COMS W4115
Programming Languages & Translators

NewLogo Final Report

Shulei Zhao (sz2352)	**Project Manager**
Pengfei Weng (pw2336)	**Language and Tools Guru**
Yuandi Jin (yj2246)	**System Architect**
Adarsh Banad (apb2138)	**System Integrator**
Yang Pan (yp2301)	**Tester and Validator**

05/06/2012

Contents

1. Introduction (By Whole Team) 3

1.1 General Introduction 3

1.2 Features 3

2. Language Tutorial (By Whole Team) 5

2.1 Your First Program 5

2.2 Variable Declaration 6

2.3 Operators 6

2.4 Loop 7

2.5 Condition 7

2.6 Drawing 7

2.7 Functions 8

2.8 Comment 9

2.9 Compile and Run NewLogo programs 9

2.10 More Complex Demos 9

3. Language Reference Manual (By Pengfei Weng) 14

3.1 Lexical Conventions 14

3.2 Data Types and Operators 15

3.3 Expressions 17

3.4 Statements 20

3.5 Function Declaration 24

3.6 Scopes 24

4. Project plan (by Shulei Zhao) 26

4.1 Process of Development 26

4.2 Roles and Responsibilities 27

4.3 Implementation Style 28

4.4 Time Line 29

4.5 Project Log 29

5. Developing Tools (By Pengfei Weng) 31

5.1 Tools Used to Create Compiler Components 31

5.2 Unusual Libraries Used in the Compiler 31

6. Translator Architecture (By Yuandi Jin) 32

6.1 Architecture Overview 32

6.2 Modules 32

6.3 Files and Work Distribution 33

7. Development and Run-time Environment (By Adarsh Banad) 34

7.1 Software Development Environment 34

7.2 Makefile 34

7.3 Run-time Environment 36

8. Test Plan (By Yang Pan) 37

8.1 Simple Function Tests 37

8.2 Complex Test Case 38

8.3 Testing Method and Testing Automation 40

9. Conclusions (By Whole Team) 41

9.1 Lessons Learned as a Team 41

9.2 Lessons Learned by Each Team Member 41

9.3 Advice for Future Teams 43

9.4 Suggestions for Instructor 43

1. Introduction

1.1 General Introduction

NewLogo is a language designed to allow users drawing 2D graphics more freedom and control the drawing process more accurately. Users will find it is much easier to learn how to use NewLogo to draw complex or irregular shapes. NewLogo will make users feel like they are drawing with a pen on a piece of paper. Users can also control the moving direction and distances of the "pen" in NewLogo to create various 2D geometries.

NewLogo provides solutions for programming beginners when they deal with some drawing work but do not know much about any existing languages because of its intuitiveness. NewLogo is also a very convenient teaching tool for teachers to help children learn about plane geometry and plane coordinate system. After children get enough experiences of programming with NewLogo, they can design their own shapes or even draw pictures with combinations of different shapes.

1.2 Features of NewLogo

1.2.1 Easy to Learn

Some existing languages like C++ or Java also get many powerful drawing functions, but users have to import the corresponding API or libraries into their programs and they have to take time to study what functions they can use and how to use them. This will make programming beginners and children feel very frustrated since they do not have enough experience in programming. The grammar of NewLogo will make the learning process much easier and the sentences are very straightforward to users. The sentences written with NewLogo are very concise and intuitive, so understanding programs in NewLogo will be much simpler.

1.2.2 More Freedom

Compared with the drawing functions in other programming languages like C++ or Java, NewLogo does not totally depend on the coordinate system to finish the drawing work. Instead, users can draw with the "pen" in NewLogo by setting values to its moving direction and distance. Also because NewLogo is not like C++ or Java which depend on the pre-defined and regular geometries so much, it can provide functions for users to operate on their self-defined shapes in an easier way. Users can enjoy the hand-painting experiences with more freedom when programming with NewLogo.

1.2.3 More Accuracy

As a programming language, NewLogo also enables users to control their drawing processes with parameters. For example, NewLogo is much more helpful when users find it difficult to set different values for the diameters of a group of circles and it can draw a regular hexagon with a simple "for" loop without repeating the same drawing method 6 times. This will help users manage the positions or distances more accurately, decrease the difficulty of creating new geometries, and make the drawing experiences more interesting.

2. Language Tutorial

NewLogo is a language using simple codes to draw complex and colorful graphs with user's intention.

The basic idea of NewLogo is using a pen to draw pictures like we did in the kindergarten. Imagine you have a pen located at the center of the canvas at the beginning. Then the purpose of your program is to move the pen so that lines can be drawn. Meanwhile, you can change pen's color to draw a colorful picture and fill a certain area with the color you want.

Our language has most features a modern programming language should have, including variable declarations, functions, value calculation and recursion.

2.1 Your First Program

Here is a simple program in NewLogo:

```
void main () {
   forward 50;
   right 90;
   forward 50;
}
```

This will draw a line of length 50, and then turn right 90 degrees, and then draw another 50-length line.

In NewLogo language, each program needs a main function to be run. Main function can call any other functions that have been defined already.

All the iteration statements, selection statements, and condition statements are in JAVA style. Besides, all the drawing statements are in script style (i.e. "forward 50;" or "right 67.5;"). For the meaning of all the drawing functions, please refer to the drawing function description.

2.2 Variable Declaration

Variable declaration contains two parts, type name and variable name. A type could be an integer, boolean, String or double. Notice every declaration needs an initial value. Here are some examples of variable declaration:

```
int a = 1;
double c = 2.3;
boolean flag = true;
String s = "hello world!";
```

2.3 Operators

Operators (From High to Low Priority)
()
++ --
-
* /
+ -
< <= > >=
== !=
!
&&
||
=

In NewLogo, we support most numerical and logical calculations. Here are some examples:

```
a = a + 1;
b = 2 + 3 * 4;
flag = (a < 1) && (b != 2);
```

2.4 Loop

We use for and while as the keywords for loops in NewLogo, which is same as Java. Here are the examples:

```
for (int i = 0; i < 3, i++){
   forward 50 * i;
}

int a = 0;
while (a != 5){
   a++;
}
```

2.5 Condition

Our condition statement uses keywords *if* and *else*. The syntax is *if (condition) {statement-list}*, or *If (condition) {statement-list} else {statement-list}*,. This is much like Java language.

Here are some examples of how to use Condition statements:

```
# if condition
if (a < b){
   a = b;
}

# if-else condition
if (a < b){
   a = b;
}
else{
   a = b -1;
}
```

2.6 Drawing

Drawing is the main mission for our NewLogo language, which is supported by the following drawing statements:

`forward int`	move forward int steps
`back int`	move back int steps
`right int/double`	turn right int degrees
`left int/double`	turn left int degrees
`pup`	raise pen up, moving without drawing
`pdown`	put pen down, ready for drawing
`pcolor int`	change the color of the pen, we provide 9 different colors
`pwidth int`	change the width of the lines drawn by the pen
`fillstart/fillend`	marks the starting/ending point to fill some geometry
`print String`	print a string to the screen in the current pen position

2.7 Functions

The syntax of our function definition is very like Java:

```
return_type function_name (parameter list)
{
        function body
}
```

All the functions must be implemented when the user define them. In NewLogo, we let all the functions as global functions, which can be called by other functions.

Here is an example of our function with recursion:

```
int fibonacci(int n)
{
   if (n <= 2)
     {return 1;}
   else
     {return fibonacci(n-1) + fibonacci(n-2);}
}
```

2.8 Comment

We use “#” to illustrate the beginning of inline comments. For example:

```
forward 5;# this is a command for going forward 5 steps
```

2.9 Compile and Run NewLogo Programs

NewLogo programs could be run under the Linux system easily.

First, open the terminal and go to the directory where the NewLogo source codes and compiler components codes locate.

Second, just input “make all” and it will generate our compiler.

Lastly, input “make test” and the compiler will compile the program in the file “test.nlg”

After that, users can see the pictures “test.nlg”.

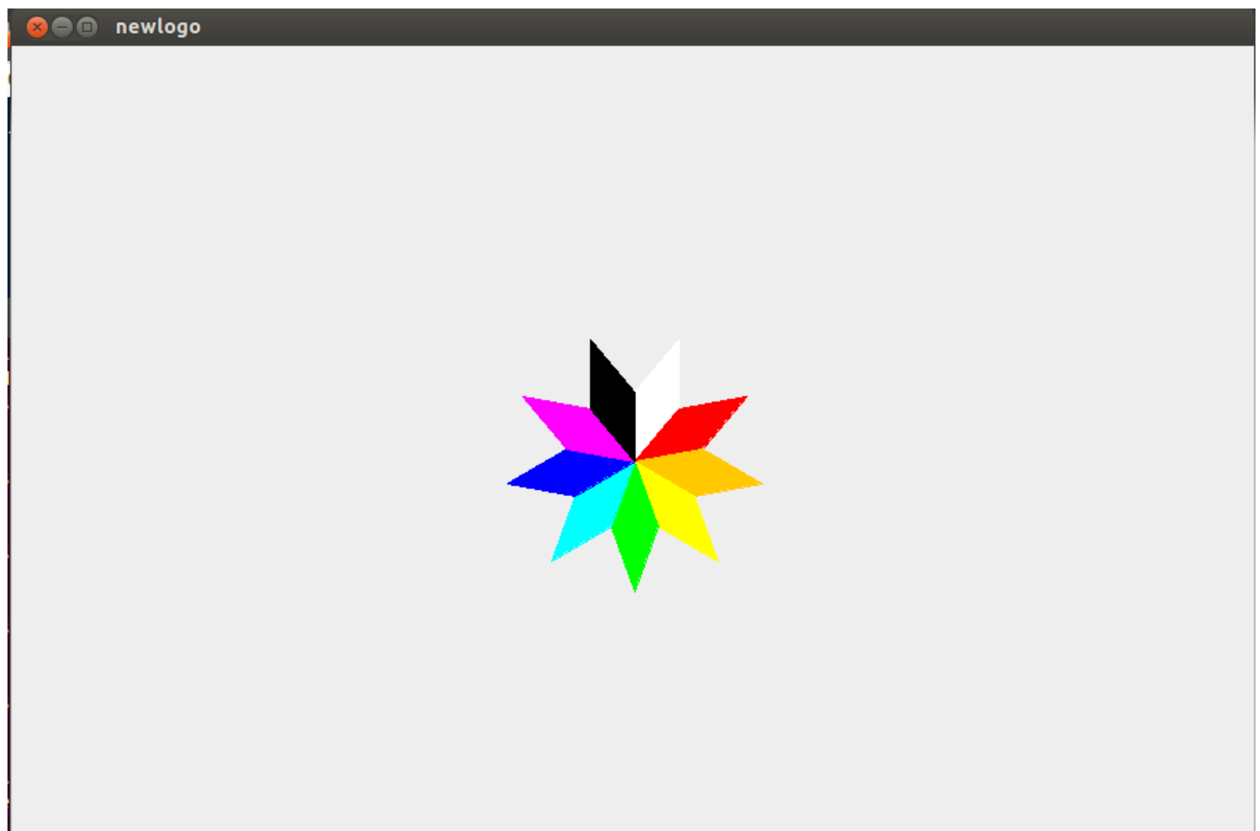
2.10 More Complex Demos

2.2.10.1 Nine-color flower

```
#this program will draw a flower with nine petals each of which
#has a unique color
void drawFlower(int n){
   for(int i = 0; i < n; i++){
      pcolor i;# draw different petals with different colors
      fillstart; # prepare to fill the petal
      forward 50;
      right 40;
      forward 50;
```

```
        right 140;
        forward 50;
        right 40;
        forward 50;
        right 140;  # draw a diamond to represent the petal
        fillend; # fill the petal
        right 40;
    }
}
void main ( ) {
    drawFlower(9); #call the function to draw the flower
}
```

The picture generated by this program is shown as below.



