Sim2D Final Report

COMS W4115 Programming Languages and Translators

David Suess

dcs2136@columbia.edu

August 11, 2008

Contents

1 Introduction

Sim2D is a language designed for simulation of the movement of objects on a 2 dimensional map. A programmer may define objects and their behavior. The interpreter will present the simulation on a 2 dimensional map for observation of the interaction of the objects. Possible applications of this language would be to simulate the traffic of air, land and sea vehicles, or the interactions of animals in an environment. The language will be simplified and interpreted for quick development and debugging.

1.1 Design

Simulations generally consist of a loop in which operations are performed. Therefore, to simplify the language, the loop for this language will be implied. A user will construct objects and their velocity as desired. A user must define a starting position, a speed and heading. For stationary objects, a user can specify a speed of zero. The simulation will automatically move the objects with nonzero speed in the heading specified. A user can then construct rules to govern changes to the movement of the objects. Any number of user customizable fields can be defined for object for use in the custom behavior of the objects. A 2 dimensional map will be rendered utilizing java swing.

1.2 Language Description

The language will look familiar to a C programmer, but it is simplified for what is needed for a 2D simulation. There are no provisions for user defined loops, or pointers. Function definition is limited to the capability of the *rule* keyword. The only basic data types are float, integer and boolean. The objects are predefined struct-like types that have built in variables that are required for the simulation. Examples of these are x and y coordinates, speed and heading. A program will generally consist of one or more object definition blocks followed by one or more *rule* statements to define custom behavior. Object definition statements are interpreted immediately and only once unless they exist within a rule. *rule* statements are executed at every iteration of the simulation loop. User defined functions may also be defined globally for use within any rule. The simulation automatically updates the x and y coordinates of an object after each iteration based on it's speed, heading and iteration rate of the simulation. The iteration rate of the simulation is determined by a command line argument to the interpreter. For further details on the structure of the language, see the Language Manual section.

2 Language Tutorial

2.1 Setup

Sim2d programs consist of a single source code file. The code may be executed with the sim2d interpreter which was written in java with jdk 1.6.0_05 and Antlr 2.7.7. Ensure both are installed and your CLASSPATH environment is set to point to antlr.jar and sim2d.jar. Here is an example of a CLASSPATH setting:

```
CLASSPATH=.;c:/Java/jdk1.6.0_05/lib;c:/Java/jdk1.6.0_05/jre/lib/rt.jar;c:/cygwin/
usr/local/lib/antlr.jar;c:/cygwin/home/proj/sim2d
```

2.2 Coding a Program

A sim2d source code file consists 3 types of definitions: One or more object definitions, zero or more rule definitions corresponding to the object definitions, and zero or more user defined functions.

2.2.1 Object Definition

An object definition looks like this:

```
object dog {
        x <- 50.0;
        y <- 50.0;
        speed <- 4.0;
        heading <- 45.0;
        visible <- true;
        hungry <- true;
}
```

The x and y fields represent the position of the object. All objects have x and y fields built in and initialized to zero, but a user may define their own starting position in the object definition as shown. The x and y fields are always of type float. The speed and heading fields are exactly what you would think, and are also built in and initialized to zero unless otherwise stated. The speed and heading fields are always of type float. The visible field is the last obligatory field and if of type boolean. It represents whether or not the object is visible on the graphical display.

2.2.2 Rule Definition

A rule definition looks like this:

```
rule dog {
        if (hungry) {
                heading <- bearing_to(cat);
                if (speed > dog.distance_to(cat) = 0) {
                        cat.speed <- 0.0;
                        cat.visible <- false;
                        speed <- 0.0;
                        hungry <- false;
                }
        }
}
```

The name of the rule identifies which object it operates on. All symbols within the rule may refer to the object that it belongs to without referring to the name of the object (as shown with the speed, heading and hungry fields which all refer to the dog object's fields). Fields of other objects may be referenced by explicitly stating the name of the object followed by the "dot" operator and the name of the field (as shown with cat.speed and cat.visible). Assignments may be made to any field with the "<-" operator. Internal and user defined functions may be called from within the rule (as shown with the internal function distance_to). Flow of the rule can be controlled with "if else" statements in the C language syntax.

2.3 Running a Program

Execute the program with the command line:

```
java sim2d <sim2d source code file>
```

A 600x600 pixel java swing jpanel will appear and begin simulating the movement of the objects defined in the source code file. The program will execute indefinitely or until the user closes the jpanel.

The sim2d interpreter supports the following command line options:

-h	Outputs the command line options
-s	Suppress map display
-i [number]	Quit after [number] iterations
-r [delay]	Time delay between iterations in msecs
-g	Show graphical AST
-t	Show textual AST
-d	Dump symbol table after each iteration
-v	Output verbose debugging text

3 Language Manual

3.1 Lexical Conventions

3.1.1 Comments

Single line comments begin with the // characters and end at the end of line. Multi line comments begin with the characters /* and end with the */ characters.

3.1.2 Identifiers

Identifiers represent the names of internal or user defined variables and internal or user defined functions.

Identifiers consist of one or more characters and must begin with one of the characters a to z upper or lower case. After the first character, the identifier may consist of characters upper or lower case a to z, digits 0 to 9 and the underscore character. Recognition of an identifier is case sensitive.

3.1.3 Keywords

Keywords are reserved words that may not be used as identifiers. They are:

rule	**object**	**speed**	**heading**	**visible**	**distance_to**
bearing_to	**x**	**y**	**if**	**else**	**true**
false	**func**				

3.1.4 Constants

Integer literals take the form of a sequence of digits (characters 0 to 9) without the presence of whitespace or a dot character. All integers are considered base 10.

Floating point literals are of the form of a sequence of digits followed one of the following:

a) An optional decimal character followed by an optional sequence of digits indicating a fractional part
b) An optional ‘e’ or ‘E’ character followed by an integer literal indicating a base 10 exponent

“optional” here means that option a) and option b) do not both need to be present, but one of the two must be present in order for the numeric literal to qualify as a floating point literal.

Booleans can be assigned and compared with the *true* and *false* keywords.

3.1.5 Operators

Operators are used for expressions, assignments and object dereferencing.

.	Dereference fields of an object
= , ! =	Boolean comparison
= , > , < , >= , <= , ! =	Integer comparison
<-	Assignment for float, integer and boolean types
and , or	Boolean logic
- , + , * , /	Arithmetic operations for float and integer types
-	Unary negative operator for integer and float types

3.1.6 Separators

Separators are used to distinguish arguments, expressions, statements, object blocks, rule blocks and function blocks.

{ } ( ) ;

3.1.7 Whitespace

Whitespace characters are space, tab, newline, carriage return, linefeed. No whitespace characters will have any significance to the language other than to separate the tokens of the language and to improve the readability of the source code.

3.2 Types

Supported types will be integer, float, boolean and object. Integers are 32 bit integers. Floats are 32 bit floating point numbers.

3.2.1 Object

The object type is a data structure that may take on as many user defined fields of **integer**, **float**, and **boolean** types desired. In addition, an object will always contain the internal float fields **x**, **y**, **speed**, **heading**, and the **boolean** field **visible**.

x, y	The 2 dimensional location of the object in the simulation.
speed, heading	The velocity of the object.
visible	Indicates whether or not the object is visible on the graphical representation of the simulation.

Objects contain the built-in functions **distance_to** and **bearing_to**:

distance_to*(object-identifier)*

returns an float value indicating the distance to the passed in object

bearing_to*(object-identifier)*

returns an float value indicating the bearing to the passed in object

3.3 Expressions

Expressions can be identifiers, constants or combinations of them joined by operators. Parentheses may be used to specify precedence in the ordering of evaluation of the operations.

3.3.1 Primary Expressions

A primary expression consists of a constant, identifier or parenthesized expression. A parenthesized expression is any expression surrounded by parenthesis, the purpose of which are to prioritize the order of operations.

3.3.2 Unary Expressions

The unary operator – inverts the sign of an integer or floating point value. That is, the value is multiplied by -1 for integer types and -1.0 for float types.

3.3.3 Logical Expressions

The logical operators **and** and **or** are binary operators that require the left and right operands to be of the boolean type. The result of an **and** is a boolean **true** if both

operands are true, and false otherwise. The result of an **or** is a boolean **false** if both operands are false, and true otherwise. Both operands are always evaluated.

3.3.4 Relational Expressions

The relational operators =, !=, <, >, <=, >= are binary operators that require the left and right operands to be of the same type and integer or float type. The result is a boolean value whether or not the left operand is (with respect to the right operand) equal to, not equal to, less than, greater than, less than or equal to, greater than or equal to respectively.

3.3.5 Arithmetic Expressions

The arithmetic operators *, /, -, + are binary operators that require the left and right operands to be of the same type and integer or float type. The result is an integer or float value depending on the type of the operands. The operations are multiplication, division, subtraction and addition respectively.

3.3.6 Object Dereference

An expression can refer to a field within an object by using the . operator in the following form:

identifier.identifier

The first identifier is the name of the object, the second identifier is the name of the field within the object. The result of this expression is a value of the type and value of the field within the object.

3.3.7 Function Call

A function call is of the form:

identifier (arguments);

The first identifier is the function name. The arguments consist of zero or more expressions separated by commas. The function name and number of arguments must match a previously defined function in the top-level of the program. By top-level we mean that it is not within any other construct (inside braces {})

3.3.8 Operator Precedence

The operators have the precedence order from low to high as follows:

or
and
=
!=
<
>
<=

>=
+
- (binary)
*
/
- (unary)
.

3.4 Statements

3.4.1 Assignment Statement

An assignment statement has the form:

identifier <- expression;

Where the identifier may be a field of an object and expression must result in a matching data type. An assignment statement may appear inside an object definition statement, inside a rule statement, inside a function definition statement or inside a conditional statement.

