFunction() { return isLibFunc; }

    public int getNumArgs() { return args.length; }

    public String getArgType(int argNum) { return args[argNum]; }
}
```

MRRSymbolTable.java

```
/* MRRSymbolTable.java
 * Part of the semantic checker for the MRRoboto language.
 * Coded by Adam Marczyk
 * Columbia University, Fall 2004
 */

import java.util.HashMap;

/* The MRRSymbolTable class represents a single scope within the entire program's
   symbol table. */
public class MRRSymbolTable {

    private HashMap table;

    public MRRSymbolTable() {
        table = new HashMap();
    }

    /* Add a symbol to this scope, indexed by its name. */
    public void addSymbol(String symbolName, MRRSymbol s) {
        table.put(symbolName, s);
    }

    /* Determine whether a given symbol is defined in this scope. */
    public boolean isDefined(String symbolName) {
        return table.containsKey(symbolName);
    }

    /* Return the type of a given symbol, indexed by its name. */
    public String getSymbolType(String symbolName) {
        if(!isDefined(symbolName)) return null;
        return ((MRRSymbol)table.get(symbolName)).getType();
    }

    /* Return whether a given symbol is a function. */
    public boolean symbolIsFunc(String symbolName) {
        if(!isDefined(symbolName)) return false;
        return ((MRRSymbol)table.get(symbolName)).isFunction();
    }

    /* Return whether a given symbol is a library function. */
    public boolean symbolIsLibFunc(String symbolName) {
        if(!isDefined(symbolName)) return false;
        return ((MRRSymbol)table.get(symbolName)).isLibFunction();
    }

    /* Determine whether a provided list of argument types matches
       the actual arguments defined for a given symbol that
       happens to be a function. */
    public boolean matchFuncArguments(String func, String[] args) {
        if(!isDefined(func) || !symbolIsFunc(func)) return false;
        MRRSymbol tmp = (MRRSymbol)table.get(func);
        if(args.length != tmp.getNumArgs()) return false;
        for(int i = 0; i < args.length; i++) {
            if(!args[i].equals(tmp.getArgType(i))) return false;
        }
        return true;
    }

}
```

Appendix D: RobotoXY

RobotoXY is a helper program to aid in developing MRRoboto programs. It runs as a Windows application and displays a constant reading of the screen coordinates and pixel color of the current mouse position.

RobotoXY.exe Program

RobotoXY.vbs

```
'*** RobotoXY.exe
'*** Coded by Jason Kopylec



Option Explicit
      Private Const SWP_NOMOVE = 2
      Private Const SWP_NOSIZE = 1
      Private Const FLAGS = SWP_NOMOVE Or SWP_NOSIZE
      Private Const HWND_TOPMOST = -1
      Private Const HWND_NOTOPMOST = -2

      Private Type POINTAPI
    x As Long
    y As Long
End Type
Private Declare Function GetCursorPos Lib "user32" (lpPoint As POINTAPI) As Long
Private Declare Function ScreenToClient Lib "user32" (ByVal hwnd As Long, _
    lpPoint As POINTAPI) As Long

Private Declare Function GetPixel Lib "gdi32" (ByVal hdc As Long, _
    ByVal x As Long, ByVal y As Long) As Long

Private Declare Function GetWindowDC Lib "user32" (ByVal hwnd As Long) _
        As Long


Private Declare Function SetWindowPos Lib "user32" (ByVal hwnd As Long, ByVal
hWndInsertAfter As Long, ByVal x As Long, ByVal y As Long, ByVal cx As Long, ByVal cy As
Long, ByVal wFlags As Long) As Long

      Public Function SetTopMostWindow(hwnd As Long, Topmost As Boolean) _
         As Long

         If Topmost = True Then 'Make the window topmost
            SetTopMostWindow = SetWindowPos(hwnd, HWND_TOPMOST, 0, 0, 0, _
               0, FLAGS)
         Else
            SetTopMostWindow = SetWindowPos(hwnd, HWND_NOTOPMOST, 0, 0, _
               0, 0, FLAGS)
            SetTopMostWindow = False
         End If
      End Function



' Get mouse X coordinates in pixels
'
' If a window handle is passed, the result is relative to the client area
' of that window, otherwise the result is relative to the screen

Function MouseX(Optional ByVal hwnd As Long) As Long
    Dim lpPoint As POINTAPI
```

```
    GetCursorPos lpPoint
    If hwnd Then ScreenToClient hwnd, lpPoint
    MouseX = lpPoint.x
End Function

' Get mouse Y coordinates in pixels
'
' If a window handle is passed, the result is relative to the client area
' of that window, otherwise the result is relative to the screen

Function MouseY(Optional ByVal hwnd As Long) As Long
    Dim lpPoint As POINTAPI
    GetCursorPos lpPoint
    If hwnd Then ScreenToClient hwnd, lpPoint
    MouseY = lpPoint.y
End Function

Private Sub Form_Load()
    Dim lR As Long
    lR = SetTopMostWindow(XY.hwnd, True)
    Label1.Caption = "(" & MouseX & ", " & MouseY & ")"
End Sub



Function Pixel_Color() As String
    'Calculate the pixel color of the current mouse position
    Dim lColor As Long
    Dim sTmp As String

    lColor = GetPixel(GetWindowDC(0), MouseX, MouseY)
    sTmp = Right$("000000" & Hex(lColor), 6)
    Pixel_Color = "RGB:" & Right$(sTmp, 2) & "" & Mid$(sTmp, 3, 2) & "" & Left$(sTmp, 2)
End Function

Private Sub Timer1_Timer()
    'Display current mouse position/color
    Label1.Caption = "(" & MouseX & ", " & MouseY & ") " & Pixel_Color()
End Sub
```