2.2.10.2 Chinese National Flag

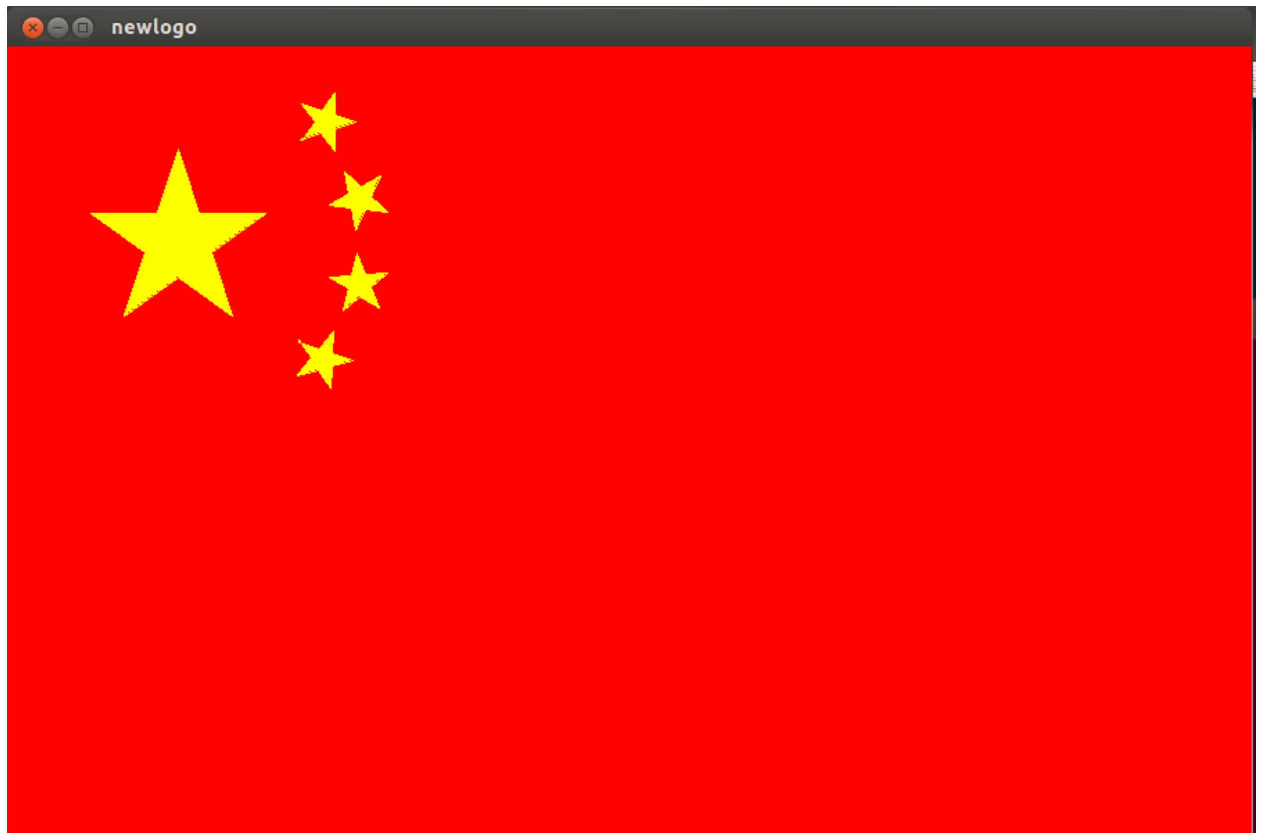
```
#this program will draw a Chinese national flag
```

```
void drawStar(int m){
   fillstart;
   left 18;
   pcolor 3;
   for(int i = 0; i < 5; i++){
      forward m;
      left 72;
      forward m;
      right 144;
   }
   fillend;
   right 18;
   return;
}
void drawStars(){
   right 108;
   drawStar(48);
   pup;
   forward 60;
   left 55;

   for(int i = 0; i < 4; i++){
      forward 120;
      pdown;
      drawStar(16);
      pup;
      back 120;
      right 25;
   }
}
void drawFrame()
{
   pup;
   forward 300;
   left 90;
```

```
   pdown;
   pcolor 1;
   fillstart;
   boolean flag = true;
   for (int i=0; i<4; i++)
   {
      if (flag)
      {
         forward 450;
         left 90;
         forward 300;
         flag = false;
      }
      else
      {
         forward 300;
         left 90;
         forward 450;
         flag = true;
      }
   }
   fillend;
   right 90;
   pup;
   back 300;
   pdown;
}
void main ( ) {
   drawFrame();
   pup;
   forward 180;
   left 90;
   forward 390;
   right 90;
   pdown;
```

```
    drawStars();
    return;
}
```

The generated picture is shown as below:



3. Language Reference Manual

3.1 Lexical Conventions

There are different classes of tokens that are supported in our NewLogo. Token types are identifier, integer, double, string, keywords and operators. As in C++ language, whitespace characters are ignored except insofar as they serve to delineate other tokens in the input stream. If the input stream has been parsed into tokens up to a given character, the next token is taken to include the longest string of characters, which could possibly constitute a token.

3.1.1 Character Set

NewLogo takes the standard ASCII character set for source codes.

3.1.2 Comments

NewLogo only supports inline comments. Comments are prefaced by the # symbol and all contents in the current line after it are regarded as comments.

```
#this is a comment.
forward 500; # this is also a comment
```

3.1.3 Identifiers

An identifier is a sequence of letters and digits. There are several rules for our identifier. For instance, the first character of the identifier must be a letter. The underscore "_" is not viewed as a letter. The upper and lower case letters are different in the identifier.

Identifiers may have different length, and at least the first 31 characters are significant for the internal identifiers while for some implementations more characters are significant.

Internal identifiers include preprocessor names and all other names without external linkage. Identifiers with external linkage are more restricted.

3.1.4 Keywords

The following table shows all the key words in our NewLogo language.

int	double	boolean	String	void	
true	false	main	return		

if	else	for	while		
forward	back	right	left	pdown	pup
pcolor	pwidth	fillstart	fillend	print	

3.1.5 Operators

NewLogo has 7 categories of operators. They are unary, additive, multiplicative, relational and logical, respectively.

Unary	- !
Multiplicative	* /
Relational	< > <= >=
Equality	== !=
Logical	&& || !
Assignment	=

3.1.6 Separators

NewLogo recognizes three types of separators of tokens. They are space, tab, and new line. The compile considers no difference among them.

3.1.7 Syntax Group

{ }: Braces are delimiter of compound statement, used in the cases of statements block and constant array initialization.

(): Parentheses are for expression grouping and argument expression list.

3.1.8 Punctuators

, : used to separate arguments for function calls.
; : used to separate statements.

3.2 Data Types and Operators

3.2.1 Data Types

There are four data types in NewLogo, including int, double, boolean and String.

`int:` The int data type stores a 32-bit signed integer.

`double:` The double data type stores a 64-bit signed floating point number.

`boolean`: Boolean type is a primitive type in NewLogo. There are only two values, true and false, in the Boolean type, which is used to determinate logic conditions. There are no conversions between integers and these two values.

`String`: The string data type is the only character data type, and stores a sequence of characters. A string is permutation of at least one character. The size of a string equals the number of characters in it.

After some variables are declared without initial values, the default initial values will be assigned based on the data types of the variables. The default initial values of the four data types are listed as below.

Data Type	Default Initial Value
`int`	0
`double`	0.0
`boolean`	false
`String`	“”

3.2.2 Precedence and Associativity Rules of Operators

The table below lists the precedence and associativity of all operators.

Tokens (From High to Low Priority)	Operators
identifiers, constants, parenthesized expression	Primary expression
()	Function calls, subscripting
++ --	Increment, decrement (postfix)
-	Unary minus
* /	Multiplicative
+ -	Additive
< <= > >=	Relational
== !=	Equality
!	Logical Not
&&	Logical And
||	Logical Or
=	Assignment

- Arithmetic Operators

In NewLogo, arithmetic operators are +, -, * and /. + means addition, - means substruction, * means multiplication and / means division. All of them are binary and

left associative. It requires that their operands must be of the same primitive types, and the result will be of the same type.

- Comparative Operators

In NewLogo, comparative operators are > (greater than), < (less than), >= (greater than or equal to), <= (less than or equal to), ! = (not equal) and == (equal). All of them are binary operators and left associative. It requires that their operands must be of the same primitive types. The return value is a boolean value indicates the predication.

- Logical Operators

Logical operators in NewLogo include && (logical and), || (logical or) and ! (logical not). && and || are binary operators and left-associative. They take two operands of type boolean, and return a boolean value. ! is unary and appears on the left side of the operand. They type of the operand must be of type boolean and the return type is also a boolean value.

- Assignment Operators

NewLogo's assignment operator is =. It's binary operator and right associative. The left operand must be a legal left value, and the right operand must be an expression. In NewLogo, a legal left value can be a variable. When an assignment is taken, the value of the expression on the right is assigned to the left value, and the new value of the left value is returned. However, we don't chained assignments.

- Parentheses

In NewLogo, parentheses in expressions are used to overwrite the precedence rules. Expression enclosed by the parentheses is calculated before applying its adjacent operators.

3.3 Expressions

Most statements in NewLogo contain expressions and most work of a program is achieved by evaluating expressions. Expressions can be broken down into operators and operands.

Operators are functionality that do something and can be represented by symbols or by special keywords. Operators require some data to operate on and such data are called operands.

To fully consider the precedence of expressions, we build a 7-level expression

structure.

3.3.1 Expression Top Level

```
expression →   variable = expression
               | expression && logical_expression
               | expression | | logical_expression
               | logical_expression
```

Assignment operator is =. It associates from right to left side. Assignment operators require a modifiable lvalue as their left operand.

3.3.2 Logical Expression

Logical operators in NewLogo include ! (logical not), && (logical and) and || (logical or). The logical not operator ! has higher priority than the other two and it operates on a single boolean operand associates from right to left. The logical and && and logical or || operate on two boolean operands and associate from left to right. The results of these logical operators are also boolean values.

```
logical_expression →   ! relational_expression
                       | relational_expression
```

3.3.3 Relational Expression

Relational operators include <, <=, >, >=, == and !=. Expressions with relational operators associate from left to right. The syntax for relational operators is as follows:

```
relational_expression   → additive_expression != additive_expression
                          | additive_expression == additive_expression
                          | additive_expression <= additive_expression
                          | additive_expression >= additive_expression
                          | additive_expression > additive_expression
                          | additive_expression < additive_expression
```

The operator < means less than, > means greater than, <= means less than or equal to, >= means greater than or equal to. The result is of type int. The value of the result is 1 if the specified relation is true, otherwise it is 0.
The operator == means equal to, != means not equal to. The operands must have the same type.

3.3.4 Additive Expression

Additive operators include + and -. Expressions with additive operators associate from

left to right. The syntax for additive operators is as follows:

```
additive_expression→ additive_expression + multiplicative_expression
                    | additive_expression - multiplicative_expression
                    | -multiplicative_expression
                    | multiplicative_expression
```

The result of the + operator is the sum of the operands. The result of the - operator is the difference of the operands.

If either of the operands is of type double, the result of the operator is of type double. Otherwise the result is of type int.

The unary operator in NewLogo is unary minus. Expressions with unary minus associate from right to left.

The result of the unary - operator is the negative of its operand which can be int or double. The result will have the same type as the operand.

3.3.5 Multiplicative Expression

Multiplicative operators include * and /. Expressions with multiplicative operators associate from left to right. The syntax for multiplicative operators is as follows:

```
multiplicative_expression→multiplicative_expression*primary_expression
                         |multiplicative_expression/primary_expression
                         |primary_expression
```

The binary * operator indicates multiplication. The binary / operator indicate division of the first operator (dividend) by the second (divisor).

3.3.6 Primary Expression

Primary expressions include identifiers, constants or parenthesized expressions. The syntax for primary expressions is as follows:

```
primary_expression → constant
                   | identifier
                   | (expression)
                   | postfix_expression
```

A constant's type is determined by its form and value.

A parenthesized expression has the same type and value as those of the expression

without the parentheses. Parentheses are used to overwrite precedence rules.

3.3.7 Postfix Expression

Postfix operators include subscripting, function calls, postfix increment and postfix decrement. The syntax for postfix operators is as follows:

postfix-expression → primary_expression
| function **(**argument-expr-list**)**
| postfix_expression **++**
| postfix_expression **--**

argument-expr-list → argument-expression
| argument-expr-list, argument-expression
| []

A function call is a postfix expression followed by parentheses containing a (possibly empty) comma-separated list of expressions that are the arguments to the function.

When postfix ++/-- is applied to a modifiable lvalue (left hand side value, a value that has an address) of type int, the result is the value of the object referred to by the lvalue. And after the result is noted, the object is incremented or decremented by 1.

3.4 Statements

Statements are executed in sequence. Statements are fall into several groups:

statement → expression-statement
| declaration-statement
| compound-statement
| selection-statement
| iteration-statement
| jump-statement
| drawing-statement

3.4.1 Expression Statements

The most common statements are expression statements. It has same format as expression and a semicolon at the end, as lists below:

expression-statement → expression;

3.4.2 Compound Statement

The compound statement is provided so that several statements can be used where one is expected. The syntax is list below:

compound-statement → {statement-list}
| {}

statement-list → statement statement-list | statement

3.4.3 Selection Statement

Selection statements choose one of several flows of control:

selection-statement → **if (**expression**)** compound-statement
| **if (**expression**)** compound-statement
else compound-statement

In the first form of if statement, the first sub-statement is executed only when the expressions is evaluated to be true. In the second form, the first sub-statement is executed when the expression is evaluated to be true and the second sub-statement is executed if the expression is false.

3.4.4 Iteration Statements

There are two kinds of iterative statements: while statement and for statement.

iteration-statement → **while (**expression**)** compound-statement
| **for (**for_init_statement expression**;** expression**)**
compound-statement

for_init_statement → expression-statement
| declaration-statement

The sub-statements in the while statement are executed repeatedly as long as the value of the expression remains unequal to 0.

In for statement, the first expression is used to declare new variables for this loop and it is evaluated only once. The second expression is evaluated before each loop, and if it becomes false, for statement is terminated. The third expression makes modifications to the variable used in the second expression and it is evaluated after each loop.

3.4.5 Jump Statements

We may use some other statements to jump to some other positions of the program in special conditions.

jump-statement → **break;**
| **continue;**
| **return;**
| **return (**expression**);**

The break and continue statements may appear in iteration-statement. It can immediately terminate the iteration, and keep executing the codes following the iteration. The continue statement terminates the current iteration and enters next iteration.

The return statement is used to exit a function and return a value from the function. The expression in the return statement is optional.