3.4.2 Conditional Statement

A conditional is of the form:

if *(expression)* {
    Zero or more assignment or conditional statements;
} **else** {
    Zero or more assignment or conditional statements;
}

The else clause here is optional.

A conditional statement may appear within a rule or function definition statement, but not an object definition statement.

3.4.3 Rule Statement

A rule is a way to give behavior to an object. A rule is of the form:

rule *identifier* {

    Zero or more assignment or conditional statements;

}

A rule statement may only appear at the top level of the program. By top-level we mean that it is not within any other construct (inside braces {})

Example:

```
rule united432 {
        heading <- bearing_to(KPHL);
}
```

The fields of the object for which the rule is defined are implicit and need not be dereferenced in the body of the rule.

Rule blocks are executed once for every iteration of the simulation loop.

3.4.4 Object Definition Statement

An object is defined and created with the object keyword:

object *identifier {*
        Zero or more assignment statements;
}

The type of a field is specified by the type of the literal on the right side of the assignment statement (Integer literal, float literal or boolean literal)

An object statement may only appear at the top level of the program. By top-level we mean that it is not within any other construct (inside braces {})

3.4.5 Internal Variables

All objects contain the float fields **x, y, speed, heading,** and the boolean field **visible**. The float fields are initialized to 0.0 if they are not explicitly defined in the object. The boolean field is initialized to true if they are not explicitly defined in the object.

Example:

```
object  KPHL {
        x <- 250;       // internal variable, always float despite int literal
        y <- 250;       // internal variable, always float despite int literal
        visible <- true;
        user_var1 <- 1;         // integer type
        user_var2 <- 1.0;       // float type
        // the heading and speed internal variables are not explicitly
        // initialized, so they will default to 0.0
}
```

3.4.6 User Function Definition

A user defined function may be made with the following keyword

func *identifier (variable number of identifiers separated by commas)* {
        Zero or more assignment or conditional statements;
}

User defined functions do not return a value and all arguments are passed by reference. The types of the variables in the arguments are determined by the caller. The caller must know the intended use of the arguments by the function, otherwise a semantic error will result.

A user function definition statement may only appear at the top level of the program. By top-level we mean that it is not within any other construct (inside braces {})

4 Project Plan

The big picture for the development plan for this project was to rapidly develop a simple version of the interpreter and to iteratively improve the interpreter is small testable steps. The steps are shown in more detail in the Project Timeline section.

4.1 Style Guide

- Source code for sim2d language files is .s2d
- Indent 3 spaces
- Opening braces should appear on the same line as the if, class or method they belong to
- Class names should be capitalized with each word capitalized, no underscores
- Member variables should be all lower case, using underscores for multiple words
- Methods should not be capitalized, each following words should be capitalized, no underscores
- All lines under 80 characters, if more needed, indent following lines
- Errors detected in the code being interpreted should throw the sim2dException with text describing the problem. Handling of the sim2dException will be in the try catch block of the main simulation loop

4.2 Project Timeline

- May 21 First day of class
- May 26-31 Brainstorm language ideas
- June 1-4 Write Proposal
- June 5-6 Install java, antlr 2, cygwin, vim
- June 7-9 Familiarize with antlr syntax and examples
- June 10-12 Develop 2d jpanel display capable of displaying dots and refresh
- June 12 Establish version control
- June 13-15 Write lexer in antlr .g file
- June 16-20 Write Language Reference Manual
- June 16-26 Write parser in antlr .g file
- June 20 Language Reference Manual Due

- June 26-30 Streamline Makefile, create automated test script

- July 1-5 Develop walker for object definition, assignments, int type
- July 6-10 Add basic interpreter and symbol table, update object def walker
- July 11-18 Add basic expression walker for if statements, rule walker
- July 19-25 Expand expression operators: > <, and or etc
- July 26-28 Add more types, bool, float
- July 29-30 Add built in and user defined functions
- Aug 4 - Add more test cases, contemplate nice-to-have's
- Aug 11 Project Due

4.3 Development Environment

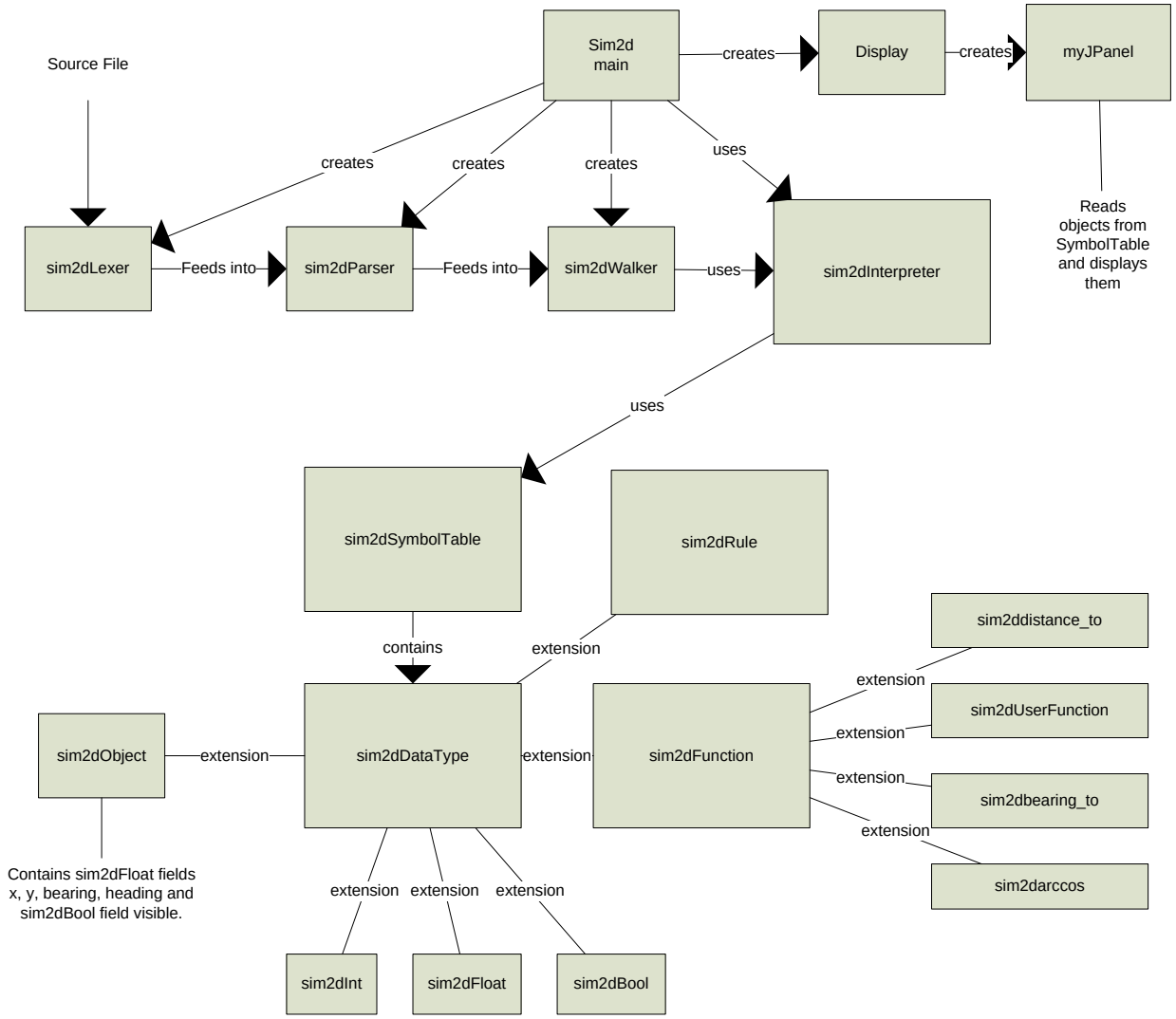
Sim2d is written in java and was developed on an i686 processor running cygwin within Windows XP, java jdk 1.6.0_05, and antlr 2.7.7. The editor used was vim. A makefile was created to compile the antlr g files and then compile the java code. Code and testing iterations consisted of editing code in vim, running the makefile on the command line, and then running the runalltests.csh (described in the Test Plan section).

4.4 Project Log

Revision	Description	Date	Author
r26	Fix bug in if body, cleaned up dog and cat test, added final report.	Aug 9, 2008	DavidSuess
r25	Functions, user functions, tests etc	Jul 30, 2008	DavidSuess
r24	added 'and' 'or' and 'not' and log class. added -d for symbol dump and -l for logging.	Jul 25, 2008	DavidSuess
r23	working operators: > < <= >= = !=	Jul 25, 2008	DavidSuess
r22	better example of what an if can do	Jul 25, 2008	DavidSuess
r21	object definition, int's, bool's, basic assignment, basic rules, display works. test_2_objects.s2d works	Jul 25, 2008	DavidSuess
r20	Basic walker for object and rule	Jul 18, 2008	DavidSuess
r19	dog and cat test case	Jul 18, 2008	DavidSuess
r18	Latest before walker work	Jul 18, 2008	DavidSuess
r17	added LRM	Jun 26, 2008	DavidSuess
r16	adding test files	Jun 26, 2008	DavidSuess
r15	lexer and parser are working	Jun 26, 2008	DavidSuess
r14	Add WindowExiter.java, ensure .jar file is complete (but does not include antlr.jar)	Jun 17, 2008	DavidSuess
r13	Proposal updates; can output expression tree	Jun 17, 2008	DavidSuess
r12	Added some changes to prop, removing rover, landmark, create, adding <-. More changes still needed.	Jun 16, 2008	DavidSuess
r11	Add original proposal.doc	Jun 16, 2008	DavidSuess
r10	no comment	Jun 16, 2008	DavidSuess
r9	Added underscores to proposal filename	Jun 16, 2008	DavidSuess
r8	Some (not all) proposal changes from prof advice, still needs updating	Jun 16, 2008	DavidSuess
r7	Added sim2d.java and antlr generated code	Jun 16, 2008	DavidSuess
r6	Added sim2d.g and proposal	Jun 16, 2008	DavidSuess
r5	no comment	Jun 12, 2008	DavidSuess

5 Architectural Design

5.1 Block Diagram



5.2 Interfaces and Flow

The sim2d main class first creates instances of sim2dLexer and sim2dParser. The lexer class extracts the tokens from the input file and the parser generates an abstract syntax tree. The main class then creates sim2dWalker and sim2dInterpreter instances and calls the walker file() method which walks the object definitions, user defined functions, and rules. As the file() walk occurs, objects and their fields are added to the symbol table via the interpreter as well as the user defined functions and rules. The next step is to create the graphical display and then begin the simulation loop. The simulation loop finds all objects within the symbol table and then finds the corresponding rules with the same name. If a rule exists, then the object's fields are added to a new scope on the symbol table and the walker expr() method is executed on the rule's body. After all rules

have been run, the x and y coordinates are computed using the speed, heading and iteration rate of the simulation. The display class then refreshes itself by updating with the current x and y coordinates and visible flag of all objects. The loop then repeats.

