COLOGO: A Graph Language
Final Report

Lixing Dong	ld2505@columbia.edu
Chao Song	cs2994@columbia.edu
Dongyang Jiang	dj2322@columbia.edu
Siyuan Lu	sl3352@columbia.edu
Zhou Ma	zm2167@columbia.edu

Advisor:
Stephen Edwards

December 22, 2011

Contents

1 Introduction

1.1 Description

Our COLOGO language is an effective programming language for drawing 2D graphics. The COLOGO language is designed in spirit of low threshold, which enables easy entry by novices and yet meet the needs of high-powered users. We can use COLOGO for education as it contains basic computer concepts appropriate for beginners. We can also draw interesting pictures and design complicated logos with COLOGO so that the language could be widely used for entertainment or commercial area.

1.2 Features

Euclidean COLOGO operates in a Euclidean space using relative measures and angles, without an origin, unlike coordinate-addressed systems such as Cartisian geometry.

Functional In our COLOGO language, users can create their own functions to perform a specific task. This help programmers to decompose the complex program to simple steps. Also, this feature allow users to reuse the code across different programs.

Recursive is supported in our COLOGO language. This allow users to simplify their code by dividing a problem to subproblems of the same type.

Iridescent COLOGO support drawing lines of different colors and line width, making your drawing experience more colorful.

1.3 Objectives

The main goal of our programming language is to provide a easy way to draw 2D graphics. These graphics, and hence our language, can be used for representing mathematical formulas, teaching geometric concepts, simple arithmetical operation and simulation of robots routing. Also, COLOGO is a appropriate language for teaching basic programming language concepts. Basic data types will be supported in our COLOGO, such as integers, and bool. Some simple data structures like array will also be implemented in it. By providing these, it's relatively easy to manipulate drawing and make it much easy to draw beautiful mathematical graphics.

2 Language Tutorial

COLOGO is a language using simple code to draw beautiful images.

2.1 A Simple Example

Here is a simple example program of COLOGO.

```
FD 5;
```

This will draw a line of length 5.

2.2 Complete Tutorial

The basic idea of COLOGO is the same as the LOGO. Imagine you have a pen located at the center of the canvas at the beginning. Then the purpose of program is to move the pen so that lines can be drawn.

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

2.2.1 Variable Declaration

Variable declaration contains two parts, a type and a variable name. A type could be an integer, boolean, or user defined objects. Here are some examples of variable declaration:

```
int a,b;
bool c;
foo d; (:foo is a object:)
int e[10]; (:array:)
```

2.2.2 Expression

COLOGO support various kinds of expressions. A complete list of unit expressions we support are listed as Table 1:

We also support compound expressions. Programmers can combiner any number of unit expressions above to generate more powerful expressions. Such as `a=b+c; f(g(a))`, etc.

2.2.3 Loop

The keyword for looping in COLOGO is Loop We support two kinds of loops. If a number is specified for a loop, say `Loop(n)`, the statement inside the loop will run for n times. If `Loop()` is used without a loop number it is considered as a infinite loop. You will have to use break to jump out of the loop.

When inside the loop, if you want to jump out of the loop, you can use Break. Also, if you want to skip the rest of the loop and continue the next iteration, use Goon.

Here are some example of how to use loops:

Name	Syntax
Assignment	`a = b`
Plus	`a + b`
Minus	`a - b`
Multiply	`a * a`
Divide	`a / a`
Not equal	`a != b`
Equal	`a == b`
Less than	`a < b`
Great than	`a > b`
less than or equal to	`a <= b`
greater than or equal to	`a >= b`
negative	`-a`
not	`Not a`
function call	`f(a)`

Table 1: Expressions in COLOGO

```
Loop(10) { a = a + 1; }
Loop() { a = a + 1; Break; }
Loop(10) { a = a + 1; Goon;}
```

2.2.4 Condition

Our condition statement use keyword If. The syntax is If (condition) then statement block End, or If (condition) then statement block else statement block End.
Here are some examples of how to use If statements:

```
If (c) { a = b; } End
If (c) { a = b; } Else { b = a; } End
```

2.2.5 Drawing

Drawing is the unique feature that COLOGO differs from other programming languages. We support various number of drawing statements.

The whole supporting drawing statements are listed in table 2.