3.4.6 Drawing Statements

The drawing statements in NewLogo are responsible for all the functions to draw geometries with a "pen" on a "canvas". There are 12 drawing statements defined in NewLogo as listed below.

pdown;
Put the pen down on the canvas and prepare to draw on the canvas.

pup;
Lift the pen off the canvas and the pen cannot draw on the canvas until it is put down again.

forward *expression*;
Move the pen forward for the distance specified by *expression* along the current direction.

back *expression*;
Move the pen backward for the distance specified by *expression* along the current direction.

left *expression*;
Turn the current direction of the pen to the left by the radian specified by *expression*.

right *expression*;
Turn the current direction of the pen to the right by the radian specified by *expression*.

pcolor *expression*;
Set the pen color to the value specified by expression.

pwidth *expression*;
Set the width of lines drawn with the pen to the value specified by expression.

fillstart;
Fill the most recently finished geometry with the current color of the pen. fillstart marks the starting point.

fillend;
fillend marks the ending point to finish the filling.

print *expression*;
Print a string to the canvas in the current position with the current color of the pen.

In the statements above, NewLogo has different requirements for the types and ranges of expressions. If the expressions are not constant, NewLogo will only check the types of them. The details are listed in the table below.

Number of Statements	Type Requirements	Range Requirements
`pdown`	-	-
`pup`	-	-
`forward`	int	-
`back`	int	-
`right`	int, double	-
`left`	int, double	-
`pcolor`	int	[0,8]
`pwidth`	int, double	[0, +∞]
`fillstart`	-	-
`fillend`	-	-
`print`	String	

The +∞ in the table above only means users can specify a constant expression with a value as large as they want. But if the value is too large, there will be an error.

3.4.7 Declaration Statements

NewLogo only allows the declarations of variables. Declarations specify the interpretation given to each identifier. The type-specifiers are used to specify the data types.

dclaration-statement → declaration
| declaration Declaration-statement

declaration → datatype identifier = expression;

In NewLogo, users can only declare one variable with one type. The initial value of the variable must be given in NewLogo.

3.5 Function Declaration

In each NewLogo program, there is exactly one main function which should be declared as below.

```
void main () {
    statement-list
}
```

```
statement-list → statement
               | statement statement-list
```

Users can declare zero, one or more new functions in NewLogo programs. The user-declared functions are all global functions. No functions can be declared in other functions and one function cannot be called before it is declared and implemented.

```
type-specifier identifier (parameter-list) {
    statement-list
}
```

```
parameter-list → parameter-declaration
               | parameter-list, parameter-declaration
```

parameter-declaration → type-specifier identifier

3.6 Scopes

3.6.1 Lexical Scope

The lexical scope of an identifier defined outside of a function extends from its definition to the end of the file. The lexical scope of an identifier within a function extends from its definition to the end of the function.

3.6.2 Function Scope

Functions have global scope. A function defined at any point in the file can be used at

any point in the file.

4. Project Plan

4.1 Process of Development

4.1.1 Team Discussion

The team usually holds a stand-up meeting after the lecture on Wednesday which lasts for about 5-10 minutes. During this meeting, each team member first talks about his problems for his current work. For each question, all the team members make suggestions for the problems and make decision together. After that, team members set the next goal of the project development and its time line.

If there is the team is reaching a milestone of the project (e.g. the white paper, the LRM, etc.), team members will choose a time and location for a longer team meeting or team programming.

Longer team meetings are usually set on Fridays and Saturdays which start from 2PM and last for 1-2 hours. During these longer meetings, the team talks about details of the project development or specific problems. For example, the project manager will review on the previous milestone and distribute the each team member’s tasks for the next one, or the language guru will explain how to solve the problems about the grammar and semantics.

4.1.2 Team Programming

After the team starts to write the codes for the compiler, team programming meetings are arranged during the end of week. In most cases, they will last for more than 4 hours. All team members will sit together and code on different parts. Since each team member has his own part of the compiler to finish, if something ambiguous or even problematic happens in some part, there is always someone who can explain to the whole team and modify the code if it is necessary.

4.1.3 Remote Cooperation

Other than the team meetings, sometimes team members also work separately and communicate with each other meanwhile. The project manager is responsible for keep the whole team “together”. After each member’s responsibility is set during the meetings, the project manager will distribute and collect the work or documents of other team members.

The documents and codes are transferred with emails and shared in the Dropbox. The log and version control is also managed with the help with them. After each milestone, the project manager will send an email to the whole group for a review (e.g. what problems have been solved, or what problems are still there) and introduce next one.

4.2 Roles and Responsibilities

Name	Role	Project Responsibilities	Coding Responsibilities
Shulei Zhao	Project Manager	Set the project schedule, hold weekly meetings with the entire team, and maintain the project log	Backend Java code, and test cases
Pengfei Weng	Language and Tools Guru	Defines the baseline process to track language changes, and teach the team how to use various tools and OCaml to build the compiler.	Scanner, parser and AST
Yuandi Jin	System Architect	build the compiler architecture, modules, and interfaces	Generator
Adarsh Banad	System Integrator	define the system integration environment and keep	Compiler, generator, and test cases

		the compiler components work together	
Yang Pan	Tester and Validator	Design test cases and executes them to test the compiler	Semantic

4.3 Implementation Style

4.3.1 Readability

Before each team member writes the code for his own parts, we make some conventions to name the files, functions and parameters. All the names are simple and straightforward.

Team members also use comments to explain the codes to make it much easier to read the code for each other and for users of the compiler.

4.3.2 Modularization

Since we built the basic frame of the whole complier before we start to code, we divide the whole compiler into several functional parts to allow team members to work on separately. But each team member still follows the same convention to define functions and variables in his own part.

In the process of debugging for the compiler, it is very helpful to use modularization method to test our code one part after one and then make modifications to fix bugs.

4.3.3 Clarity

Team members follow the coding conventions of the OCaml language by indenting the code when necessary to standardize the code. Whitespace and empty lines are also used to clearly separate different segments of the code to guarantee the clarity.

4.4 Time Line

02/15/2012	Language functionality and characteristics discussed
02/22/2012	Main language functions defined and whitepaper finished
03/21/2012	Grammar completed and language reference manual finished
03/23/2012	Development tools and environment setup
04/24/2012	Scanner, parser, AST and backend Java code completed
05/03/2012	Compiler and generator completed
05/05/2012	Semantics error checking completed
05/06/2012	Report finished and project finalized

4.5 Project Log

02/10/2012	First team meeting: design and select language, assign roles to each member
02/12/2012	Second team meeting: finalize language concepts and start to write white paper
02/22/2012	First milestone: finish white paper
03/09/2012	Third team meeting: design the basic grammar
03/16/2012	Fourth team meeting: discuss details about grammar and start to write LRM and user tutorial
03/22/2012	Second milestone: finish the LRM and user tutorial
03/23/2012	Fourth team meeting: choose Java to be the backend language, Linux to be platform and OCaml to build compiler, and start to learn OCaml
04/20/2012	Fifth team meeting: build the basic frame of our compiler
04/24/2012	Third milestone: finish the scanner, parser, AST and backend Java code
05/03/2012	Fourth milestone: finish the compiler and generator and test the whole compiler with programs with no errors
05/04/2012	Fifth milestone: finish semantic error checking
05/05/2012	Sixth milestone: test the whole compiler with programs with or without errors, and start the report

05/06/2012	Seventh milestone: finish the report and finalize the code

5. Developing Tools

5.1 Tools Used to Create Compiler Components

We use the Eclipse Java IDE to develop and test our backed Java code. We use OCaml to build the components of the compiler.

5.2 Unusual Libraries Used in the Compiler

We do not use any unusual libraries when we design our compiler with OCaml. For the backend Java code, we use a popular Java GUI widget toolkit called Swing to draw the pictures.

6. Translator Architecture

6.1 Architecture Overview

In NewLogo, we have essentially 5 modules: scanner, parser/ast, semantic, , generator and Java code frame. Our compiler takes as input a source program in NewLogo language, and outputs a Java program. The block diagram is shown as follows:

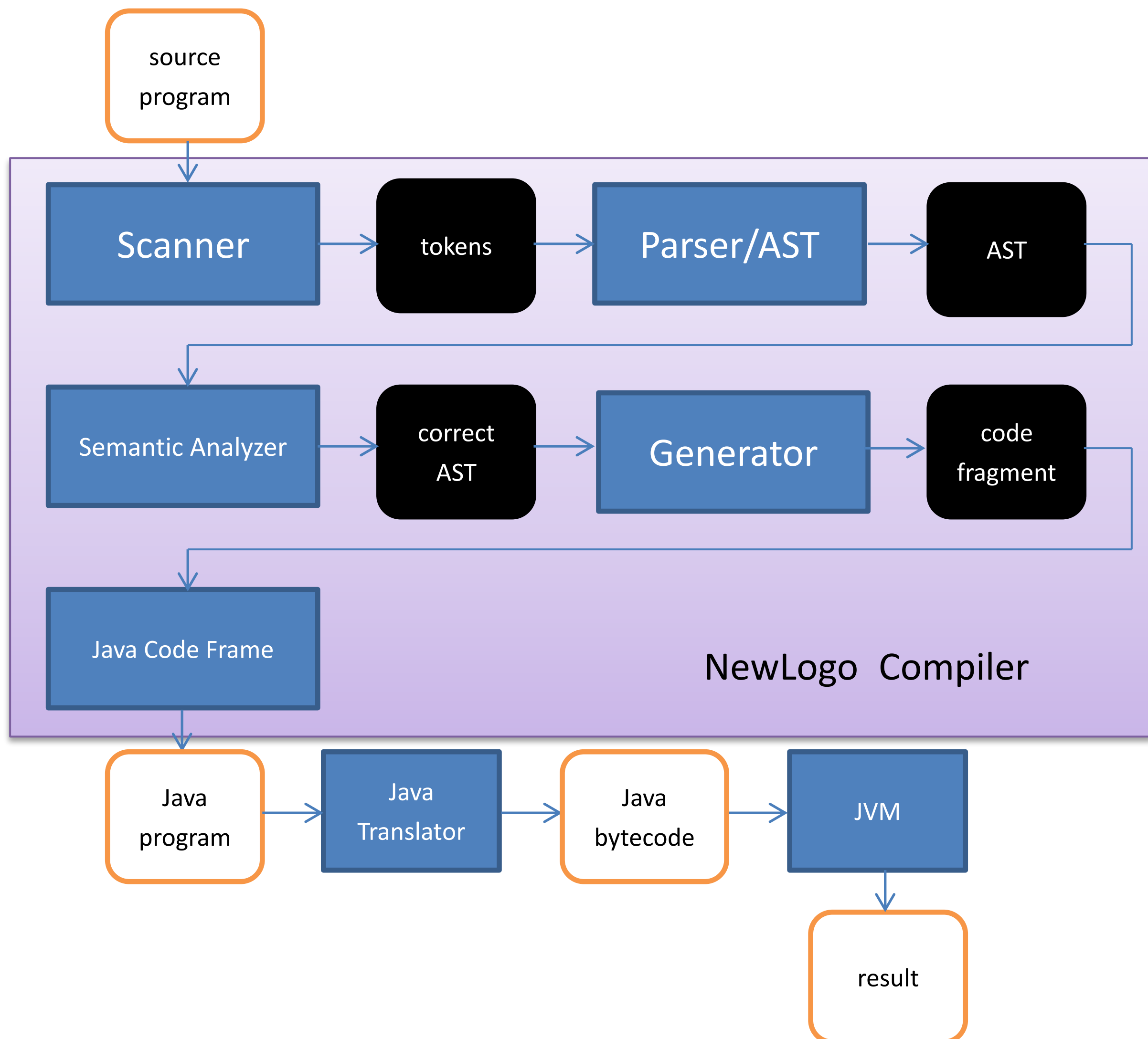


Figure 6.1 Architectural Block Diagram

6.2 Modules

The first module is scanner doing lexical analysis. It reads the stream of characters

making up the source program and groups them into lexemes. Then it outputs a token for each lexeme.

The second module is parser/AST. It uses tokens produced by the scanner to create an abstract syntax tree (AST). In the AST, each interior node represents an operation and the children of the node represent the arguments of the operation.

The third module is semantic. This semantic analyzer uses information with the syntax tree to build symbol tables and checks the source program for semantic consistency with the language definition. If there are any semantic errors, the AST will not be passed to the generator (the next module).

The fourth module is generator. Using the semantically correct syntax tree, it generates code in Java which will be inserted into our Java code frame.

The last module is Java code frame. We have prepared a Java program framework for our implementation of code. Once we put the code from generator in some specified positions of this program, we can have a complete Java program and ready to feed it into the Java translator. The rest is done by Java translator and Java Virtual Machine.

6.3 Files and Work Distribution

Each of us focuses on a module or two. But working as a team, we often discuss together.

The files for each module and our work distribution are shown as follows:

Module	Files	Written by	Modified by
Scanner	header.ml	Pengfei Weng	
	scanner.mll	Pengfei Weng	
Parser/AST	parser.mly	Pengfei Weng	Yuandi Jin, Adarsh Banad
	ast.ml	Pengfei Weng	Yuandi Jin, Adarsh Banad
	parser.mli	Pengfei Weng	
Semantic Analyzer	semantic.ml	Yang Pan, Adarsh Banad	All other members
	sementic.mli	Yang Pan, Adarsh Banad	
Generator	generator.ml	Yuandi Jin, Adarsh Banad	
	generator.mli	Yuandi Jin, Adarsh Banad	
Java code frame	NewLogoPanel (copy).java	Shulei Zhao	
	NewLogoFrame.java	Shulei Zhao	

Table 6.1 Files and Work Distribution

7. Development and Run-time Environment

7.1 Software Development Environment

7.1.1 Eclipse

Eclipse is a multi-language software development environment comprising an integrated development environment (IDE) and an extensible plug-in system. The Eclipse Java Development Tools were used to develop and test the code in our Java template file.