The sim2dException class provides a means for any component to indicate an error in the program. Any component may throw this exception and the main class will catch it and display the error to the standard output.

The sim2dWalker class provides methods for walking the abstract syntax tree. These methods are called by the sim2d main class.

The sim2dSymbolTable class provides put, get, enterScope, leaveScope, and dump methods. The put method is used by the interpreter to add a symbol to the current scope. The enterScope and leaveScope methods create and remove a symbol table on the top of the stack. When a new scope is entered, all put operations put the symbol into the current scope which is the scope on the top of the stack. The get method searches for a symbol name in the symbol table by first checking the scope on the top of the stack, and then the next highest and so on. The dump method dumps the symbol table to standard output and is used for debugging and automated testing.

The sim2dInterpreter class provides a layer of indirection between the walker and the symbol table. This is useful for the unique behavior of the object data type. When a rule or function is called, the interpreter handles the adding of fields and arguments into the new scope that is created.

Sim2dDataType is the base class of all data types, functions, object and rules. The sim2dDataType defines stubs for all the methods that all derived classes can define meaningful actions for, such as add, subtract, etc. The stubs will initiate an sim2dException so that if a derived class does not implement it, it is an error for a program to try to use it.

All components were created by me, David Suess.

6 Test Plan

The interpreter was tested with a shell script, runalltests.csh, that runs the interpreter on a battery of sim2d source code files. The shell script utilizes the command line options –s (to suppress the graphical display, -i (to terminate the simulation after a set number of iterations), -d (to dump the entire symbol table after each iteration of the simulation loop), -r (to eliminate the delay between iterations for fast execution) and –t (to output the abstract syntax tree. The shell script saves the output of each test and diff's it with a known good output. If the output matches, then the test passes, otherwise, fail is printed. Each test is written with the intention to test a single concept of the language, such as object definition, and if statement, addition, subtraction, etc. The tests are executed in order of increasing complexity in order to ease identification of a problem

with the language. The name of the test source code file identifies the key function that is meant to be tested.

6.1 List of tests

test_1_obj_1_field
test_1_obj_2_fields
test_1_obj_3_fields
test_1_obj_4_fields
test_1_obj_5_fields
test_float
test_uminus
test_hdg_0_speed_2
test_hdg_135_speed_2
test_hdg_180_speed_2
test_hdg_225_speed_2
test_hdg_270_speed_2
test_hdg_315_speed_2
test_hdg_360_speed_2
test_hdg_45_speed_2
test_hdg_90_speed_2
test_rule_w_assign
test_rule_w_if
test_if_mult_stmts
test_object_def
test_bearing_to
test_deref
test_distance_to
test_dog_and_cat
test_orbit

6.2 Example Code For Dog and Cat Chase Simulation

```
// This program will simulate a dog chasing a cat.  The dog will make a
// bee line for the cat.  The cat will run in the opposite direction when the
// dog enters a certain distance.
object dog {
        x <- 50.0;
        y <- 50.0;
        speed <- 4.0;
        heading <- 45.0;
                visible <- true;
                hungry <- true;
}
object cat {
        x <- 150.0;
```

```
        y <- 150.0;
        speed <- 0.0;
        heading <- 45.0;
        visible <- true;
}

rule cat {
        if ((visible = true) and (cat.distance_to(dog) < 20.0)) {
                // cat will run from dog in opposite direction
                heading <- cat.bearing_to(dog) + 180.0;
                speed <- 3.0;
        }
}

rule dog {
        if (hungry) {
                heading <- bearing_to(cat);
                if (speed > dog.distance_to(cat) = 0) {
                        cat.speed <- 0.0;
                        cat.visible <- false;
                        speed <- 0.0;
                        hungry <- false;
                }
        }
}
```

6.3 Example code for planet orbit simulation

```
// This is a simplistic example of several planets orbiting around the sun
// The distances and orbital velocities are fictitious, but could be made
// realistic with some pre calculation.
object sun {
        x <- 300;
        y <- 300;
        speed <- 0;
}

object mercury{
        x <- 300;
        y <- 290;
        speed <- 1;
        distance <- 0.0;
        fixed_distance <- 10.0;
}
```

```
object venus {
	x <- 300;
	y <- 270;
	speed <- 1;
	distance <- 0.0;
	fixed_distance <- 30.0;
}

object earth{
	x <- 300;
	y <- 250;
	speed <- 1;
	fixed_distance <- 50.0;
	distance <- 50.0;
	angle <- 0.0;
}

object mars{
	x <- 300;
	y <- 210;
	speed <- 1;
	distance <- 0.0;
	fixed_distance <- 90.0;
}

func orbit(d, h) {
	d <- distance_to(sun);
	h <- bearing_to(sun) + arccos(speed/2.0/fixed_distance);
}

rule earth {
	orbit(distance, heading);
}

rule mercury {
	orbit(distance, heading);
}

rule venus {
	orbit(distance, heading);
}

rule mars {
```

```
        orbit(distance, heading);
}
```

All testing was created and performed by me, David Suess.

7 Lessons Learned

The main thing I learned from this project is how simple each layer of indirection within the interpreter can be. The overall size of the code was much smaller than I expected and most layers, such as the symbol table, are very simple. The concepts taught in class guide you through every layer in the project. However, the ANTLR syntax was difficult for me to grasp. The ANTLR syntax ought to be studied often and early by anyone who endeavors a project of this type.

7.1 Advice

Adding user defined functions late in the project was difficult. My proposal did not include user defined functions, but after achieving all of the crucial elements of the language, I decided that user defined functions were required in order to allow a programmer is extend my language effectively. I was able to add a rudimentary user defined function capability, but it's typing mechanism is still lacking. Time restraints prevented me from specifying the construct more concretely.

Size of code is surprisingly small. I was surprised to see how small the code needed to be. Professor Edwards had said that if your code seems to grow quickly, then you are doing something wrong. This was very true for me and I am very surprised how small the code for the interpreter is.

ANTLR syntax is tricky. The ANTLR syntax was difficult for me to grasp. I spent many hours examining the class notes and examples of grammar files in order to navigate my way through the lexer, parser and walker. I highly recommend a student dive into their ANTLR files early.

Get a simple version of your interpreter working first before delving into complex details. It was very beneficial for me to get a simple integer assignment statement to work first, before delving into scope, types, functions etc. After that, it becomes clear how to expand the language and you can build up the complexity and test it step by step.

8 Appendix

8.1 Code

8.1.1 sim2d.g

```
class sim2dLexer extends Lexer;
```

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

DIV     options { testLiterals = true; } : '/' ;
STAR    options { testLiterals = true; } : '*' ;
PLUS    options { testLiterals = true; } : '+' ;
MINUS   options { testLiterals = true; } : '-' ;
ASSIGN  options { testLiterals = true; } : "<-" ;
EQ : '=';
NEQ : "!=";
LT : '<';
GT : '>';
LTEQ : "<=";
GTEQ : ">=";
COMMA : ',';
SEMI  : ';' ;
DEREF : '.' ;

PARENS options { testLiterals = true; } : '(' | ')' ;
BRACES options { testLiterals = true; } : '{' | '}' ;


ID options {testLiterals = true;} : ('a'..'z' | 'A'..'Z')
      ('a'..'z' | 'A'..'Z' | '_' | '0'..'9')*
      ;

NUMBER  : (DIGIT)+ ('.' (DIGIT)*)? (('E'|'e') ('+'|'-')? (DIGIT)+)?
      ;

NL      : ('\n' | ('\r' '\n') => '\r' '\n' | '\r')
          { $setType(Token.SKIP); newline(); }
        ;
COMMENT : ( "/*" (
                    options {greedy=false;} :
                    (NL)
                    | ~( '\n' | '\r' )
                )* "*/"
          | "//" (~( '\n' | '\r' ))* (NL)
        )                       { $setType(Token.SKIP); }
        ;

STRING  : '"'!
              (  ~('"' | '\n')
               | ('"'!'"')
              )*
          '"'!
        ;

protected EXP : 'e' ('+' | '-')? ( '0' .. '9')+
      ;
protected DIGIT : ( '0' .. '9' )
      ;