2.2.6 Functions

The syntax of our function definition is:

```
Func functionname(parameter list) : return type
{
   function body
}
```

Statement	Description
`FD (int)`	Move forward (int) pixels
`BK (int)`	Move back (int) pixels
`LF (int)`	Turn left (int) degree
`RT (int)`	Turn right (int) degree
`RESET`	Reset pen position to center of canvas
`CLS`	Clear screen
`PU`	Pen up (stop drawing)
`PD`	Pen down (start drawing)
`PF`	Pen flip (switch drawing state)
`WD (int)`	Set line width to (int) pixels
`RGB (int), (int), (int)`	Set the color of drawing

Table 2: Drawing Statements in COLOGO

A example of function definition is:

```
Func Fibonacci(int n) : int
{
    If ( n < 2 )
        Return 1;
    Else
        Return Fibonacci(n-1) + Fibonacci(n-2);
    End
}
```

2.2.7 Object

The syntax of our object definition is as follows:

```
Obj objectname {element list}
```

Note that our object do not support functions.

To declare an instance of a obj, use the following syntax:

```
objectname variablename;
```

To get a reference of a element of a object, use '`.`'. For example:

```
Obj foo
{
   int bar;
   int baz;
}
foo a;
a.baz = 2;
```

2.2.8 Comment

We use `"(:"` and `":)"` to indicate the start and end of comments. For example:

```
(:this is a comment:)
```

2.3 Compile COLOGO programs

To compile COLOGO language files, you must first compile the compiler. Use makefile provided in the package to install the compiler. Then, use command

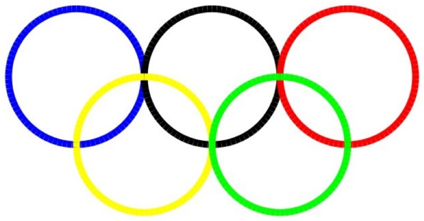
```
./compiler -c inputFileName outputFileName
```

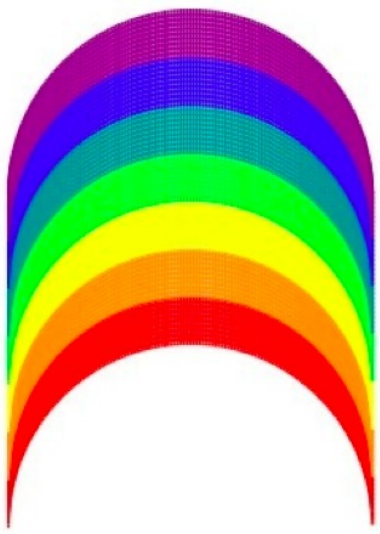
to compile source code. To print AST tree, use -a instead:

```
./compiler -a inputFileName > outputFileName
```

2.4 More colorful examples

Using our COLOGO language, programmers could bable to generate all kinds of colorful images. Here are some more colorful examples:





3 Language Manual

3.1 Introduction

COLOGO language is an effective programming language for drawing 2D graphics. The COLOGO language is designed in spirit of low threshold, which enables easy entry by novices and yet meet the needs of high-powered users. We can use COLOGO for education as it contains basic computer concepts appropriate for beginners. We can also draw interesting pictures and design complicated logos with COLOGO so that the language could be widely used for entertainment or commercial area.

3.2 Lexical Conventions

The first step to compile our language is lexical analysis. In this step, the imported files are added in, and the program is recognized as a sequence of tokens.

3.2.1 Character Set

COLOGO supports ASCII character set.

3.2.2 Identifier

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 also 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 marco names and all other names without external linkage. Identifiers with external linkage are more restricted

3.2.3 Comments

Comments are introduced by (: and ended by :). Comments are not allowed to be nested. When a comment starts with a (: , the comment will be ended by the next occurrence of :).

3.2.4 Keyword

FD	BK	LF	RT	RESET
CLS	PU	PD	PF	RGB
int	bool	Obj	Not	If
Else	Goon	Break	Return	End
Func	True	False	Loop	

In general, keywords are separated into four categories:

Drawing functional
Logical operator
Variable type indication
Certain Statements

3.2.5 Operators

COLOGO has 6 categories of operators. They are unary, additive, multiplicative, relational, logical and object reference operator, respectively:

Unary

`- !`

Additive

`+ -`

Multiplicative

`* /`

Relational

`== != < <= > >=`

Logical

`Not`

Object reference operator

`.`

3.2.6 Separators

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

3.2.7 Syntax group

`{}`: Braces are delimiter of compound statement, used in the cases of statements block and constant array initialization.
`[]`: Brackets are used for array index dereference.
`()`: Parenthesis are for expression grouping and argument express

3.3 Lvalue

Lvalue is an expression that refers to a region of storage. It is required by certain operators. Refer to the operator part to see which operators expect an lvalue.