7.1.2 Dropbox

Dropbox is a Web-based file hosting service that uses networked storage to enable users to store and share files and folders with others across the Internet using file synchronization. With synchronization across all our computers and version control, Dropbox was used to store our code.

7.1.3 OCaml

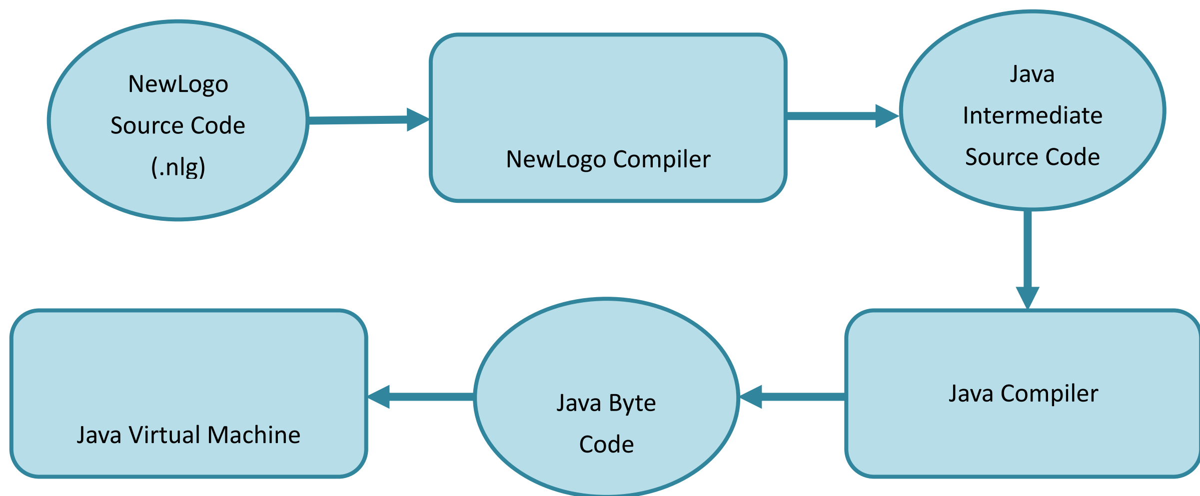
OCaml, originally known as Objective Caml, is the main implementation of the Caml programming language, which is a dialect of the ML programming language family. We programmed our compiler in OCaml.

7.2 Makefile

```
# Makefile for PLT

# VARIOUS VARIABLES
LEX = ocamllex
YACC = ocamlyacc

CC = ocamlc
# type "make test" to test
test: compiler
	./compiler -c ./test.nlg ./NewLogoPanel.java
	javac –nowarn NewLogoFrame.java
	java NewLogoFrame

.PHONY : test
```

```
all: compiler

compiler: parser.cmo scanner.cmo compiler.cmo semantic.cmo generator.cmo header.cmo
	$(CC) -o compiler header.cmo ast.cmo parser.cmo scanner.cmo semantic.cmo generator.cmo compiler.cmo

header.cmo: header.ml
	$(CC) -c header.ml

scanner.ml: scanner.mll header.cmo
	$(LEX) scanner.mll

scanner.cmo: scanner.ml parser.cmo ast.cmo
	$(CC) -c scanner.ml

parser.mli parser.ml: parser.mly scanner.ml
	$(YACC) -v parser.mly

semantic.cmo: semantic.ml semantic.cmi
	$(CC) -c semantic.ml

semantic.cmi: semantic.mli ast.cmo
	$(CC) -c semantic.mli

ast.cmo: ast.ml
	$(CC) -c ast.ml

parser.cmo: parser.cmi parser.ml
	$(CC) -c parser.ml

parser.cmi: parser.mli ast.cmo
	$(CC) -c parser.mli

compiler.cmo: compiler.ml semantic.cmo ast.cmo generator.cmo
	$(CC) -c compiler.ml

generator.cmi: generator.mli ast.cmo
	$(CC) -c generator.mli

generator.cmo: generator.ml generator.cmi
	$(CC) -c generator.ml
```

7.3 Run-time Environment



The NewLogo compiler generates intermediate Java source code. A Java compiler is required in order for the NewLogo compiler to create a runnable program, which is packaged with the Java Development Kit (JDK). The intermediate Java source code is compiled into Java byte code, which requires the Java Run-time Environment (JRE) in order to be run. The compiled Java byte code is run on the Java Virtual Machine, which produces the program's output.

8. Test Plan

In the test folder, we generated plenty of positive and negative test cases in order to test our compiler.

8.1 Simple Function Tests

We will start by testing some simple cases. NewLogo enables user to draw forward. Here is a test to draw a line length 20.

```
void main ( ) {
     int a = 20;
     forward a;
}
```

The result is shown in Figure 8.1

We can then draw a circle as shown in Figure 8.2,

```
void main(){
   for(int j = 0; j < 60; j++){
      forward 20;
      right 6;
   }
}
```

Also, we could draw a rectangle, shown in Figure 8.3, and paint it with color as you want.

```
void main(){
   fillstart;
   for(int j = 0; j < 4; j++){
      forward 200;
      right 90;
   }
   fillend;
}
```

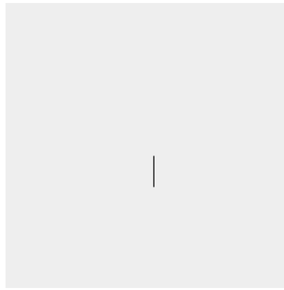


Figure 8.1 A Line

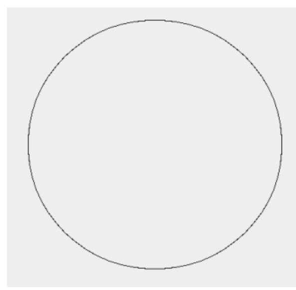


Figure 8.2 A Circle

Figure 8.3 A Black Rectangle

8.2 Complex Test Case

Here are some elegant examples. The first case is to draw three colorful flowers, as shown in 8.4. The case includes function definitions.

```
void drawFlower(int n, int m){
   for(int i = 0; i < n; i++){
      pcolor i;
      fillstart;
      forward m;
      right 40;
      forward m;
      right 140;
      forward m;
      right 40;
      forward m;
      right 140;
      fillend;
      right 40;
   }
   return;
}

void drawFlowers(){
   for(int i = 0; i < 3; i++){
      pdown;
      drawFlower(9, 20);
      pup;
      forward 70;
   }
   return;
}

void main ( ) {
```

```
    drawFlowers();
    return;
}
```

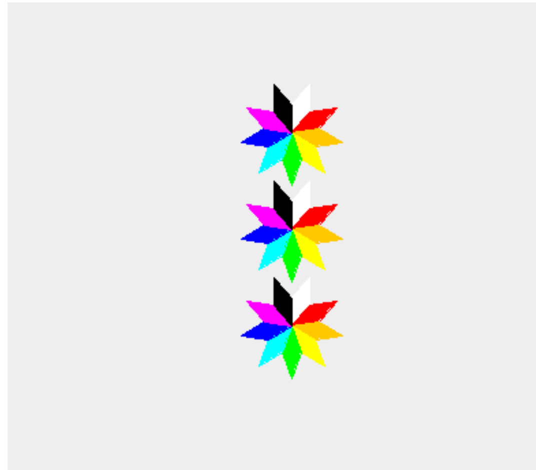
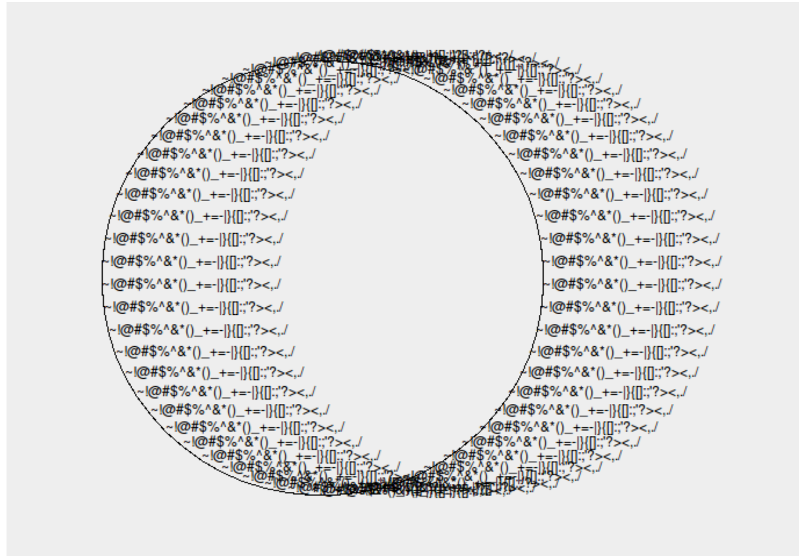


Figure 8.4 Three Colorful Flowers

The second case use the string to manipulate 3D effects, as shown in 8.5.

```
int notmain ( ) {
    int a = 50;
    boolean b = true;
    boolean c = false;
    int i = 2*(8 - 4);
    return i;
}

void main ( ) {
    int a = 50;
    boolean b = true;
    boolean c = false;
    int i = 2*(8-4);
    String s = "~!@#$%^&*()_+=-|}{[]:;'?><,./ ";
    for(int j = 0; j < 60; j++){
        forward 20;
        print s;
        right 6;
    }
}
```



8.5 3D-Effect

8.3 Testing Method and Testing Automation

To guarantee that the compiler can work correctly, we use regression testing for each piece of code.

Initially we chose test cases by selecting basic features such as Forward, Backward, and Paint Color etc, to ensure the compiler works as expected.

The next step was to design complicated functions such as drawing flags or flowers that use different statement, such as for and while, combined together.

Finally we assigned and modified the value of each data type in different ways, and check if the compiler still works as expected.

9. Conclusions

9.1 Lessons Learned as a Team

Work as a team but not five individuals: Working together seems to be less efficient than working separately, but in this project, working separately before discussing the details of the work could only make the work much more complicated, since the structure of the whole compiler is relatively more complex. If all the team members can sit together, the communication will be very convenient and this will help to finish the work earlier.

Remember the roles but do not be limited by the roles: After the roles are assigned, each member should remember their won responsibilities and try to solve problems of their roles as soon as possible. Keep the structure of the team clear could also help to find the right person for specific problems or queries. However, it is possible that some team members meet serious problems or some part of work is obviously more complicated than others, so other team members should also try the best to help. Team members should understand both of the differences and connections among the roles.

9.2 Lessons Learned by Each Team Member

Shulei Zhao (Project Manager)

The role of the project manager in the team seems to be the "least technical" one, but actually it is an important role to guarantee the quality of the team's work and set the height level to the project. Project managers should take their responsibilities and keep the information transferred among all team members without any delay.

On the other hand, actually all team member have the responsibility to keep the whole team work "seamlessly". If one gets some new idea or finds some problems, he should speak out as early as possible.

Pengfei Weng (Language and Tools Guru)

Never doubt that you can be a guru.

Care more about the user, don't design a confusing grammar.

Don't be too ambitious to design a language like C++. Practically, you only need to build a really small part of it.

Study the existing prominent language, from which you will learn a view of designing a language.

Yuandi Jin (System Architect)

To clearly understand the work flow is probably the most important in the whole process of this project. We can start only if we know the work flow well. The compiler we build is an entire system consisting of different modules. In order to make it work, every member should know the role of every part.

This project starts at the beginning of the course. If we want to start early, there can be some parts that we have not studied in class yet. And in fact, not every part is covered in class. So for these parts, learning them first by ourselves is very important. For example, the method to build the generator in our NewLogo compiler is not discussed much in class. To learn from projects of previous students may be a good way.

Adarsh Banad (System Integrator)

Start early: Semester long projects take a semester to complete, and so it is never too early to start.

Ask for help: TAs, classmates and team members can all offer advice and point you in the right direction when you're stuck and unsure how to proceed.

Yang Pan (Tester and Validator)

NewLogo is a great project to understand in the course Programming Languages and Translators. It is helpful as we have gone through most of the concepts introduced in the course. What we have learnt in the lectures brings the theoretic background and basis, while what we have fulfilled in the project brings the experience which broadens my sight in programming languages and adds up to my knowledge.

I am responsible for semantic checking. It is not easy to generate a symbol table for error checking. Fortunately, after step by step understanding the knowledge in class, as well as the code in the book, the symbol table is finally finished. I also need to understand the grammar and syntax tree in order to finish the semantic part. Thanks for my teammate to help me fast understand those parts of the project.

I have learned how to use Linux, how to program using OCaml, how to check error, and how to do the testing. I have learned a lot from this project.

9.3 Advice for Future Teams

Start early: This suggestion will appear in the first place almost every time. Problems could be much more complicated than you have ever imagined. This project does require a lot of time and work to complete. Some parts of the code can't be written until previous parts are near completion. For example, semantic error checking requires the AST in order to be written. This is another reason to start early.

Keep the team agile: Do not be limited by the roles of members, it is still necessary to work together and help each other in some cases. Make modifications on the original plan whenever you want, but as early as possible.

Try new tools: Although Lex and Yacc are introduced in the lectures and in the textbook, there is still no problem if you decide to use another language (e.g. OCaml). Since the basic knowledge of the compilers will not change, using a new language can bring you the same destination. If you are to use a new language like OCaml, learn it as soon as possible! It is very interesting. Online tutorials are very helpful.