WS : (' ' | '\t' ) //Whitespace
```

```
{ $setType(Token.SKIP); } ;// Action: ignore

class sim2dParser extends Parser;
options {
      k = 2;
   buildAST = true;
}

tokens {
  STATEMENTS;
  FUNCTION;
  ARGS;
  FUNC_DEF_ARGS;
}

file : ( object_stmt | func_def | rule_stmt )+ EOF!
        { #file = #([STATEMENTS,"FILE"],file); }
    ;

any_stmt : object_stmt
         | if_stmt
         | assign_stmt
         | return_stmt
         | function_stmt
              ;

object_stmt : "object"^ ID
              "{"!
              (assign_stmt)*
              "}"!
                        ;

seq_of_stmts : ( any_stmt )*
                    {#seq_of_stmts = #([STATEMENTS,"SEQ_OF_STMTS"],
seq_of_stmts); }
                       ;

rule_stmt : "rule"^ ID
            "{"!
                       ( seq_of_stmts )
            "}"!
                    ;

if_stmt : "if"^ expr
            "{"!
                       ( seq_of_stmts )
            "}"!
                       (options {greedy = true;}:
                           "else"!
                "{"!
                               ( assign_stmt | if_stmt | function_stmt )*
                "}"!
            )?
                    ;

assign_stmt : id "<-"^ expr SEMI!
              ;
```

```
func_def : "func"^ id func_def_args
           "{"!
                       ( seq_of_stmts )
           "}"!
                    ;

return_stmt : "return"^ (expr)? SEMI!
      ;

func_def_args : "("! (farg_list)? ")"!
  { #func_def_args = #([FUNC_DEF_ARGS], func_def_args); }
      ;

function : ID args
           {#function = #([FUNCTION], function); };

function_stmt : function SEMI!;

args : "("! (arg_list)? ")"!
  { #args = #([ARGS], args); }
           ;

farg_list : ID ( COMMA! ID )*
         ;
arg_list : expr ( COMMA! expr )*
         ;

id : ID ( DEREF^ ID )*;

expr : and_expr ( "or"^ and_expr)*;
and_expr : comp_expr ( "and"^ comp_expr)*;
comp_expr : plus_minus_expr ( ( EQ^ | NEQ^ | LT^ | GT^ | LTEQ^ | GTEQ^ )
plus_minus_expr )?;
plus_minus_expr : mult_div_expr (( "+"^ | "-"^) mult_div_expr)* ;
mult_div_expr : deref_expr (("*"^ | "/"^) deref_expr)* ;
deref_expr : atom ( "."^ atom )*;
atom : NUMBER | STRING | "true" | "false" | id | function | "("! expr ")"! ;
;
```

```
{
import java.util.Vector;
}

class sim2dWalker extends TreeParser;

{
    static Sim2dDataType null_data = new Sim2dDataType( "<NULL>" );
}

assign_stmt_in_rule returns [Sim2dDataType r]
```

```
{ Sim2dDataType a,b; r=null_data;
  Sim2dInterpreter interp = Sim2dInterpreter.getInstance(); }
      : #("<-" a=expr b=expr)
                  {
                        r=b;
                        interp.assign(a, b);
                        log.out("walker assign_stmt_in_rule, "
                                    + a.getName() + " = " + b.getName());
                  }
      ;

assign_stmt_in_obj_decl returns [Sim2dDataType r]
{ Sim2dDataType a,b; r=null_data;
  Sim2dInterpreter interp = Sim2dInterpreter.getInstance(); }
      : #("<-" a=expr b=expr)
                  {
                        r=b;
                        interp.put_into_obj(a, b);
                        log.out("walker assign_stmt_in_obj_decl, "
                                    + a.getName() + " = " + b.getName());
                  }
      ;


expr returns [Sim2dDataType r]
{ Sim2dDataType a,b; r=null_data;
  Sim2dDataType[] bs;
  Sim2dInterpreter interp = Sim2dInterpreter.getInstance(); }
   :  #(STATEMENTS (r=expr)*
            )
       | #("-" a=expr b=expr)  { r = a.subtract(b); }
       | #("+" a=expr b=expr)  { r = a.add(b); }
       | #("*" a=expr b=expr)  { r = a.multiply(b); }
       | #("/" a=expr b=expr)  { r = a.lfracts(b); }
       | #(GTEQ a=expr b=expr) { r = a.gteq( b ); }
       | #(LTEQ a=expr b=expr) { r = a.lteq( b ); }
       | #(GT a=expr b=expr)   { r = a.gt( b ); }
       | #(LT a=expr b=expr)   { r = a.lt( b ); }
       | #(EQ a=expr b=expr)   { r = a.eq( b ); }
       | #(NEQ a=expr b=expr)  { r = a.neq( b ); }
       | #("or" a=expr right_or:.)
                  {
                        if ( a instanceof Sim2dBool )
                              r = ( ((Sim2dBool)a).var ? a : expr(#right_or)
);
                        else
                              r = a.or( expr(#right_or) );
                  }
       | #("and" a=expr right_and:.)
                  {
                        if ( a instanceof Sim2dBool )
                              r = ( ((Sim2dBool)a).var ? expr(#right_and) : a
);
                        else
                              r = a.and( expr(#right_and) );
                  }
       | #("not" a=expr)            { r = a.not(); }
```

```
       | #("<-" a=expr b=expr)
                  {
                        r=b;
                        interp.assign(a, b);
                        log.out("walker expr "
                                    + a
                                    + " <- "
                                    + b);
                  }
    | #(DEREF a=expr b=expr
           {
                        Sim2dObject o =
(Sim2dObject)interp.getSymbol(a.getName());
                        r = o.getField(b.getName());
                  }
       )
    | #(fu:FUNCTION a=expr bs=arguments
            {
                  Sim2dDataType d = interp.getSymbol(a.getName() );
                  if (!(d instanceof Sim2dFunction)) {
                        throw new Sim2dException("Undefined function " +
a.getName());
               }
                  Sim2dFunction f = (Sim2dFunction)d;
               log.out("got function call " + a.getName());
                  interp.enterScope();
                  r = f.execute(interp.sym, bs);
                  interp.leaveScope();
            }
            )
    | #("if" a=expr ifbody:. (elsebody:.)?)
        {
           if ( !( a instanceof Sim2dBool ) )
               return a.error( "if: expression should be bool" );
           if ( ((Sim2dBool)a).var )
               r = expr( #ifbody );
           else if ( null != elsebody )
               r = expr( #elsebody );
        }

       | "true"  { r = new Sim2dBool(true);  }
       | "false" { r = new Sim2dBool(false); }

    | id:ID
                  {
                        r = interp.getSymbol(id.getText() );
                  }
    | num:NUMBER
                  {
                        r = interp.getNumber( num.getText() );
                  }
      ;

arguments returns [ Sim2dDataType[] rargs ]
{
    Sim2dInterpreter interp = Sim2dInterpreter.getInstance();
```

```
    Sim2dDataType a;
    rargs = null;
    Vector v;
}
        : #(ARGS          { v = new Vector(); }
                ( a=expr      { v.add( a ); }
                )*
           )                  { rargs = interp.convertExprList( v ); }
}
        ;

file returns [ Sim2dDataType r ]
{ Sim2dDataType a; r=null_data; }
      :  #(STATEMENTS (a=object_or_rule_stmt)*
                  {
                     log.out("walker: FILE ");
                  }
             )
   ;

object_or_rule_stmt returns [ Sim2dDataType r ]
{ Sim2dDataType a; r=null_data;
  String[] fargs;
  Sim2dInterpreter interp = Sim2dInterpreter.getInstance();}
   :  #("object" oname:ID
                  {
                     log.out("object " + oname.getText());
                        Sim2dObject newobj = new
Sim2dObject(oname.getText());
                        interp.put(newobj);
                        interp.setObjectDefinitionMode(true);
                  }
                  (a=assign_stmt_in_obj_decl)*
                  {
                     interp.objectPostProcessing();// add fields that all
objects have
                        interp.setObjectDefinitionMode(false);
                        log.out("object def complete");
                  }
             )
funcbody:.)
    | #("func" fname:ID fargs=tlist funcbody:.)
                {
                      log.out("func " + fname.getText());
                        interp.put(new Sim2dUserFunction(interp.walker,
fname.getText(),
                                                fargs, #funcbody ));

                  }
    | #("rule" rname:ID rulebody:.)
                {
                      log.out("rule " + rname.getText());
                        interp.put(new Sim2dRule(rname.getText(), #rulebody
));

                  }
```

```
      ;

tlist returns [ String[] fargs ]
{
    Vector v;
    fargs = null;
}
        : #(FUNC_DEF_ARGS { v = new Vector(); }
              (s:ID       { v.add( s.getText() ); }
              )*
           )              { fargs = new String[v.size()];
                                    v.toArray(fargs); }
        ;
```

8.1.2 Sim2dBool.java

```
import java.io.PrintWriter;

class Sim2dBool extends Sim2dDataType {
    boolean var;

    Sim2dBool( boolean var ) {
        this.var = var;
    }

    public Sim2dBool( String name, boolean x ) {
              super(name);
        var = x;
    }


    public String typename() {
        return "bool";
    }

    public Sim2dDataType copy() {
        return new Sim2dBool( var );
    }

    public Sim2dDataType assign( Sim2dDataType rvalue ) {
             log.out("bool assign");
             if (rvalue instanceof Sim2dBool)
                   var = ((Sim2dBool)rvalue).var;
             else
                   throw new Sim2dException("Incompatible types in"
                               + " assignment, " + typename() + " <- "
                               + rvalue.typename());
             return this;
       }


    public void print( PrintWriter w ) {
        if ( name != null )
            w.print( name + " = " );
        w.println( var ? "true" : "false" );
```

```
    }


    public Sim2dDataType and( Sim2dDataType b ) {
        if ( b instanceof Sim2dBool )
            return new Sim2dBool( var && ((Sim2dBool)b).var );
        return error( b, "and" );
    }

    public Sim2dDataType or( Sim2dDataType b ) {
        if ( b instanceof Sim2dBool )
            return new Sim2dBool( var || ((Sim2dBool)b).var );
        return error( b, "or" );
    }

    public Sim2dDataType not() {
        return new Sim2dBool( !var );
    }

    public Sim2dDataType eq( Sim2dDataType b ) {
        // not exclusive or
        if ( b instanceof Sim2dBool )
            return new Sim2dBool( ( var && ((Sim2dBool)b).var )
                                || ( !var && !((Sim2dBool)b).var ) );
        return error( b, "==" );
    }

    public Sim2dDataType neq( Sim2dDataType b ) {
        // exclusive or
        if ( b instanceof Sim2dBool )
            return new Sim2dBool( ( var && !((Sim2dBool)b).var )
                                || ( !var && ((Sim2dBool)b).var ) );
        return error( b, "!=" );
    }
       public String toString() {
              return typename() + " " + name + "=" + var;
       }
}
```

8.1.3 Sim2dDataType.java