3.4 Declarations

Declarations create variables with several attributes: variable name, type, variable value(optional).

3.4.1 Type Specifier

Primitive Types
There are two primitive types in COLOGO. They are declared as below:

```
int id = value;
bool id = value;
```

Where id is the name of variable and value is an expression or a primitive value.

Array Type
For each primitive type, COLOGO has a corresponding array container. They are:

```
int id[length];
bool id[length];
```

Where id is the name of variable and length is the number of elements contained in the array. The above form will initialize the array as zero for int, and false for bool type.

Object Type
COLOGO allows the user to integrate multiple primitive type and form a complex object type such that all the primitive type variables can be passed and referred to together. The declaration are as follows:

```
Obj id
{
primitive-declaration-list
}
```

Where id stands for the name of variable and the primitive-declaration-list stands for a list of primitive declaration in the form of primitive- declaration-1; primitive-declaration-2; etc.

3.5 Statements

In COLOGO, statements are executed in sequence. They fall into several groups.

```
Statement:
  expression-statement
  compound-statement
  selection-statement
  iteration-statement
  jump-statement
```

3.5.1 Expression Statement

Most statements in COLOGO are expression statements, which have the form expression-statement:

```
expression-opt;
```

Most expression statements are assignments or function calls. All side effects from the expression are completed before the next statement is executed. If the expression is missing, the construction is called a null statement; it is often used to supply an empty body to an iteration statement to place a label.

3.5.2 Compound Statement

So that several statements can be used where one is expected, the compound statement (also called “block”) is provided. The body of a function definition is a compound statement.

```
compound-statement:
  { declaration-list-opt statement-list-opt }

declaration-list:
  declaration-list
  declaration

statement-list:
  statement-list
  statement
```

If an identifier in the declaration-list was in scope outside the block, the outer declaration is suspended within the block, after which it resumes its force. An identifier may be declared only once in the same block. These rules apply to identifiers in the same name space; identifiers in different name spaces are treated as distinct. Initialization of automatic objects is performed each time the block is entered at the top and proceeds in the order of the declarators. If a jump into the block is executed, these initializations are not performed. Initialization of static objects is performed only once, before the program begins execution.

3.5.3 Selection-Statement

Selection statements choose one of several flows of control.

```
selection-statement:
If (expression) statement End
If (expression) statement Else statement End
```

In both forms of the If statement, the expression, which must have arithmetic or pointer type, is evaluated, including all side effects, and if it compares unequal to false, the first substatement is executed. In the second form, the second substatement is executed if the expression is false. The else ambiguity is resolved by introducing keyword End

3.5.4 Iteration-Statement

Iteration statements specify looping.
iteration-statement:

```
Loop (expression(opt)) statement
```

In the LOOP statement, the parameter expression must have BOOL type; it is evaluated before each iteration, and if it becomes equal to 0, the LOOP is terminated. Side-effects from each expression are completed immediately after its evaluation.

3.5.5 Jump-Statement

A GOON statement may appear only within an iteration statement. It causes control to pass to the loop-continuation portion of the smallest enclosing such statement. More precisely, within each of the statements

```
Loop (...) { ...; Goon; }
```

A Break statement may appear only in an iteration statement or, and terminates execution of the smallest enclosing such statement; control passes to the statement following the terminated statement.
A function returns to its caller by the Return statement. When Return is followed by an expression, the value is returned to the caller of the function. The expression is converted, as by assignment, to the type returned by the function in which it appears.
Running to the end of a function is equivalent to a return with no expression. In either case, the returned value is undefined.