9.4 Suggestions for Instructor

We hope more materials about the tools and software used to build the compiler (e.g. ANTLR, OCaml, Yacc, etc.) will be introduced. It will help students save a lot of time to build their compilers.

We are also very interested in the differences and comparisons between the current popular languages (e.g. Java, C++, Python, Ruby, etc.). For example, in what areas they could perform as powerful languages and what are their main weaknesses

We also hope the professor could introduce something about the future and developing trends of the programming languages. For example, will the object-oriented language still be popular in next 50 years?

Appendix Source Code

header.ml

*Written by Pengfei Weng

```
exception IllegalInputError of string * int * int

(*the offset of current line*)
let columnstart lexbuf =
    let pos = lexbuf.Lexing.lex_curr_p in
    pos.Lexing.pos_cnum - pos.Lexing.pos_bol

(*current line number*)
let linecounter lexbuf =
    let pos = lexbuf.Lexing.lex_curr_p in
    pos.Lexing.pos_lnum


(*deal with the accurate pos_lnum and pos_bol value, the function
is from internet*)
let incr_linenum lexbuf =
    let pos = lexbuf.Lexing.lex_curr_p in
    lexbuf.Lexing.lex_curr_p <- { pos with
        Lexing.pos_lnum = pos.Lexing.pos_lnum + 1;
        Lexing.pos_bol = pos.Lexing.pos_cnum;
    }
```

scanner.mll

*Written by Pengfei Weng

```
{
open Header
open Ast
open Parser
}

(*define some useful components for convienence*)
let letter = ['a'-'z''A'-'Z']
let digit = ['0'-'9']
let identify = letter(letter|digit|'_')*
let other = ('~'|'!'|'@'|'#'|'$'|'%'|'^'|'&'|'*'|'('|')'|'_'|'-
'|'+'|'='|'|'|'\\'|']'|'}'|'['|'{'|'\''|';'|':'|'/'|'?'|'>'|'.'|'
<'|','|' ')

let integer = ['1'-'9']digit* | '0'
let double = integer"."digit+

let str = '"'(letter|digit|other)*'"'

rule token = parse
  [' ' '\t' '\r']    { token lexbuf } (*skip all the blanks*)
| [ '\n' ]
{
    line_num := !line_num + 1;  (*line_num is a reference
variable defined in Ast.ml*)
    incr_linenum lexbuf;   (*this function defined in Header.ml*)
    token lexbuf
}
| "#" { comment lexbuf }
| '(' { LPAREN }
| ')' { RPAREN }
| '[' { LBRACK }
| ']' { RBRACK }
| '{' { LBRACE }
| '}' { RBRACE }
| ';' { SEMI }
| ':' { COLON }
| ',' { COMMA }
| "++" { DOUBLEPLUS } (* '' changed to "" -Yuandi *)
| "--" { DOUBLEMINUS } (* '' changed to "" -Yuandi *)
```

```
| '+' { PLUS }
| '-' { MINUS }
| '*' { TIMES }
| '/' { DIVIDE }
| '=' { ASSIGN }
| "==" { EQ }
| "!=" { NEQ }
| '<' { ST }
| "<=" { LEQ }
| '>' { GT }
| ">=" { GEQ }
| str as lxm
{
    try
        CONTENT(lxm)
    with e ->
        let errormsg = "Syntax Error: String '" ^ lxm ^ "'
overflow at line: " ^ string_of_int !line_num in
        Printf.printf "%s\n"  errormsg;
        CONTENT(string_of_int 0)
}
| "main" { MAIN } (* new *)
| "if" { IF }
| "else" { ELSE }
| "while" { WHILE }
| "for" { FOR }
| "return" { RETURN }
| "true" {TRUE}
| "false" {FALSE}
| "int" { INT }
| "double" { DOUBLE }
| "boolean" { BOOL }
| "String" { STRING }
| "void" { VOID }
| "!" { NOT }
| "&&" { AND }
| "||" { OR }
(*Drawing commands*)
| "forward" { FD }
| "back" { BK }
| "right" { RT }
| "left" { LT }
| "pdown" { PD }
| "pup" { PU }
| "pcolor" { PC }
| "pwidth" { PW }
```

```
| "fillstart" { FILLS } (* new *)
| "fillend" { FILLE } (* new *)
| "print" { PRINT } (* new *)
| "reset" { RESET }
| integer as lxm
{
    try
        INTV(int_of_string lxm)
    with e ->
        let errormsg = "Syntax Error: Integer '" ^ lxm ^ "'
overflow at line: " ^ string_of_int !line_num in
        Printf.printf "%s\n"  errormsg;
        INTV(0)
}
| double as lxm
{
    try
        DOUBLEV(float_of_string lxm)
    with e ->
        let errormsg = "Syntax Error: Double value '" ^ lxm ^ "'
overflow at line: " ^ string_of_int !line_num in
        Printf.printf "%s\n"  errormsg;
        DOUBLEV(float_of_int 0)
}
| identify as lxm { ID(lxm) }
| eof { EOF }
| _ as char  (*All the other input is regarded as illegal input!*)
{
    let line = linecounter lexbuf in
    let col = columnstart lexbuf in
    raise ( IllegalInputError(Char.escaped char, line, col) );
}



and comment = parse
    "\n" { token lexbuf } (*if change the line, it goes to the
token*)
    | _    { comment lexbuf} (*else it's still a comment*)
```

parser.mly

*Written by Pengfei Weng

*Modified by Yuandi Jin and Adarsh Banad

```
%{
open Header
open Ast

let parsing_succeed = ref true


let nullexpr = ExprOfCstExpr(CstExprOfInt(0), 0)


let nullstmtlist = [StmtOfExprStmt(ExprStmt(nullexpr))]

%}

%token PLUS MINUS TIMES DIVIDE COLON EOF DOUBLEPLUS DOUBLEMINUS
%token LBRACK RBRACK LPAREN RPAREN LBRACE RBRACE
%token ASSIGN SEMI COMMA
%token INT DOUBLE STRING BOOL VOID
%token MAIN
%token CONTENT

/*drawing related statements*/
%token FD BK RT LT PD PU PC PW FILLS FILLE RESET PRINT
%token IF ELSE WHILE FOR RETURN BREAK CONTINUE

/*boolean operator*/
%token EQ NEQ ST GT GEQ LEQ NOT


%token AND OR NOT
%token TRUE FALSE

%token <int> INTV
%token <float> DOUBLEV
%token <string> ID
%token <string> CONTENT


/*binary operator*/     /* need to add associativity of ADD MINUS etc.
-Yuandi */
%nonassoc NOT
%left OR
%left AND
```

```
%left EQ NEQ
%left ST GT LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE
%nonassoc UMINUS

/*primary_expression operators*/
%left COMMA
%right ASSIGN


%start program /*define the start point*/
%type <Ast.program> program
%%


program:
{ [] }
| free_sentence program { $1 :: $2 }
;


free_sentence:
| function_definition { StcOfFunDef($1) }
| statement { StcOfStmt($1) }
;


/* function definition part*/
function_definition:
| datatype MAIN LPAREN parameter_list RPAREN LBRACE statement_list
RBRACE { MAINFunctionDef($1,$4,$7) }
| datatype ID LPAREN parameter_list RPAREN LBRACE statement_list RBRACE
{ FunctionDef($1,$2,$4,$7) }
;

parameter_list:
{ [] }
| parameter { [$1] }
| parameter COMMA parameter_list { $1 :: $3 }
;

parameter:
| datatype ID { ParameterType($1,$2) }
;




statement:
```

```
| expression_statement { StmtOfExprStmt($1) }
| draw_statement { StmtOfDrawStmt($1) }
| LBRACE statement_list RBRACE { StmtList($2) }
| declaration_statement { StmtOfDeclaration($1) }
| iteration_statement  { StmtOfIterationStmt($1) }
| selection_statement  { StmtOfSelectionStmt($1) }
| jump_statement { StmtOfJumpStmt($1) }
;



expression_statement:
| expression SEMI { ExprStmt($1) }
| expression error
{
      Printf.printf "%s" "Syntax Error: Are you missing a ';'\n";
      parsing_succeed := false;
      ExprStmt($1)
}



/*this part uses binary operation*/
draw_statement:
| FD expression SEMI { DrawStmt(FD, $2) }
| BK expression SEMI { DrawStmt(BK, $2) }
| RT expression SEMI { DrawStmt(RT, $2) }
| LT expression SEMI { DrawStmt(LT, $2) }
| PD SEMI { DrawStmtWithoutExp(PD) }
| PU SEMI { DrawStmtWithoutExp(PU) }
| RESET SEMI { DrawStmtWithoutExp(RESET) }
| PC expression SEMI { DrawStmt(PC, $2) }
| PW expression SEMI { DrawStmt(PW, $2) }
| FILLS SEMI { DrawStmtWithoutExp(FILLS) } /* new */
| FILLE SEMI { DrawStmtWithoutExp(FILLE) } /* new */
| PRINT expression SEMI { DrawStmt(PRINT, $2) } /* new */
;



statement_list:
| statement { [$1] }  /*By Jason*/
| statement statement_list  { $1 :: $2 }
;



declaration_statement:
| declarator { [$1] }
```

```
| declarator declaration_statement { $1 :: $2 }
;


declarator:
| datatype ID ASSIGN expression SEMI{ DECL($1,$2,!line_num,$4) }
;




selection_statement:
| IF LPAREN expression RPAREN LBRACE statement_list RBRACE
{ SelectionStmtIf($3,$6) }
| IF LPAREN expression RPAREN LBRACE statement_list RBRACE ELSE LBRACE
statement_list RBRACE { SelectionStmtIfElse($3,$6,$10) }
;


iteration_statement:
| WHILE LPAREN expression RPAREN LBRACE statement_list RBRACE
{ IterationStmtOfWhile($3,$6) }
| FOR LPAREN for_init_statement expression SEMI expression RPAREN
LBRACE statement_list RBRACE { IterationStmtOfFor($3,$4,$6,$9) }
;


for_init_statement:
| expression_statement { FORWithExpr($1) }
| declaration_statement { FORWithDec($1) }
;



jump_statement:
| BREAK { JumpStmtOfBreak(!line_num) }
| CONTINUE { JumpStmtOfContinue(!line_num) }
| RETURN expression SEMI { JumpStmtOfReturnExpr($2) }
| RETURN SEMI { JumpStmtOfReturn(!line_num) }
;


datatype:
| INT { TypeOfInt }
| DOUBLE { TypeOfDouble }
| BOOL { TypeOfBoolean }
| VOID { TypeOfVoid }
| STRING { TypeOfString }
;
```

```
/* By Jason */
expression:
| constant_expression { ExprOfCstExpr($1, !line_num) }
| LPAREN expression RPAREN { ExprWithParen($2) }
| LPAREN error
{
    Printf.printf "%s\n" "'(' is mismatched";
    nullexpr
}
| lvalue ASSIGN expression { ExprOfAssign($1,$3) }
| expression PLUS expression{ ExprOfBineryOp($1, PLUS, $3) }
| expression MINUS expression{ ExprOfBineryOp($1, MINUS, $3) }
| MINUS expression{ ExprOfBineryOp(nullexpr, MINUS, $2) } /* Modified
by Yuandi. */
| expression TIMES expression{ ExprOfBineryOp($1, TIMES, $3) }
| expression DIVIDE expression{ ExprOfBineryOp($1, DIVIDE, $3) }
| expression NEQ expression { ExprOfBineryOp($1, NEQ, $3) }
| expression EQ expression { ExprOfBineryOp($1, EQ, $3) }
| expression ST expression { ExprOfBineryOp($1, ST, $3) }
| expression GT expression { ExprOfBineryOp($1, GT, $3) }
| expression LEQ expression { ExprOfBineryOp($1, LEQ, $3) }
| expression GEQ expression { ExprOfBineryOp($1, GEQ, $3) }
| expression AND expression { ExprOfBineryOp($1, AND, $3) }
| expression OR expression { ExprOfBineryOp($1, OR, $3) }
| expression DOUBLEPLUS { ExprOfSingleOp(DOUBLEPLUS, $1) } /* Modified
by Yuandi. Changed $2 to $1, ID to expression. */
| expression DOUBLEMINUS { ExprOfSingleOp(DOUBLEMINUS, $1) } /*
Modified by Yuandi. Changed $2 to $1, ID to expression. */
| NOT expression { ExprOfSingleOp(NOT, $2) }
| ID LPAREN argument_expression_list RPAREN { ExprOfFunction($1,
$3, !line_num) }
;


constant_expression:
| INTV  { CstExprOfInt($1) }
| DOUBLEV { CstExprOfDouble($1) }
| TRUE  { CstExprOfTrue }
| FALSE { CstExprOfFalse }
| ID { CstExprOfID($1, !line_num) }
| CONTENT { CstExprOfSTRING($1) }
;

/*identifier:
| ID { Identifier($1, !line_num) }
*/


lvalue:
```

```
|ID { LvalueOfIdentifier($1, !line_num) }
;


argument_expression_list:
{ [] } /*modified by Pengfei*/
| expression { [$1] }
| expression COMMA argument_expression_list { $1 :: $3 }
;
```