```
public class Sim2dDataType
{
      String name;

      public Sim2dDataType() {
            name = null;
      }

   public Sim2dDataType copy() {
      return new Sim2dDataType( name );
   }


      public Sim2dDataType( String name ) {
            setName(name);
```

```
  }

  public void setName( String name ) {
        this.name = name;
  }

  public String getName( ) {
        return name;
  }

  public Sim2dDataType multiply(Sim2dDataType b) {
  return error( b, "+" );
  }

  public Sim2dDataType error( String msg ) {
        //throw new Sim2dException( "illegal operation: " + msg
        System.out.println( "illegal operation: " + msg
                    + "( <" + typename() + "> "
                    + ( name != null ? name : "<?>" )
                    + " )" );
        return null;
  }

  public String typename() {
        return "unknown";
  }

  public Sim2dDataType error( Sim2dDataType b, String msg ) {
        if ( null == b )
              return error( msg );
        //throw new Sim2dException(
        System.out.println( "illegal operation: " + msg
                    + "( <" + typename() + "> "
                    + ( name != null ? name : "<?>" )
                    + " and "
                    + "<" + typename() + "> "
                    + ( name != null ? name : "<?>" )
                    + " )" );
        return null;
  }

public Sim2dDataType assign( Sim2dInt rvalue ) {
    return error( rvalue, "<-" );
}
public Sim2dDataType assign( Sim2dDataType rvalue ) {
    return error( rvalue, "<-" );
}

public Sim2dDataType transpose() {
    return error( "\'" );
}

public Sim2dDataType uminus() {
    return error( "-" );
}

public Sim2dDataType plus( Sim2dDataType b ) {
```

```
    return error( b, "+" );
}

public Sim2dDataType add( Sim2dDataType b ) {
    return error( b, "+=" );
}

public Sim2dDataType minus( Sim2dDataType b ) {
    return error( b, "-" );
}

public Sim2dDataType subtract( Sim2dDataType b ) {
    return error( b, "-=" );
}

public Sim2dDataType times( Sim2dDataType b ) {
    return error( b, "*" );
}

public Sim2dDataType mul( Sim2dDataType b ) {
    return error( b, "*=" );
}

public Sim2dDataType lfracts( Sim2dDataType b ) {
    return error( b, "/" );
}

public Sim2dDataType rfracts( Sim2dDataType b ) {
    return error( b, "/\'" );
}

public Sim2dDataType ldiv( Sim2dDataType b ) {
    return error( b, "/=" );
}

public Sim2dDataType rdiv( Sim2dDataType b ) {
    return error( b, "/\'=" );
}

public Sim2dDataType modulus( Sim2dDataType b ) {
    return error( b, "%" );
}

public Sim2dDataType rem( Sim2dDataType b ) {
    return error( b, "%=" );
}

public Sim2dDataType gt( Sim2dDataType b ) {
    return error( b, ">" );
}

public Sim2dDataType gteq( Sim2dDataType b ) {
    return error( b, ">=" );
}

public Sim2dDataType lt( Sim2dDataType b ) {
    return error( b, "<" );
```

```
    }

    public Sim2dDataType lteq( Sim2dDataType b ) {
        return error( b, "<=" );
    }

    public Sim2dDataType eq( Sim2dDataType b ) {
        return error( b, "==" );
    }

    public Sim2dDataType neq( Sim2dDataType b ) {
        return error( b, "!=" );
    }

    public Sim2dDataType and( Sim2dDataType b ) {
        return error( b, "and" );
    }

    public Sim2dDataType or( Sim2dDataType b ) {
        return error( b, "or" );
    }

    public Sim2dDataType not() {
        return error( "not" );
    }

       public String toString() {
              return typename() + " " + name;
       }

}
```

8.1.4 Sim2dException.java

```
class Sim2dException extends RuntimeException {
    Sim2dException( String msg ) {
        System.err.println( "Error: "+msg );
    }
}
```

8.1.5 Sim2dFloat.java

```
import java.io.PrintWriter;

class Sim2dFloat extends Sim2dDataType {
    float var;

    public Sim2dFloat( float x ) {
        var = x;
    }

    public Sim2dDataType assign( Sim2dDataType rvalue ) {
             if (rvalue instanceof Sim2dFloat)
                   var = ((Sim2dFloat)rvalue).var;
             else
```

```
                throw new Sim2dException("Incompatible types in"
                            + " assignment, " + typename() + " <- "
                            + rvalue.typename());
            return this;
    }

public Sim2dDataType assign( Sim2dFloat rvalue ) {
            var = rvalue.var;
            return this;
    }

public Sim2dFloat( String name, float x ) {
             super(name);
    var = x;
}

public String typename() {
    return "float";
}

public Sim2dDataType copy() {
    return new Sim2dFloat( name, var );
}

public static float floatValue( Sim2dDataType b ) {
    if ( b instanceof Sim2dFloat )
        return ((Sim2dFloat)b).var;
    b.error( "cast to float" );
    return 0;
}

public void print( PrintWriter w ) {
    if ( name != null )
        w.print( name + " = " );
    w.println( Float.toString( var ) );
}

public Sim2dDataType uminus() {
    return new Sim2dFloat( -var );
}

public Sim2dDataType plus( Sim2dDataType b ) {
    return new Sim2dFloat( var + floatValue(b) );
}

public Sim2dDataType add( Sim2dDataType b ) {
    var += floatValue( b );
    return this;
}

public Sim2dDataType minus( Sim2dDataType b ) {
      return new Sim2dFloat( var - floatValue(b) );
}

public Sim2dDataType subtract( Sim2dDataType b ) {
    var -= floatValue( b );
    return this;
```

```
}

public Sim2dDataType times( Sim2dDataType b ) {
    return new Sim2dFloat( var * floatValue(b) );
}

public Sim2dDataType multiply( Sim2dDataType b ) {
    return new Sim2dFloat( var * floatValue(b) );
}

public Sim2dDataType lfracts( Sim2dDataType b ) {
    return new Sim2dFloat( var / floatValue(b) );
}

public Sim2dDataType rfracts( Sim2dDataType b ) {
    return lfracts( b );
}

public Sim2dDataType ldiv( Sim2dDataType b ) {
    var /= floatValue(b);
    return this;
}

public Sim2dDataType rdiv( Sim2dDataType b ) {
    return ldiv( b );
}

public Sim2dDataType modulus( Sim2dDataType b ) {
    return new Sim2dFloat( var % floatValue(b) );
}


public Sim2dDataType rem( Sim2dDataType b ) {
    var %= floatValue( b );
    return this;
}

public Sim2dDataType gt( Sim2dDataType b ) {
    if ( b instanceof Sim2dFloat )
        return new Sim2dBool( var > floatValue(b) );
    if ( b instanceof Sim2dInt )
        return new Sim2dBool( var > (float)(Sim2dInt.intValue(b)) );
    return b.lt( this );
}

public Sim2dDataType gteq( Sim2dDataType b ) {
    if ( b instanceof Sim2dFloat )
        return new Sim2dBool( var >= floatValue(b) );
    if ( b instanceof Sim2dInt )
        return new Sim2dBool( var >= (float)(Sim2dInt.intValue(b)) );
    return b.lteq( this );
}

public Sim2dDataType lt( Sim2dDataType b ) {
    if ( b instanceof Sim2dFloat )
        return new Sim2dBool( var < floatValue(b) );
    if ( b instanceof Sim2dInt )
```

```
            return new Sim2dBool( var < (float)(Sim2dInt.intValue(b)) );
        return b.gt( this );
    }

    public Sim2dDataType lteq( Sim2dDataType b ) {
        if ( b instanceof Sim2dFloat )
            return new Sim2dBool( var <= floatValue(b) );
        if ( b instanceof Sim2dInt )
            return new Sim2dBool( var <= (float)(Sim2dInt.intValue(b)) );
        return b.gteq( this );
    }

    public Sim2dDataType eq( Sim2dDataType b ) {
        if ( b instanceof Sim2dFloat )
            return new Sim2dBool( var == floatValue(b) );
        return b.eq( this );
    }

    public Sim2dDataType neq( Sim2dDataType b ) {
        if ( b instanceof Sim2dFloat )
            return new Sim2dBool( var != floatValue(b) );
        return b.neq( this );
    }

       public String toString() {
              return typename() + " " + name + "=" + var;
       }
}
```

8.1.6 Sim2dFunction.java

```
import java.util.LinkedHashMap;
import java.io.PrintWriter;
import antlr.collections.AST;

class Sim2dFunction extends Sim2dDataType {

    public Sim2dFunction( String inname ) {
              super(inname);
    }

    public String typename() {
        return "function";
    }

       public Sim2dDataType execute(Sim2dSymbolTable symt, Sim2dDataType[]
args)
     throws antlr.RecognitionException    {
              throw new Sim2dException("Function " + name + " body
undefined");
       }

       public String toString() {
             String str = typename() + " " + name;
             return str;
       }
```