Parser.mli

*Written by Pengfei Weng

```
type token =
  | PLUS
  | MINUS
  | TIMES
  | DIVIDE
  | COLON
  | EOF
  | DOUBLEPLUS
  | DOUBLEMINUS
  | LBRACK
  | RBRACK
  | LPAREN
  | RPAREN
  | LBRACE
  | RBRACE
  | ASSIGN
  | SEMI
  | COMMA
  | INT
  | DOUBLE
  | STRING
  | BOOL
  | VOID
  | MAIN
  | CONTENT of (string)
  | FD
  | BK
  | RT
  | LT
  | PD
  | PU
  | PC
  | PW
  | FILLS
  | FILLE
  | RESET
  | PRINT
  | IF
  | ELSE
  | WHILE
  | FOR
```

```
  | RETURN
  | BREAK
  | CONTINUE
  | EQ
  | NEQ
  | ST
  | GT
  | GEQ
  | LEQ
  | NOT
  | AND
  | OR
  | TRUE
  | FALSE
  | INTV of (int)
  | DOUBLEV of (float)
  | ID of (string)

val program :
  (Lexing.lexbuf  -> token) -> Lexing.lexbuf -> Ast.program
```

Ast.ml

*Written by Pengfei Weng

*Modified by Yuandi Jin and Adarsh Banad

```
(*
 * Short:
 *
 * var  - variable
 * def  - definition
 * dec  - declaration
 * cst  - const
 * stc  - sentence
 * func - function
 * expr - expression
 * stmt - statement
 *
 *)

type operator = PLUS | MINUS | TIMES | DIVIDE | ASSIGN | EQ | NEQ
| ST | LEQ | GT |GEQ | AND | OR | NOT | DOUBLEPLUS | DOUBLEMINUS
(* Added NOT *)
(* Added DOUBLEPLUS DOUBLEMINUS *)
type action = FD | BK | RT | LT | PD | PU | PC | PW | FILLS |
FILLE | RESET | PRINT (* Added FILLS FILLE PRINT *)

let line_num = ref 1;;

(*rules*)
type program = stc list

and stc =
    StcOfStmt of stmt
    | StcOfFunDef of func_def

and func_def =
    MAINFunctionDef of datatype * parameter_list * stmt list
    | FunctionDef of datatype * string * parameter_list * stmt
list (* Modified by Yuandi *)

and parameter_list = parameter list


and parameter =
```

```
    ParameterType of datatype * string (* Modified by Yuandi *)

and datatype =
    TypeOfInt
    | TypeOfDouble
    | TypeOfBoolean
    | TypeOfVoid
    | TypeOfString
    | TypeOfUndeclare



and stmt =
    StmtOfDeclaration of declaration_stmt
    | StmtOfExprStmt of expr_stmt
    | StmtOfSelectionStmt of selection_stmt
    | StmtOfIterationStmt of iteration_stmt
    | StmtOfJumpStmt of jump_stmt
    | StmtOfDrawStmt of draw_stmt
    | StmtList of statement_list

and statement_list = stmt list

(* need to initiate parameter_list? YES *)


and declaration_stmt = declarator list

and declarator =
    DECL of datatype * string * int * expr  (* Initialization is
required in declairation -Yuandi *)

and expr_stmt =
    ExprStmt of expr

and draw_stmt =
    DrawStmt of action * expr
    | DrawStmtWithoutExp of action

and selection_stmt =
    SelectionStmtIf of expr * statement_list
    | SelectionStmtIfElse of expr * statement_list *
statement_list

and iteration_stmt =
    IterationStmtOfWhile of expr * statement_list
```

```
    | IterationStmtOfFor of for_init_statement * expr * expr *
statement_list

and for_init_statement =
    FORWithExpr of expr_stmt
    | FORWithDec of declaration_stmt

and jump_stmt =
    JumpStmtOfContinue of int
    | JumpStmtOfBreak of int
    | JumpStmtOfReturnExpr of expr
    | JumpStmtOfReturn of int


(* Modified by Jason *)
and expr =
    ExprOfCstExpr of cst_expr * int
    | ExprOfAssign of lvalue * expr
    | ExprOfBineryOp of expr * operator * expr
    | ExprOfSingleOp of operator * expr
    | ExprOfFunction of string * argument_list * int
    | ExprWithParen of expr

and cst_expr =
    CstExprOfInt of int
    | CstExprOfDouble of float
    | CstExprOfTrue
    | CstExprOfFalse
    | CstExprOfID of string * int
    | CstExprOfSTRING of string


and argument_list = expr list


and lvalue =
    LvalueOfIdentifier of string * int
```

semantic.ml

*Written by Yang Pan and Adarsh Banad
*Modified by all other members

```
open Ast

module StringMap = Map.Make(String)

module Semantic =
struct

(*function with different return type and parameter type*)
type func_record =
{
    return_tp : datatype;
    tplist : datatype list;
}


(*symbol table for function and for each local region*)
type env =
{
    mutable funcmap : func_record StringMap.t;
    mutable table_list : (datatype StringMap.t) list;
    mutable objmap :  (datatype StringMap.t) StringMap.t;
}


(* variable of type env, it is global, table_list is the array
for map *)
let ienv =
{
    funcmap = StringMap.empty;
    table_list = [StringMap.empty];
    objmap = StringMap.empty;
}


type tpli =
{
    line : int;
    tp : datatype;
}
```

```
(* whether no error exists *)
let checking_succeed = ref true

(*whether current node is in loop*)
let nLoop = ref 0
let inFunc = ref 0
let return = ref 0
let tpFunc = ref ""
let currFuncName = ref ""

let numberOfMain = ref 0

(*whether has break or not*)
let bHasBreak = ref false

let bInLoop () =
    if !nLoop > 0 then
        true
    else
        false

let bInFunc () =
    if !inFunc > 0 then
        true
    else
        false

let returned () =
    if !return > 0 then
        true
    else
        false


(*transfer datatype into string for compare*)
let rec string_of_datatype (t: datatype) =
    match t with
    TypeOfInt -> "int"
    | TypeOfDouble -> "double"
    | TypeOfBoolean -> "bool"
    | TypeOfUndeclare -> "undeclare"
    | TypeOfVoid -> "void"
      | TypeOfString -> "String"


let rec string_of_action (a: action) = (*Added by Adarsh*)
```

```
    match a with
     FD -> "foward" (* forward *)
    | BK -> "back" (* back *)
    | RT -> "right" (* turn right *)
    | LT -> "left" (* turn left *)
    | PD -> "pdown" (* pen down *)
    | PU -> "pup" (* pen up *)
    | RESET -> "reset" (* reset *)
    | PC -> "pcolor" (* pen color *)
    | PW -> "pwidth" (* pen width *)
    | FILLS -> "fillstart" (* fill start *)
    | FILLE -> "fillend" (* fill end *)
    | PRINT -> "print" (* print *)


(* find the type of string s *)
let fold_fun (s, tp) (map: 'a StringMap.t) =
    match tp with
    TypeOfUndeclare ->
        (try
            let tp = StringMap.find s map in
            (s, tp)
        with Not_found ->
            (s, TypeOfUndeclare))
    | _ ->
        (s, tp)

(*check if the current variable has been defined before or not*)
let check_variable (s: string) (li: int) =
    let (_, tpn) =
        List.fold_left fold_fun (s, TypeOfUndeclare)
ienv.table_list
    in
    match tpn with
    TypeOfUndeclare ->
        let errormsg = "Semantic Error: Undeclared variable '" ^
s ^ "' at line " ^ (string_of_int li) in
        Printf.printf "%s\n" errormsg;
        checking_succeed := false;
        { tp = TypeOfUndeclare; line = li}
    | _ ->
        { tp = tpn; line = li}


(*test if the local variable is already exists*)
let find_variable_in_local (s: string) =
```

```
    let local = List.hd ienv.table_list in
    StringMap.mem s local


(*test if the variable is type undeclared, 'let' will store the
variable into variable list*)
let add2table (s: string) (tp: datatype) (li: int) =
    if find_variable_in_local s then
        let errormsg =
            "Semantic Error: Redefinition of variable '" ^ s ^ "'
at line: " ^ string_of_int li ^ "\n" in
        Printf.printf "%s" errormsg;
checking_succeed := false
    else
        let local = List.hd ienv.table_list in
        let local = StringMap.add s tp local in
        ienv.table_list <- local :: List.tl ienv.table_list


let check_int (e1: tpli)  =
    match e1.tp with
    TypeOfInt -> ()
    | TypeOfUndeclare -> ()
    | _ ->
        let errormsg = "Semantic Error: expected type 'int' but
have type '" ^ string_of_datatype e1.tp ^ "' at line " ^
(string_of_int e1.line) in
        Printf.printf "%s\n" errormsg;
        checking_succeed := false


let check_string (e1: tpli) =
    match e1.tp with
    TypeOfString -> ()
    | TypeOfUndeclare -> ()
    | _ ->
        let errormsg = "Semantic Error: expected type 'string'
but have type '" ^ string_of_datatype e1.tp ^ "' at line " ^
(string_of_int e1.line) in
        Printf.printf "%s\n" errormsg;
        checking_succeed := false


let check_bool (e1: tpli)  =
    match e1.tp with
    TypeOfBoolean ->
```

```
        ()
    | TypeOfUndeclare -> ()
    | _ ->()


 let check_rgb (e1: tpli) =
     check_int e1


(*check rules*)

let rec check (lst: stc list) =
ienv.objmap <- StringMap.add "int" StringMap.empty ienv.objmap;
ienv.objmap <- StringMap.add "double" StringMap.empty ienv.objmap;
ienv.objmap <- StringMap.add "bool" StringMap.empty ienv.objmap;
ienv.objmap <- StringMap.add "String" StringMap.empty ienv.objmap;
ienv.objmap <- StringMap.add "void" StringMap.empty ienv.objmap;
    match lst with
    | [] -> ()
    | head::[] -> check_stc head
    | head::tail ->
        check_stc head;
        check tail


and check_stc (s: stc) =
    match s with
    | StcOfFunDef f -> check_fun_dec f;
    | StcOfStmt s -> check_stmt s;


and check_cst_expr (e: cst_expr) (li: int) =
    match e with
    CstExprOfInt (s) -> {tp = TypeOfInt; line = li}
    | CstExprOfDouble (s) -> {tp = TypeOfDouble; line = li}
    | CstExprOfTrue
    | CstExprOfFalse -> {tp = TypeOfBoolean; line = li}
     | CstExprOfID (id, li)-> check_variable id li    |
CstExprOfSTRING  (s) -> {tp = TypeOfString; line = li}

and check_lvalue (l: lvalue)=
    match l with
    LvalueOfIdentifier (id, li) ->
        check_variable id li
```

```
and check_typematch (l: tpli) (v: tpli) =
    if string_of_datatype l.tp = "undeclare" then
        l
    else if string_of_datatype v.tp = "undeclare" then
        v
    else
    if l.tp = v.tp then
        l
    else
        let errormsg = "Semantic Error: expected type '" ^
string_of_datatype l.tp ^ "' but have type'" ^ string_of_datatype
v.tp ^ "' at line " ^ (string_of_int l.line) in
        Printf.printf "%s\n" errormsg;
        checking_succeed := false;
        l


and check_assign (l: tpli) (v: tpli)  =
    check_typematch l v


 and check_bop (e1: tpli) (e2: tpli) =
    if (string_of_datatype e1.tp = "int" || string_of_datatype
e1.tp = "double") then
        check_typematch e1 e2
    else
        let errormsg = "Semantic Error: no operation defined for
type '" ^ string_of_datatype e1.tp ^ "' at line " ^
(string_of_int e1.line) in
        Printf.printf "%s\n" errormsg;
        checking_succeed := false;
        e1


 and check_boolop (e1: tpli) (e2: tpli)  =
    if string_of_datatype e1.tp = "boolean" then
      let e = {
        line = e1.line;
        tp = TypeOfBoolean;
    } in
        check_typematch e1 e2; e

    else (let errormsg = "Semantic Error: no operation defined
for type '" ^ string_of_datatype e1.tp ^ "' at line " ^
(string_of_int e1.line) in
            Printf.printf "%s\n" errormsg;
```

```
            checking_succeed := false;
            {
                line = e1.line;
                tp = TypeOfBoolean;
            });
    {
        line = e1.line;
        tp = TypeOfBoolean;
    }

and get_func_call_para_list (lst: expr list)=
match lst with
    | [] -> [TypeOfVoid]
    | head :: [] -> [(check_expr head).tp]
    | head :: tail ->
        let head = (check_expr head).tp in
        let tail = get_func_call_para_list tail in
        head :: tail


and check_func_call_para (id: string) (lst: expr list) (li: int)=
let lst = get_func_call_para_list lst in
let errormsg = "Semantic Error: Function '" ^ id ^ "' argument do
not match with definition at  line " ^ (string_of_int li) in
     try
          match StringMap.find id ienv.funcmap with
                |lst ->{tp = TypeOfUndeclare; line = li}
                |_ -> Printf.printf "%s\n" errormsg;
                          checking_succeed := false;
                     {tp = TypeOfUndeclare; line = li}
     with Not_found ->
Printf.printf "%s\n" errormsg;
            checking_succeed := false;
            {tp = TypeOfUndeclare; line = li}


and check_func_return_type (id: string) (li: int)=
try

        {tp = (StringMap.find id ienv.funcmap).return_tp; line =
li}
    with Not_found ->
            let errormsg = "Semantic Error: Undeclared function
'" ^ id ^ "' at  line " ^ (string_of_int li) in
            Printf.printf "%s\n" errormsg;
            checking_succeed := false;
```

```
            {tp = TypeOfUndeclare; line = li}


and check_expr (e: expr) =
   match e with
   | ExprOfCstExpr (e, li) -> check_cst_expr e li
   | ExprOfAssign (l, v) ->
      let l = check_lvalue l in
      let v = check_expr v in
      check_assign l v
    | ExprOfBineryOp (e1, op, e2) ->
         let e1 = check_expr e1 in
      let e2 = check_expr e2 in
         (match op with
         | EQ
         | NEQ
         | ST
         | LEQ
         | GT
         | GEQ -> check_bop e1 e2
         | AND
         | OR  -> check_boolop e1 e2
         | _ -> check_bop e1 e2
         )
    | ExprOfSingleOp (op, e) ->
      let e = check_expr e in
         (match op with
         | NOT ->
              check_bool e;
              e
         | _ -> check_bop e e
         )
   | ExprOfFunction (id, lst, li) ->
    ignore(check_func_call_para id lst li);
      check_func_return_type id li;
   | ExprWithParen (e) ->
         check_expr e


and check_expr_stmt (e: expr_stmt) =
   match e with
   ExprStmt e -> ignore(check_expr e); ()


and check_datatype (t: datatype) =
   let st = string_of_datatype t in
```

```
      try
         StringMap.find st ienv.objmap;
      t
    with Not_found ->
            let errormsg = "Semantic Error: Undeclared type '" ^
st ^ "' " in
            Printf.printf "%s\n" errormsg;
            checking_succeed := false;
            TypeOfUndeclare


and check_declarator  (d: declarator) = (*changed by Adarsh*)
      match d with DECL (tp, s, li, ex) ->
           let l = {tp = tp;line = li} in
           let v = check_expr ex in
           (add2table s tp li;
         check_assign l v)


 and check_var_dec (vc:  declarator list) = (*changed by adarsh*)
    match vc with
      |  [] -> ()
    | head::[] -> check_declarators head
    | head::tail ->
         check_declarator head;
         check_var_dec tail;


and check_declarators  (d: declarator) = (*changed by Adarsh*)
      match d with DECL (tp, s, li, ex) ->
           let l = {tp = tp;line = li} in
           let v = check_expr ex in
           (add2table s tp li;
         check_assign_s l v)


and check_typematchs (l: tpli) (v: tpli) =
    if string_of_datatype l.tp = "undeclare" then
        ()
    else if string_of_datatype v.tp = "undeclare" then
        ()
    else if l.tp = v.tp then
        ()
    else
```

```
        let errormsg = "Semantic Error: expected type '" ^
string_of_datatype l.tp ^ "' but have type'" ^ string_of_datatype
v.tp ^ "' at line " ^ (string_of_int l.line) in
        Printf.printf "%s\n" errormsg;
        checking_succeed := false;
        ()


and check_assign_s (l: tpli) (v: tpli)  =
    check_typematchs l v


 and assign_local_var (tp: datatype) (id: string) =
    add2table id tp 0


 and check_para (para: parameter) =
    match para with ParameterType (tp, id) ->
    let tp = check_datatype tp in
    assign_local_var tp id;
    tp;


 and check_paralist (paralist: parameter list) =
    match paralist with
    | [] -> [TypeOfVoid]
    | head::[] -> [check_para head]
    | head::tail ->
        let head = check_para head in
        let tail = check_paralist tail in
        head :: tail


 and assign_func (id: string) (tplist: datatype list) (tp :
datatype) =
   match  StringMap.mem id ienv.funcmap with
|false ->
    let item =
    {
        return_tp = tp;
        tplist = tplist;
    } in
    ienv.funcmap <- StringMap.add id item ienv.funcmap
|true -> let errormsg =
            "Semantic Error: Redefinition of function '" ^ id ^
"'\n" in
```

```
        Printf.printf "%s" errormsg;
checking_succeed := false


and check_id_para (s:string)(plst: parameter_list) (tp:
datatype)=
    let tplist = check_paralist plst in
    currFuncName := s;
    assign_func s tplist tp



 and check_func_body (lst: stmt list) =
    match lst with
    | [] -> ()
    | head::[] -> check_stmt head
    | head::tail ->
        check_stmt head;
        check_func_body tail


 and check_fun_dec (f: func_def) =
    match f with
    | MAINFunctionDef (dt, plst, cs) ->
      numberOfMain := !numberOfMain + 1;
          if !numberOfMain <> 1 then
              let errormsg = "Semantic Error: Redefinition of
main." in
                      Printf.printf "%s\n" errormsg;
                checking_succeed := false;
    | FunctionDef (dt, s, plst, cs) ->
      ienv.table_list <- StringMap.empty :: ienv.table_list;
      check_id_para s plst dt;
      let dt = check_datatype dt in
           tpFunc := string_of_datatype dt;
           inFunc :=1;
           check_func_body cs;
      match not(returned ()) && (!tpFunc <> "void") with
           |true ->let errormsg = "Semantic Error: Function '"
^ !currFuncName ^ "' without return." ^ !tpFunc in
                      Printf.printf "%s\n" errormsg;
                checking_succeed := false;
                ienv.table_list <- List.tl ienv.table_list;
                inFunc := 0;
                tpFunc := "";
                return := 0;
```

```
                  currFuncName := ""
               |false -> ienv.table_list <- List.tl ienv.table_list;
                  inFunc := 0;
                  tpFunc := "";
                  return := 0;
                  currFuncName := ""


and check_stmt (s: stmt) =
    match s with
    | StmtOfDeclaration v -> check_var_dec v;
    | StmtOfExprStmt e -> check_expr_stmt e
    | StmtOfSelectionStmt s -> check_selection_stmt s
    | StmtOfIterationStmt is -> check_iteration_stmt is
    | StmtOfJumpStmt js -> check_jump_stmt js
    | StmtOfDrawStmt ds -> check_draw_stmt ds
    | StmtList sl -> check_stmt_list sl


 and check_draw_stmt (ds: draw_stmt) =
    match ds with
    | DrawStmt (a, e) ->
     check_act_expr a e
    | DrawStmtWithoutExp act -> check_action act


 and check_action (act: action)=
    match act with
    | FILLE
    | FILLS
    | RESET
     | PD
     | PU  -> ()
     | _-> let errormsg =
            "Semantic Error: Action '" ^ string_of_action act ^
"' must has parameter \n" in
        Printf.printf "%s" errormsg;
     checking_succeed := false


 and check_act_expr (a:action)(e:expr) =
     match a with
     | FD
     | BK
     | PC
     | PW -> let e = check_expr e in
```

```
                check_int e
      | RT
      | LT   -> ()
      | PRINT -> let e =check_expr e in
                    check_string e


 and check_jump_stmt (s: jump_stmt) =
    match s with
    JumpStmtOfContinue li->
        if not(bInLoop ()) then
         let errormsg = "Semantic Error: Continue statement must
be in loop at line " ^ (string_of_int li) in
          Printf.printf "%s\n" errormsg;
           checking_succeed := false
    | JumpStmtOfBreak li->
        (if not(bInLoop ()) then
         let errormsg = "Semantic Error: Break statement must be
in loop at line " ^ (string_of_int li) in
          Printf.printf "%s\n" errormsg;
           checking_succeed := false);
      bHasBreak := true
    | JumpStmtOfReturn li ->
        if not(bInFunc ()) then
         let errormsg = "Semantic Error: Return statement must be
in function at line " ^ (string_of_int li) in
          Printf.printf "%s\n" errormsg;
           checking_succeed := false
      else if (!tpFunc <> "void") then
         let errormsg = "Semantic Error: Return type should be '"
^ !tpFunc ^ "' but is 'void' at line " ^ (string_of_int li) in
          Printf.printf "%s\n" errormsg;
           checking_succeed := false
      else return := !return + 1;
    | JumpStmtOfReturnExpr e ->
      let e = check_expr e in
      if not(bInFunc ()) then
         let errormsg = "Semantic Error: Return statement must be
in function at line " ^ (string_of_int e.line) in
          Printf.printf "%s\n" errormsg;
           checking_succeed := false
      else if (!tpFunc <> string_of_datatype e.tp) then
         let errormsg = "Semantic Error: Return type should be '"
^ !tpFunc ^ "' but is '" ^ string_of_datatype e.tp ^ "' at line "
^ (string_of_int e.line) in
          Printf.printf "%s\n" errormsg;
```

```
            checking_succeed := false
        else return := !return + 1;


and check_for_init_statement(s: for_init_statement)=
match s with
|  FORWithExpr (e) -> check_expr_stmt e
|  FORWithDec  (d) -> check_var_dec d


 and check_iteration_stmt (s: iteration_stmt) =
    nLoop := !nLoop + 1;
    ienv.table_list <- StringMap.empty :: ienv.table_list;
    match s with
    | IterationStmtOfWhile (e, lst) ->
        check_stmt_list lst;
        let e = check_expr e in
        check_bool e

    | IterationStmtOfFor (s_init, e_condition, e_loop, lst) ->
           check_for_init_statement s_init;
           check_stmt_list lst;
           check_expr e_loop;
           let e = check_expr e_condition in

        bHasBreak := false;

    ienv.table_list <- List.tl ienv.table_list;
    nLoop := !nLoop - 1


 and check_selection_stmt (s: selection_stmt) =  (*changed by
adarsh*)
    match s with
    | SelectionStmtIf(e, lst) ->
        check_stmt_list lst;
        let e = check_expr e in
        check_bool e

    | SelectionStmtIfElse(e, lst, elst) ->
        check_stmt_list lst;
        check_stmt_list elst;
        let e = check_expr e in
        check_bool e
```

```
 and check_stmt_list (lst: stmt list) =
    match lst with
    | [] -> ()
    | head :: [] -> check_stmt head
    | head :: tail ->
        check_stmt head;
        check_stmt_list tail

end
```

semantic.mli

*Written by Yang Pan

```
module Semantic:
    sig
    val check : Ast.program -> unit
    val checking_succeed : bool ref
    end
```