```
}
```

8.1.7 Sim2dInt.java

```
import java.io.PrintWriter;

class Sim2dInt extends Sim2dDataType {
    int var;

    public Sim2dInt( int x ) {
        var = x;
    }

    public Sim2dDataType assign( Sim2dDataType rvalue ) {
             if (rvalue instanceof Sim2dInt)
                   var = ((Sim2dInt)rvalue).var;
             else
                   throw new Sim2dException("Incompatible types in"
                               + " assignment, " + typename() + " <- "
                               + rvalue.typename());
             return this;
       }

    public Sim2dDataType assign( Sim2dInt rvalue ) {
             var = rvalue.var;
             return this;
       }

    public Sim2dInt( String name, int x ) {
              super(name);
        var = x;
    }

    public String typename() {
        return "int";
    }

    public Sim2dDataType copy() {
        return new Sim2dInt( name, var );
    }

    public static int intValue( Sim2dDataType b ) {
        if ( b instanceof Sim2dInt )
            return ((Sim2dInt)b).var;
        b.error( "cast to int" );
        return 0;
    }

    public void print( PrintWriter w ) {
        if ( name != null )
            w.print( name + " = " );
        w.println( Integer.toString( var ) );
    }

    public Sim2dDataType uminus() {
        return new Sim2dInt( -var );
```

```
}

public Sim2dDataType plus( Sim2dDataType b ) {
    return new Sim2dInt( var + intValue(b) );
}

public Sim2dDataType add( Sim2dDataType b ) {
    var += intValue( b );
    return this;
}

public Sim2dDataType minus( Sim2dDataType b ) {
    return new Sim2dInt( var - intValue(b) );
}

public Sim2dDataType subtract( Sim2dDataType b ) {
    var -= intValue( b );
    return this;
}

public Sim2dDataType times( Sim2dDataType b ) {
    return new Sim2dInt( var * intValue(b) );
}

public Sim2dDataType multiply( Sim2dDataType b ) {
    return new Sim2dInt( var * intValue(b) );
}

public Sim2dDataType lfracts( Sim2dDataType b ) {
    return new Sim2dInt( var / intValue(b) );
}

public Sim2dDataType rfracts( Sim2dDataType b ) {
    return lfracts( b );
}

public Sim2dDataType ldiv( Sim2dDataType b ) {
    var /= intValue(b);
    return this;
}

public Sim2dDataType rdiv( Sim2dDataType b ) {
    return ldiv( b );
}

public Sim2dDataType modulus( Sim2dDataType b ) {
    return new Sim2dInt( var % intValue(b) );
}


public Sim2dDataType rem( Sim2dDataType b ) {
    var %= intValue( b );
    return this;
}

public Sim2dDataType gt( Sim2dDataType b ) {
    if ( b instanceof Sim2dInt )
```

```
            return new Sim2dBool( var > intValue(b) );
        return b.lt( this );
    }

    public Sim2dDataType gteq( Sim2dDataType b ) {
        if ( b instanceof Sim2dInt )
            return new Sim2dBool( var >= intValue(b) );
        return b.lteq( this );
    }

    public Sim2dDataType lt( Sim2dDataType b ) {
        if ( b instanceof Sim2dInt )
            return new Sim2dBool( var < intValue(b) );
        return b.gt( this );
    }

    public Sim2dDataType lteq( Sim2dDataType b ) {
        if ( b instanceof Sim2dInt )
            return new Sim2dBool( var <= intValue(b) );
        return b.gteq( this );
    }

    public Sim2dDataType eq( Sim2dDataType b ) {
        if ( b instanceof Sim2dInt )
            return new Sim2dBool( var == intValue(b) );
        return b.eq( this );
    }

    public Sim2dDataType neq( Sim2dDataType b ) {
        if ( b instanceof Sim2dInt )
            return new Sim2dBool( var != intValue(b) );
        return b.neq( this );
    }

       public String toString() {
              return typename() + " " + name + "=" + var;
       }
}
```

8.1.8 Sim2dInterpreter.java

```
import java.util.*;
import java.io.*;
import antlr.CommonAST;
import antlr.collections.AST;
import antlr.RecognitionException;
import antlr.TokenStreamException;
import antlr.TokenStreamIOException;
import antlr.debug.misc.ASTFrame;
class Sim2dInterpreter {
      sim2dWalker walker;
   private static Sim2dInterpreter ref;
      private boolean object_definition_mode;
      public Sim2dSymbolTable sym;
   private Sim2dInterpreter() {
      sym = new Sim2dSymbolTable( null );
```

```
            object_definition_mode = false;
       put(new Sim2ddistance_to("distance_to"));
       put(new Sim2dbearing_to("bearing_to"));
       put(new Sim2darccos("arccos"));
    }
    public static Sim2dInterpreter getInstance()
    {
      if (ref == null)
          ref = new Sim2dInterpreter();
      return ref;
    }
    public void setWalker(sim2dWalker inwalker) {
       walker = inwalker;
    }
    public boolean put(Sim2dDataType data) {
            return sym.put(data);
       }
    public boolean put_into_obj(Sim2dDataType lvalue, Sim2dDataType rvalue) {
            if (lvalue.name == null) {
                  throw new Sim2dException("Invalid lvalue");
            }
            Sim2dDataType symbol = sym.getLast();
       log.out("put_into_obj:  " + symbol.name);
            if (!(symbol instanceof Sim2dObject)) {
                  throw new Sim2dException(" trying to put data into object,
but last symbol in symbol table is not a Sim2dObject");
            }
            Sim2dDataType newsymbol = rvalue.copy();
            newsymbol.setName(lvalue.getName());
            Sim2dObject objsymbol = (Sim2dObject)symbol;
            objsymbol.put(newsymbol);
            return true;
       }

    public boolean assign(Sim2dDataType lvalue, Sim2dDataType rvalue) {
            if (lvalue.name == null) {
                  throw new Sim2dException("Invalid lvalue");
            }
            lvalue.assign(rvalue);

            return true;
       }

    public Sim2dDataType[] convertExprList( Vector v ) {
        Sim2dDataType[] x = new Sim2dDataType[v.size()];
        for ( int i=0; i<x.length; i++ )
            x[i] = (Sim2dDataType) v.elementAt( i );
        return x;
    }


       public void objectPostProcessing () {
            Sim2dDataType symbol = sym.getLast();
       log.out("objectPostProcessing:  " + symbol.name);
            if (!(symbol instanceof Sim2dObject)) {
                  throw new Sim2dException(" trying to put data into object,
but last symbol in symbol table is not a Sim2dObject");
```

```
            }
            Sim2dObject objsymbol = (Sim2dObject)symbol;
            objsymbol.putInternalFields();
      }
      public Sim2dDataType getSymbol(String name) {
            Sim2dDataType symbol = sym.get(name);
            if (getObjectDefinitionMode()) {
                  if (symbol == null) {
                        return new Sim2dDataType(name);
                  } else {
                        throw new Sim2dException("Multiple symbols named " +
name);
                  }
            } else {
                  if (symbol == null) {
                        throw new Sim2dException("Unknown symbol " + name);
                  }

            }
            return symbol;
      }
      public boolean getObjectDefinitionMode() {
            return object_definition_mode;
      }
      public void setObjectDefinitionMode(boolean mode) {
            object_definition_mode = mode;
      }
      public Sim2dDataType getNumber(String a) {
            try {
               return new Sim2dInt(Integer.parseInt(a));
            } catch (NumberFormatException e) {
            }
            try {
               return new Sim2dFloat(Float.parseFloat(a));
            } catch (NumberFormatException e) {
                  throw new Sim2dException("invalid number");
            }
      }
      public void enterScope() {
            Sim2dSymbolTable newsym;
            newsym = new Sim2dSymbolTable( sym );
            sym = newsym;
      }
      public void leaveScope() {
            sym = sym.parent;
            // The symbol table for the scope we are leaving
            // will be destroyed by the garbage collector
      }
      public boolean runRule(Sim2dObject obj)
     throws antlr.RecognitionException    {
            boolean rule_found = false;
            for (Sim2dDataType symbol : sym) {
                  if (symbol.getName().equals(obj.getName())
                              && symbol instanceof Sim2dRule ) {
                        Sim2dRule rule = (Sim2dRule)symbol;
                        // create a new symobl table for the scope of the
rule
```

```
                        // and put the objects symbols into it
                        enterScope();
                        for (Sim2dDataType field : obj.fields.values()) {
                              sym.put(field);
                        }
                        walker.expr(rule.getBody());
                        //walker.rbody(rule.getBody());
                        // do we need to copy the symbols in current scope to
                        // the parent scope?, could be bad, need references
                        leaveScope();
                        rule_found = true;
                  }
            }
            return rule_found;

      }

}
```

8.1.9 Sim2dObject.java

```
import java.util.LinkedHashMap;
import java.io.PrintWriter;

class Sim2dObject extends Sim2dDataType {
       public LinkedHashMap<String, Sim2dDataType> fields;

    public Sim2dObject( String name ) {
              super(name);
        fields = new LinkedHashMap<String, Sim2dDataType>();
    }

    public String typename() {
        return "object";
    }


    public void print( PrintWriter w ) {
        if ( name != null )
            w.print( name + " = " );
    }

       public void put(Sim2dDataType data) {
             log.out("Adding field to object " + name + " field is " +
data.name);
             if (fields.containsKey(data.name)) {
                   System.out.println("Error:  object " + name + " has
multiple declarations of the field " + data.name);
                   System.exit(1);
             }
             fields.put(data.name, data);
       }

       public String toString() {
```

```
            String str = typename() + " " + name + "(" + fields.size() + ")
{";
            boolean first = true;
            for (String f : fields.keySet()) {
                  if (!first) {
                     str += ",";
                  }
                  str += " " + getField(f).toString();
                  first = false;
            }
            str += "}";
            return str;
      }

      public Sim2dDataType getField(String name) {
            return fields.get(name);
      }

      public void putIfAbsent(Sim2dDataType newfield) {
            if (!fields.containsKey(newfield.name)) {
                  put(newfield);
            }
      }

      public void putIfAbsent(Sim2dFloat newfield) {
            if (fields.containsKey(newfield.name)) {
                  Sim2dDataType f = fields.get(newfield.name);
                  if (f instanceof Sim2dInt) {
                        fields.remove(f.name);
                        newfield.var = ((Sim2dInt)f).var;
                  }
            }
            put(newfield);
      }

      public void putInternalFields() {
            putIfAbsent(new Sim2dFloat("x", 0));
            putIfAbsent(new Sim2dFloat("y", 0));
            putIfAbsent(new Sim2dFloat("heading", 0));
            putIfAbsent(new Sim2dFloat("speed", 0));
            putIfAbsent(new Sim2dBool("visible", true));
      }
}
```

8.1.10 Sim2dRule.java