generator.ml

*Written by Yuandi Jin and Adarsh Banad

```
(* parameter_list is a list. stmt_list is a list, argument_list is also
a list. *)
(* after AST, "()" is not in parameter_list, "{}" is not in
compound_stmt. *)
(* parameter is FORMAL PARAMETER, argument is ACTUAL PARAMETER. *)
(* - Yuandi *)

open Ast

module StringMap = Map.Make(String)

module Generator =
struct


let rec generate (lst: stc list) =
    match lst with
    | [] -> "\n"
    | head::[] -> generate_stc head ^ "\n"
    | head::tail ->
       let h = generate_stc head in
       let t = generate tail in
       h ^ "\n" ^ t


and generate_stc (s: stc) =
    match s with
    | StcOfStmt s -> generate_stmt s
    | StcOfFunDef f -> generate_func_def f

and generate_stmt (s: stmt) =
    match s with
    | StmtOfDeclaration ds -> generate_dec_stmt ds

    | StmtOfExprStmt es -> generate_expr_stmt es

    | StmtOfSelectionStmt ss -> generate_selection_stmt ss

    | StmtOfIterationStmt is -> generate_iteration_stmt is

    | StmtOfJumpStmt js -> generate_jump_stmt js

    | StmtOfDrawStmt ds -> generate_draw_stmt ds

    | StmtList cs -> generate_stmt_list cs
```

```
and generate_dec_stmt (ds: declarator list) =
    match ds with
    | [] -> ""
    | head :: [] -> generate_declarator head ^"\n"
    | head :: tail ->
        let head = generate_declarator head in
        let tail = generate_dec_stmt tail in
        head ^ tail


and generate_declarator (d: declarator) =
    match d with
    | DECL (dt, s,li, e) ->
        let dt = generate_datatype dt in
        let e = generate_expr e in
        dt ^ " " ^ s ^ "=" ^ e ^ ";\n"(*modified by Adarsh*)

and generate_datatype (dt: datatype) =
    match dt with
    | TypeOfInt -> "int"
    | TypeOfDouble -> "double"
    | TypeOfBoolean -> "boolean"
    | TypeOfVoid -> "void"
    | TypeOfString -> "String"
    | TypeOfUndeclare -> ""

and generate_expr_stmt (es: expr_stmt) =
    match es with
    | ExprStmt e -> generate_expr e ^ ";\n"



and generate_expr (e: expr) =
    match e with
    | ExprOfCstExpr (c, li) -> generate_cst_expr c
    | ExprOfAssign (l, e) ->
        let l = generate_lvalue l in
        let e = generate_expr e in
        l ^ " = " ^ e
    | ExprOfBineryOp (e1, op, e2) ->
        let e1 = generate_expr e1 in
        let op = generate_op op in
        let e2 = generate_expr e2 in
        e1 ^ " " ^ op ^ " " ^ e2
    | ExprOfSingleOp (op, e) ->
        let op = generate_op op in
        let e = generate_expr e in
        op ^ e
    | ExprOfFunction (id, lst, li) ->
```

```
        let lst = generate_argument_lst lst in
        id ^ "(" ^ lst ^ "g" ^ ")" (*modified by Pengfei*)
    | ExprWithParen (e) ->
        let e = generate_expr e in
        "(" ^ e ^ ")"

and generate_cst_expr (e: cst_expr) =
    match e with
    | CstExprOfInt i -> string_of_int i
    | CstExprOfDouble d -> string_of_float d
    | CstExprOfTrue -> "true"
    | CstExprOfFalse -> "false"
    | CstExprOfID (eid,li) -> eid
    | CstExprOfSTRING(estr) -> estr


and generate_lvalue (l: lvalue) =
    match l with
    | LvalueOfIdentifier (id, li) -> id

and generate_op (op: operator) =
    match op with
    | PLUS -> "+"
    | MINUS -> "-"
    | TIMES -> "*"
    | DIVIDE -> "/"
    | ASSIGN -> "="
    | EQ -> "=="
    | NEQ -> "!="
    | ST -> "<"
    | LEQ -> "<="
    | GT -> ">"
    | GEQ -> ">="
    | AND -> "&&"
    | OR -> "||"
    | NOT -> "!" (*added NOT -Adarsh*)
    | DOUBLEPLUS -> "++"
    | DOUBLEMINUS -> "--"
    (* Added DOUBLEPLUS DOUBLEMINUS -Yuandi *)

and generate_argument_lst (lst: expr list) =
    match lst with
    [] -> ""
    | head :: [] -> generate_expr head ^ ", "
    | head :: tail ->
        let head = generate_expr head in
        let tail = generate_argument_lst tail in
        head ^ ", " ^ tail (*modified by Pengfei*)
```

```
and generate_selection_stmt (ss: selection_stmt) = (* We don't support
elseif -Yuandi *)
    match ss with
    SelectionStmtIf (e, lst) ->
        let e = generate_expr e in
        let lst = generate_stmt_list lst in
        "if (" ^ e ^ ")\n" ^
        "{\n" ^
        lst ^
        "}\n"

    | SelectionStmtIfElse (e, iflst, elselst) ->
        let e = generate_expr e in
        let iflst = generate_stmt_list iflst in
        let elselst = generate_stmt_list elselst in
        "if (" ^ e ^ ")\n" ^
        "{\n" ^
        iflst ^
        "}\n" ^
        "else\n" ^
        "{\n" ^
        elselst ^
        "}\n"

and generate_iteration_stmt (is: iteration_stmt) =
    match is with
    | IterationStmtOfWhile (e, lst) ->
        let lst = generate_stmt_list lst in
        let e = generate_expr e in
        "while(" ^ e ^ ")\n" ^
        "{\n" ^
        lst ^
        "}\n"
    | IterationStmtOfFor (s_init, e_condition, e_loop, lst) ->
        let s_init = generate_for_init_stmt s_init in
        let e_condition = generate_expr e_condition in
        let e_loop = generate_expr e_loop in
        let lst = generate_stmt_list lst in
        "for (" ^ s_init ^ e_condition ^ "; " ^ e_loop ^ ")\n" ^
(*Modified by Adarsh*)
        "{\n" ^
        lst ^
        "}\n"

and generate_for_init_stmt (fis: for_init_statement) =
    match fis with
    | FORWithExpr fe -> generate_expr_stmt fe
    | FORWithDec fd -> generate_dec_stmt fd

and generate_jump_stmt (js: jump_stmt) =
```

```
    match js with
    JumpStmtOfContinue li-> "continue;\n"
    | JumpStmtOfBreak li-> "break;\n"
    | JumpStmtOfReturnExpr e ->
        let e = generate_expr e in
        "return " ^ e ^ ";\n"
    | JumpStmtOfReturn li-> "return;\n"

and generate_draw_stmt (ds: draw_stmt) = (*Modified by Adarsh*)
    match ds with
    | DrawStmt (act,e) ->
      generate_action act ^ generate_expr e ^ ", g);\n"
    | DrawStmtWithoutExp act ->
        generate_action act ^ "g);\n"

(* Modified by Yuandi *)
(* Deleted CLEAN, added FILLS FILLE PRINT *)
and generate_action (a: action) = (*Added by Adarsh*)
    match a with
    | FD -> "FD(" (* forward *)
    | BK -> "BK(" (* back *)
    | RT -> "RT(" (* turn right *)
    | LT -> "LT(" (* turn left *)
    | PD -> "PD(" (* pen down *)
    | PU -> "PU(" (* pen up *)
    | RESET -> "RESET(" (* reset *)
    | PC -> "PC(" (* pen color *)
    | PW -> "PW(" (* pen width *)
    | FILLS -> "FILLS(" (* fill start *)
    | FILLE -> "FILLE(" (* fill end *)
    | PRINT -> "PRINT(" (* print *)

and generate_stmt_list (sl: stmt list) =
    match sl with
    | [] -> ""
    | head :: [] -> generate_stmt head ^"\n"
    | head :: tail ->
        let head = generate_stmt head in
        let tail = generate_stmt_list tail in
        head ^ tail


(* Modified by Yuandi *)
(* Added MAINFunctionDef *)
and generate_func_def (f: func_def) =
    match f with
    | MAINFunctionDef (dt, plst, cs) ->
        let cs = generate_stmt_list cs in
```

```
        "public void paintComponent(Graphics 
g){\nreInitialize(g);\nsuper.paintComponent(g);\n" ^ cs ^ "\n}\n"    | 
FunctionDef (dt, s, plst, cs) ->
        let dt = generate_datatype dt in
        let plst = generate_parameter_list plst in
        let cs = generate_stmt_list cs in
        "public " ^ dt ^ " " ^ s ^ "(" ^ plst ^ "Graphics g" ^ ") {\n" 
^ cs ^ "\n}\n"(*modified by Pengfei*)

and generate_parameter_list (lst: parameter list) =
    match lst with
    | [] -> ""
    | head :: [] -> generate_parameter head ^ ", "
    | head :: tail ->
        let head = generate_parameter head in
        let tail = generate_parameter_list tail in
        head ^ ", " ^ tail (*modified by Pengfei*)


and generate_parameter (p: parameter) =
    match p with
    | ParameterType (dt, id) ->
        let dt = generate_datatype dt in
        dt ^ " " ^ id

end
```

generator.mli

*Written by Yuandi Jin and Adarsh Banad

```
module Generator:
    sig
    val generate : Ast.program -> string
    end
```

compiler.ml

*Written by Adarsh Banad

```
open Header
open Semantic
open Parser
open Generator


type action = Compile


let generate_code (program: Ast.program) =
    let str = Generator.generate program in
    let lines = ref [] in
    (*read from template file*)
    let file = open_in "NewLogoPanel (copy).java" in
    for i = 1 to 170 do (* changed by Yuandi *)
        lines := input_line file :: !lines
    done;
    let s1 = String.concat "\n" (List.rev !lines) ^ "\n" in
    let s2 =
        try
            lines := [];
            while true do
                lines := input_line file :: !lines
            done;
            ""
        with End_of_file ->
            close_in file;
            String.concat "\n" (List.rev !lines) ^ "\n"
    in
    let file = open_out Sys.argv.(3) in
    (*concate template file and compiled js code*)
    Printf.fprintf file "%s" (s1 ^ str ^ s2);
    close_out file


let _ =
    let action = Compile in

    let srcfile = open_in Sys.argv.(2) in
    let lexbuf = Lexing.from_channel srcfile in
    let program = Parser.program Scanner.token lexbuf in
```

```
    match action with
        | Compile ->
                Semantic.check program;
                match !Semantic.checking_succeed with
                false -> ()
                | true ->
                    generate_code program(*modified by
shuleizhao*)
```

NewLogoFrame.java

*Written by Shulei Zhao

```
import javax.swing.JFrame;

@SuppressWarnings("serial")
public class NewLogoFrame extends JFrame {

	public static NewLogoPanel panel = new NewLogoPanel();

	public NewLogoFrame()
	{
		//properties of the frame
		super("newlogo");// title of the panel
		this.setSize(900, 600);// size of the frame

		this.setLocation(150, 150);// location of the frame on the 
screen
		this.getContentPane().add(panel);// add the panel
		this.setVisible(true);
		this.setDefaultCloseOperation(EXIT_ON_CLOSE);// exit the 
program after the frame is closed
	}

	public static void main(String[] args)
	{
		new NewLogoFrame();
	}
}
```

NewLogoPanel (copy).java

*Written by Shulei Zhao

```
import java.awt.BasicStroke;
import java.awt.Color;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.Point;
import java.util.Vector;

import javax.swing.*;

@SuppressWarnings("serial")
public class NewLogoPanel extends JPanel{

            public boolean isFilling;
            public boolean isPenDown;
            public double directionX;
            public double directionY;
            public Point endPoint;// end point of drawing, i.e. current
location of the pen
            public Point startPoint;// start point of drawing
            public Vector<Integer> xFillPoints;
            public Vector<Integer> yFillPoints;

            public NewLogoPanel(){
                  isPenDown = true;
                  startPoint = new Point(450,300);
                  endPoint = new Point(450,300);
                  directionX = 0.0;
                  directionY = 1.0;//initial direction is to right
                  xFillPoints = new Vector<Integer>();
                  yFillPoints = new Vector<Integer>();
            }

            public void reInitialize(Graphics graphics){
                  isPenDown = true;// pen is up at first
                  startPoint = new Point(450,300);
                  endPoint = new Point(450,300);// pen locates at the
up-left corner of the panel at first
                  directionX = 0.0;
                  directionY = 1.0;// direction is RIGHT at first
                  graphics.setColor(Color.BLACK);// color of the pen is
black at first
                  ((Graphics2D) graphics).setStroke(new
BasicStroke(1));// the width of the pen is 1 at first
                  xFillPoints = new Vector<Integer>();//
```

```
			yFillPoints = new Vector<Integer>();// remove all
filling points
		}

		public void changeDirection(double angleD, boolean
isLeft){//to deal with right and left(input angle is in degrees)

			//check whether this is a "left" operation
			if(!isLeft){
				angleD *= -1;
			}

			double currentAngle = Math.atan2(directionY,
directionX);// get current direction angle in radians, y first and x
second
			double newAngle = currentAngle +
Math.toRadians(angleD);// get new direction angle in radians
			//update direction
			directionX = Math.cos(newAngle);
			directionY = Math.sin(newAngle);
		}

		public void forward(int distance, boolean isBack, Graphics
graphics){//to deal with forward and back

			//check whether this is a "back" operation
			if(isBack){
				distance *= -1;
			}

			//update start and end points
			Point tempPoint = startPoint;
			startPoint.setLocation(endPoint);
			endPoint.setLocation(Math.rint(tempPoint.getX() +
distance * directionX), Math.rint(tempPoint.getY() - distance *
directionY));

			if(isPenDown){// draw the line only if the pen is
down
				graphics.drawLine(startPoint.x, startPoint.y,
endPoint.x, endPoint.y);
			}

			if(isFilling){
				xFillPoints.add(endPoint.x);
				yFillPoints.add(endPoint.y);
			}
		}

		public void beginFill(){
```

```
                xFillPoints.removeAllElements();
                yFillPoints.removeAllElements();

                isFilling = true;
                xFillPoints.add(endPoint.x);
                yFillPoints.add(endPoint.y);
          }

          public void endFill(Graphics graphics){
                int[] xArray = new int[xFillPoints.size()];
                int[] yArray = new int[xFillPoints.size()];

                for(int i = 0; i < xFillPoints.size(); i++){
                      xArray[i] = xFillPoints.elementAt(i);
                      yArray[i] = yFillPoints.elementAt(i);
                }

                if(xFillPoints.size() > 2 && isPenDown){
                      graphics.fillPolygon(xArray, yArray,
xFillPoints.size());

                      xFillPoints.removeAllElements();
                      yFillPoints.removeAllElements();
                }
                isFilling = false;
          }

          public void penUp(){
                isPenDown = false;
          }

          public void penDown(){
                isPenDown = true;
          }

          public void print(String s, Graphics graphics){

                // print the string at the current location of the
pen
                if(isPenDown){
                      graphics.drawString(s, endPoint.x, endPoint.y);
                }
          }

          public void setPenColor(int color, Graphics graphics){
                Color c = Color.BLACK;//default color is black

                switch(color){
                      case 0:
```

```
                        c = Color.WHITE;
                        break;
                  case 1:
                        c = Color.RED;
                        break;
                  case 2:
                        c = Color.ORANGE;
                        break;
                  case 3:
                        c = Color.YELLOW;
                        break;
                  case 4:
                        c = Color.GREEN;
                        break;
                  case 5:
                        c = Color.CYAN;
                        break;
                  case 6:
                        c = Color.BLUE;
                        break;
                  case 7:
                        c = Color.MAGENTA;
                        break;
                  default:
                        break;
            }

            graphics.setColor(c);
      }

      public void setPenWidth(int width, Graphics graphics){
            ((Graphics2D) graphics).setStroke(new
BasicStroke(width));
      }
      //-----------------------------------
      // translated code will be added here










      //-----------------------------------
      public void FD(int distance, Graphics graphics){
            forward(distance, false, graphics);
      }
```

```
	public void BK(int distance, Graphics graphics){
		forward(distance, true, graphics);
	}

	public void RT(double angleD, Graphics graphics){
		changeDirection(angleD, false);
	}

	public void LT(double angleD, Graphics graphics){
		changeDirection(angleD, true);
	}

	public void PD(Graphics graphics){
		penDown();
	}

	public void PU(Graphics graphics){
		penUp();
	}

	public void PC(int color, Graphics graphics){
		setPenColor(color, graphics);
	}

	public void PW(int width, Graphics graphics){
		setPenWidth(width,graphics);
	}

	public void FILLS(Graphics graphics){
		beginFill();
	}

	public void FILLE(Graphics graphics){
		endFill(graphics);
	}

	public void PRINT(String s, Graphics graphics){
		print(s, graphics);
	}

}
```