```
import java.util.LinkedHashMap;
import java.io.PrintWriter;
import antlr.collections.AST;

class Sim2dRule extends Sim2dDataType {
    AST body;

    public Sim2dRule( String inname , AST inbody) {
              super(inname);
              body = inbody;
```

```
    }

    public String typename() {
        return "rule";
    }

    public AST getBody() {
        return body;
    }

       public String toString() {
             String str = typename() + " " + name;
             return str;
       }

}
```

8.1.11 Sim2dSymbolTable.java

```
import java.util.LinkedList;
import java.io.PrintWriter;

class Sim2dSymbolTable extends LinkedList <Sim2dDataType> {
    Sim2dSymbolTable parent;
    boolean read_only;
    int indent = 0;

    public Sim2dSymbolTable( Sim2dSymbolTable nparent ) {
       parent = nparent;
       read_only = false;
    }

       public Sim2dDataType get(String name) {
             for (Sim2dDataType symbol : this) {
                   if (symbol.getName().equals(name)) {
                         return symbol;
                   }
             }
             if (parent != null) {
                   return parent.get(name);
             }
             return null;
       }

       public boolean put(Sim2dDataType data) {
             return add(data);
       }

       public boolean enterScope() {
             return true;
       }

       public boolean leaveScope() {
             return true;
       }
```

```
        public void Dump() {
           log.out("Symbol Table Dump:");
              Dump(this);
        }
        public void Dump(Sim2dSymbolTable scope) {
              if (scope == null) {
                    return;
              }
              indent--;
              Dump(scope.parent);
        indent++;
              for ( Sim2dDataType symbol : scope) {
                    for (int i=0; i<indent; i++) {
                       System.out.print("   ");
                    }

                    if (!(symbol instanceof Sim2dFunction)) {
                       System.out.println(symbol);
                    }
              }
        }
}
```

8.1.12 Sim2dUserFunction.java

```
import java.util.LinkedHashMap;
import java.io.PrintWriter;
import antlr.collections.AST;

class Sim2dUserFunction extends Sim2dFunction {
    AST body;
       sim2dWalker walker;
       String [] argaliases;

    public Sim2dUserFunction( sim2dWalker inwalker, String inname,
                                        String[] inargs, AST inbody) {
              super(inname);
              body = inbody;
              walker = inwalker;
              if (inargs == null)
                    log.out("inargs null");

              argaliases = inargs;
    }

    public String typename() {
        return "function";
    }

    public AST getBody() {
        return body;
    }

       public Sim2dDataType execute(Sim2dSymbolTable symt, Sim2dDataType[]
args)
     throws antlr.RecognitionException    {
```

```
                log.out("in " + this + " calling walker");
                if (walker == null)
                      log.out("its null");
                String[] oldnames = new String[args.length];
                if (oldnames == null)
                      log.out("oldnames null");
                if (argaliases == null)
                      log.out("argaliases null");
         log.out("oldnames.length = " + oldnames.length +
                              "argaliases.length = " + argaliases.length);
                int i = 0;
                for (Sim2dDataType arg : args) {
                      oldnames[i] = arg.getName();
                      log.out("setname " + arg.getName() + " to " +
argaliases[i]);
                      arg.setName(argaliases[i]);
                      i++;
                      symt.put(arg);
                }
                //symt.Dump();
                Sim2dDataType ret = walker.expr(getBody());
                i = 0;
                for (Sim2dDataType arg : args) {
                      log.out("setname " + arg.getName() + " back to " +
oldnames[i]);
                      arg.setName(oldnames[i++]);
                }
                return ret;
        }

        public String toString() {
              String str = typename() + " " + name;
              return str;
        }
}
```

8.1.13 Sim2darccos.java

```
class Sim2darccos extends Sim2dFunction {

    public Sim2darccos( String inname ) {
              super(inname);
    }

    public String typename() {
        return "arccos";
    }

       public Sim2dDataType execute(Sim2dSymbolTable symt, Sim2dDataType[]
args) {
                log.out("Executing function " + typename());
                if (args.length != 1) {
                throw new Sim2dException("function " + typename() + " was
passed "
                                    + args.length + " arguments, but only accepts
1");
```

```
                }
                float input;
                if (args[0] instanceof Sim2dInt) {
                        input = (float)((Sim2dInt)args[0]).var;
        } else if (args[0] instanceof Sim2dFloat) {
                        input = ((Sim2dFloat)args[0]).var;
           } else {
                    throw new Sim2dException("Invalid type '" +
args[0].typename()
                                    + "' passed to function " + typename());
                }

                return new Sim2dFloat((float)Math.toDegrees(Math.acos(input)));
       }

}
```

8.1.14 Sim2dbearing_to.java

```
import java.util.LinkedHashMap;
import java.io.PrintWriter;
import antlr.collections.AST;

class Sim2dbearing_to extends Sim2dFunction {

    public Sim2dbearing_to( String inname ) {
                super(inname);
    }

    public String typename() {
        return "bearing_to";
    }

       public Sim2dDataType execute(Sim2dSymbolTable symt, Sim2dDataType[]
args) {
                log.out("Executing function " + typename());
                if (args.length != 1) {
                throw new Sim2dException("function bearing_to was passed "
                                + args.length + " arguments, but only accepts
1");
                }
                if (!(args[0] instanceof Sim2dObject)) {
                throw new Sim2dException("first argument to distance_to is not
an object");
                }
                // The local object x and y symbols should be on top of symbol
                // table stack
                Sim2dFloat x1 = (Sim2dFloat)symt.get("x");
                Sim2dFloat y1 = (Sim2dFloat)symt.get("y");

                // The other object is passed as argument.  Get it's x and y
                Sim2dObject to = (Sim2dObject)args[0];
                Sim2dFloat x2 = (Sim2dFloat)to.getField("x");
                Sim2dFloat y2 = (Sim2dFloat)to.getField("y");
                float ang = (float)Math.toDegrees(Math.atan2( x2.var - x1.var,
```

```
y1.var - y2.var));
              if (ang > 360.0) {
                    ang -= 360.0;
              } else if (ang < 0.0) {
                    ang += 360.0;
              }

              return new Sim2dFloat(ang);
       }

}
```

8.1.15 Sim2ddistance_to.java

```
import java.util.LinkedHashMap;
import java.io.PrintWriter;
import antlr.collections.AST;

class Sim2ddistance_to extends Sim2dFunction {

    public Sim2ddistance_to( String inname ) {
              super(inname);
    }

    public String typename() {
        return "distance_to";
    }

       public Sim2dDataType execute(Sim2dSymbolTable symt, Sim2dDataType[]
args) {
              log.out("Executing function " + typename());
              if (args.length != 1) {
              throw new Sim2dException("function distance_to was passed "
                                + args.length + " arguments, but only accepts
1");
              }
              if (!(args[0] instanceof Sim2dObject)) {
              throw new Sim2dException("first argument to distance_to is not
an object");
              }
              // The local object x and y symbols should be on top of symbol
              // table stack
              Sim2dFloat x1 = (Sim2dFloat)symt.get("x");
              Sim2dFloat y1 = (Sim2dFloat)symt.get("y");

              // The other object is passed as argument.  Get it's x and y
              Sim2dObject to = (Sim2dObject)args[0];
              Sim2dFloat x2 = (Sim2dFloat)to.getField("x");
              Sim2dFloat y2 = (Sim2dFloat)to.getField("y");
              return new Sim2dFloat(
                           (float)Math.pow( Math.pow(y2.var - y1.var, 2.0) +
                                            Math.pow(x2.var - x1.var, 2.0),
0.5));
       }
```

```
}
```

8.1.16 WindowExiter.java

```
import java.awt.event.*;
public class WindowExiter extends WindowAdapter {
      public void windowClosing(WindowEvent event) {
            System.exit(0);
      }
}
```

8.1.17 log.java

```
public class log {
      static boolean enabled = false;
      public static void on() {
            enabled = true;
      }
      public static void off() {
            enabled = false;
      }
      public static void out(String text) {
            if (enabled) {
               System.out.println(text);
            }
      }
}
```

8.1.18 myJPanel.java

```
import java.awt.*;
import javax.swing.*;

class myJPanel extends JPanel
{
  Sim2dSymbolTable sym;

  public myJPanel(Sim2dSymbolTable insym) {
        sym = insym;
  }

  public void paintComponent(Graphics g) {
     super.paintComponent(g);

        for (Sim2dDataType symbol : sym) {
              if (symbol instanceof Sim2dObject) {
                    Sim2dObject objsymbol = (Sim2dObject)symbol;
                    Sim2dBool visible =
                       (Sim2dBool)objsymbol.getField("visible");
                    if (visible.var) {
                       Sim2dFloat x = (Sim2dFloat)objsymbol.getField("x");
                       Sim2dFloat y = (Sim2dFloat)objsymbol.getField("y");
                       g.fillOval((int)(x.var), (int)(y.var), 3, 3);
                    }
              }
```

```
        }
    }
}
```

8.1.19 sim2d.java

```
import java.io.*;
import antlr.CommonAST;
import antlr.collections.AST;
import antlr.debug.misc.ASTFrame;
import antlr.DumpASTVisitor;
import antlr.RecognitionException;
import antlr.TokenStreamException;

class sim2d {
      Display display = null;
      boolean graphical_ast = false;
      boolean text_ast = false;
      boolean skip_simulation = false;
      boolean dump_symbol_table = false;
      boolean suppress_display = false;
      int iteration_delay = 250;
      int quit_count = -1;
      String src_file = "";

      private static void dumpAST(AST node, int level) {
            if (node != null) {
                  for (int i = 0; i < level; i++)
                        System.out.print("   ");
                  System.out.println(//"Type = " + node.getType() +
                              //+ " Text = " +
                              node.getText());
                  dumpAST(node.getFirstChild(), level+1);
                  dumpAST(node.getNextSibling(), level);
            }
      }
      public static void main(String[] args) {
            sim2d newsim = new sim2d(args);
      }
      public sim2d(String args[]) {
            for (int i = 0; i < args.length; i++) {
                  if (args[i].equals("-g")) {
                        graphical_ast = true;
                  } else if (args[i].equals("-t")) {
                        text_ast = true;
                  } else if (args[i].equals("-s")) {
                        suppress_display = true;
                  } else if (args[i].equals("-v")) {
                        log.on();
                  } else if (args[i].equals("-r")) {
                        iteration_delay = new Integer(args[++i]);
                  } else if (args[i].equals("-i")) {
                        quit_count = new Integer(args[++i]);
                  } else if (args[i].equals("-d")) {
                        dump_symbol_table = true;
                  } else if ((args[i].equals("-h"))
```

```
                        || (args[i].equals("--help"))
                        || (args[i].equals("-help"))) {
                    System.out.println("Sim2d options:\n"
                       + "   -h            This message\n"
                       + "   -s            Suppress map display\n"
                       + "   -i [number]   Quit after [number]
iterations\n"
                       + "   -r [delay]    Time delay between iterations
in msecs\n"
                       + "   -g            Show graphical AST\n"
                       + "   -t            Show textual AST\n"
                       + "   -d            Dump symbol table after each
iteration\n"
                       + "   -v            Output verbose debugging
text\n");
                    System.exit(0);
                } else {
                    src_file = args[i];
                }
            }

            try {
                FileInputStream filename = new FileInputStream(src_file);
                DataInputStream input = new DataInputStream(filename);
                sim2dLexer lexer = new sim2dLexer(
                    new DataInputStream(input));
                sim2dParser parser = new sim2dParser(lexer);
                parser.setFilename(src_file);
                // Parse the input expression
                parser.file();
                CommonAST t = (CommonAST)parser.getAST();
                // Print the resulting tree out in LISP notation
                if (text_ast) {
                   System.out.println(t.toStringList());
                }
                if (graphical_ast) {
                   // Open a window for AST displayed graphically
                   ASTFrame frame = new ASTFrame(
                             "AST from the Sim2d parser", t);
                   frame.setVisible(true);
                }

                sim2dWalker walker = new sim2dWalker();
                // Traverse the tree created by the parser
                Sim2dInterpreter interp = Sim2dInterpreter.getInstance();
                interp.setWalker(walker);
                walker.file(t);
                if (dump_symbol_table) {
                    interp.sym.Dump();
                }
                int iter = 0;
                if (!suppress_display) {
                display = new Display(interp.sym);
                }
                while (iter != quit_count) {
                   for (Sim2dDataType symbol : interp.sym) {
                      if (symbol instanceof Sim2dObject) {
```

```
                        try {
                           Thread.currentThread().sleep(iteration_delay);
                        } catch (InterruptedException e) {
                     }
                     log.out("Running rule " + symbol.getName());
                     interp.runRule((Sim2dObject)symbol);
                     Sim2dObject objsymbol = (Sim2dObject)symbol;
                     Sim2dFloat x = (Sim2dFloat)objsymbol.getField("x");
                     Sim2dFloat y = (Sim2dFloat)objsymbol.getField("y");
                     Sim2dFloat heading =
                         (Sim2dFloat)objsymbol.getField("heading");
                     Sim2dFloat speed =
                         (Sim2dFloat)objsymbol.getField("speed");
                     x.var += speed.var *
                         Math.sin(Math.toRadians(heading.var));
                     y.var -= speed.var *
                         Math.cos(Math.toRadians(heading.var));


                  }
               }
               if (dump_symbol_table) {
                  interp.sym.Dump();
               }
               if (!suppress_display) {
                  display.refresh();
               }
               if (quit_count > 0) {
                  iter++;
               }
            }
            System.exit(0);
         } catch (FileNotFoundException e) {
            System.err.println("Error (file): "+e.getMessage());
            System.exit(-1);
         }
         catch(TokenStreamException e) {
            System.err.println("Error (token): "+e.getMessage());
         }
         catch(RecognitionException e) {
            System.err.println("Error (recog): "+e.getMessage());
         }
         catch(Exception e) {
            System.err.println("Error (excep): " +e);
            e.printStackTrace();
         }
      }
}
```

8.2 Makefile

```
# comment
JFLAGS =
#JFLAGS = -g
JC = javac
#JC = jikes
```

```
.SUFFIXES:  .java .class

.java.class:
	$(JC) $(JFLAGS) $*.java

CLASSES = \
		Display.class \
		WindowExiter.class \
		myJPanel.class \
		sim2d.class \
		Sim2dInterpreter.class \
		Sim2dSymbolTable.class \
		Sim2dDataType.class \
		Sim2dException.class \
		Sim2dInt.class \
		Sim2dFloat.class \
		Sim2dBool.class \
		Sim2dObject.class \
		Sim2dFunction.class \
		Sim2dUserFunction.class \
		Sim2ddistance_to.class \
		Sim2dbearing_to.class \
		Sim2darccos.class \
		Sim2dRule.class \
		log.class

CLASSES_FROM_GENERATED_CODE = \
		sim2dLexer.class \
		sim2dLexerTokenTypes.class \
		sim2dParser.class \
		sim2dWalker.class \
		sim2dLexerTokenTypes.class

#default: sim2d.jar

sim2d.jar: ranantlr $(CLASSES_FROM_GENERATED_CODE) $(CLASSES)
	jar cvf $@ $(CLASSES) $(CLASSES_FROM_GENERATED_CODE)

sim2dLexer.class: sim2dLexer.java
sim2dParser.class: sim2dParser.java
sim2dWalker.class: sim2dWalker.java

ranantlr: sim2d.g
	antlr sim2d.g
	touch ranantlr

# -ea turns on assertion
run: sim2d.jar
	java -ea Display

classes: $(CLASSES:.java=.class)

clean:
	rm -f *.class sim2d.jar
	rm -f *.smap sim2dLexer.java sim2dWalker.java sim2dParser.java
	rm -f sim2dLexerTokenTypes.java sim2dTokenTypes.java
	rm -f sim2dLexerTokenTypes.txt
```

```
rm -f ranantlr
```

8.3 Test Scripts

8.3.1 runalltests.csh

```
#!/bin/csh
date
runtest.csh $* -s -i 10 -r 0 -d -t test_1_obj_1_field.s2d
runtest.csh $* -s -i 10 -r 0 -d -t test_1_obj_2_fields.s2d
runtest.csh $* -s -i 10 -r 0 -d -t test_1_obj_3_fields.s2d
runtest.csh $* -s -i 10 -r 0 -d -t test_1_obj_4_fields.s2d
runtest.csh $* -s -i 10 -r 0 -d -t test_1_obj_5_fields.s2d
runtest.csh $* -s -i 10 -r 0 -d -t test_hdg_0_speed_2.s2d
runtest.csh $* -s -i 10 -r 0 -d -t test_hdg_45_speed_2.s2d
runtest.csh $* -s -i 10 -r 0 -d -t test_hdg_90_speed_2.s2d
runtest.csh $* -s -i 10 -r 0 -d -t test_hdg_135_speed_2.s2d
runtest.csh $* -s -i 10 -r 0 -d -t test_hdg_180_speed_2.s2d
runtest.csh $* -s -i 10 -r 0 -d -t test_hdg_225_speed_2.s2d
runtest.csh $* -s -i 10 -r 0 -d -t test_hdg_270_speed_2.s2d
runtest.csh $* -s -i 10 -r 0 -d -t test_hdg_315_speed_2.s2d
runtest.csh $* -s -i 10 -r 0 -d -t test_hdg_360_speed_2.s2d
runtest.csh $* -s -i 3 -r 0 -d -t test_rule_w_assign.s2d
runtest.csh $* -s -i 3 -r 0 -d -t test_rule_w_if.s2d
runtest.csh $* -s -i 3 -r 0 -d -t test_deref.s2d
```

8.3.2 runtest.csh

```
#!/bin/csh
set TEST_DIR="test"

set GREEN="\033[32m"
set RED="\033[31m"
set FAIL_OPTS=""
set OPTIONS=""
while ( ${#argv} > 0 && ! $?ARGSDONE )
      switch ( "$1" )
      # use -nc option to turn of colorization
            case "-nc"
                  set GREEN=""
                  set RED=""
                  shift
                  breaksw
            default
               set ext=`echo $1 | cut -d. -f2`
               if ( "$ext" == "s2d" ) then
                     set TEST=`echo $1 | cut -d. -f1`
         else
                     set OPTIONS="$OPTIONS $1"
                  endif
                  shift
                  breaksw
      endsw
end
```

```
set ENDCOLOR="\033[00m"

java sim2d ${OPTIONS} $TEST_DIR/${TEST}.s2d > $TEST_DIR/${TEST}.out
diff $TEST_DIR/${TEST}.gold $TEST_DIR/${TEST}.out
if ( "$status" != "0" ) then
   echo ${TEST} " \t${RED}FAILED${ENDCOLOR}"
   if ( "$FAIL_OPTS" != "" ) then
      java sim2d $FAIL_OPTS < $TEST_DIR/${TEST}.s2d
      endif
else
   echo ${TEST} " \t${GREEN}PASSED${ENDCOLOR}"
endif
```

9 References

DEC C Language Reference Manual

http://www.tru64unix.compaq.com/docs/base_doc/DOCUMENTATION/V40F_HTML/AQTLTBTE/TITLE.HTM

Mx Language

http://www1.cs.columbia.edu/~sedwards/classes/2003/w4115/mx030521.zip

Antlr 2.7.7 Documentation and Examples

http://www.antlr2.org/download.html