3.5.6 Draw Statement

Draw statements include operations for the turtle, which provide the drawing functionality. We have 9 kinds of drawing statements.

```
FD expression; : Turtle move forward expression distance.
BK expression; : Turtle move backward expression distance.
LF expression; : Turtle turn left expression degree.
RT expression; : Turtle turn right expression degree.
RESET; :Reset the turtle to original position.
CLS; : Clear the screen.
PF; : Pen flip.
PD; : Pen down.
PU; : Pen up
```

3.6 Scope

A program need not all be compiled at one time: the source text may be kept in several files containing translation units. Communication among the functions of a program may be carried out both through calls and through manipulation of external data. In our language the only one scope to consider is the lexical scope of an identifier which is the region of the program text within which the identifier's characteristics are understood; Identifiers fall into several name spaces that do not interfere with one another; the same identifier may be used for different purposes, even in the same scope, if the uses are in different name spaces. The scope of a parameter of a function definition begins at the start of the block defining the function and persists through the function; the scope of a parameter in a function declaration ends at the end of the declarator. The scope of an identifier declared at the head of a block begins at the end of its declarator, and persists to the end of the block. The scope of a structure, union, or enumeration tag, or an enumeration constant, begins at its appearance in a type specifier, and persists to the end of a translation unit (for declarations at the external level) or to the end of the block (for declarations within a function). If an identifier is explicitly declared at the head of a block, including the block constituting a function, any declaration of the identifier outside the block is suspended until the end of the block.

3.7 Grammer

```
program :
        | free_sentence program

free_sentence : function_definition
              | obj_definition
              | statement

function_definition : FUNC function_declarator COLON cltype LBRACE in_func_sentenc
                    | FUNC function_declarator LBRACE in_func_sentence_list RBRACE

function_declarator : ID LPAREN parameter_list RPAREN
                    | ID LPAREN RPAREN

parameter_list : parameter
               | parameter COMMA parameter_list

parameter : cltype ID

in_func_sentence_list : in_func_sentence
                      | in_func_sentence in_func_sentence_list

in_func_sentence : statement
```

```
16  obj_definition : OBJ ID LBRACE in_obj_sentence_list RBRACE

17  in_obj_sentence_list : in_obj_sentence
18                       | in_obj_sentence in_obj_sentence_list

19  in_obj_sentence : variable_declaration

20  variable_declaration : cltype declarator_list SEMI

21  declarator_list : declarator
22                  | declarator COMMA declarator_list

23  cltype : INT
24         | REAL
25         | ID
26         | BOOL

27  declarator : ID
28             | ID LBRACK RBRACK
29             | ID LBRACK constant_expression RBRACK

30  statement : variable_declaration
31            | expression_statement
32            | draw_statement
33            | iteration_statement
34            | selection_statement
35            | jump_statement

36  expression_statement : expression SEMI
37                       | expression error

38  draw_statement : FD expression SEMI
39                 | BK expression SEMI
40                 | LF expression SEMI
41                 | RT expression SEMI
42                 | RESET SEMI
43                 | CLS SEMI
44                 | PF SEMI
45                 | PD SEMI
46                 | PU SEMI
47                 | WD expression SEMI
48                 | RGB expression COMMA expression COMMA expression SEMI
```

```
49                 | RGB error SEMI

50  selection_statement : IF LPAREN expression RPAREN statement_list END
51                      | IF LPAREN expression RPAREN statement_list ELSE statement_lis
52                      | IF LPAREN expression RPAREN error
53                      | IF LPAREN expression RPAREN statement_list ELSE error

54  statement_list : statement
55                 | statement statement_list

56  iteration_statement : LOOP LPAREN expression RPAREN LBRACE statement_list RBRACE
57                      | LOOP LPAREN RPAREN LBRACE statement_list RBRACE
58                      | LOOP LPAREN RPAREN LBRACE error

59  jump_statement : GOON SEMI
60                 | BREAK SEMI
61                 | RETURN expression SEMI
62                 | RETURN SEMI

63  expression : constant_expression
64             | LPAREN expression RPAREN
65             | LPAREN error
66             | primary
67             | lvalue ASSIGN expression
68             | expression PLUS expression
69             | expression MINUS expression
70             | expression TIMES expression
71             | expression DIVIDE expression
72             | expression NEQ expression
73             | expression EQ expression
74             | expression LT expression
75             | expression GT expression
76             | expression LEQ expression
77             | expression GEQ expression
78             | MINUS expression
79             | NOT expression
80             | ID LPAREN argument_expression_list RPAREN

81  identifier : ID
82             | ID DOT identifier

83  primary : identifier
84          | identifier LBRACK expression RBRACK
```

```
85          | identifier LBRACK error

86  lvalue : identifier
87         | identifier LBRACK expression RBRACK
88         | identifier LBRACK error

89  argument_expression_list : expression
90                           | expression COMMA argument_expression_list

91  constant_expression : INTV
92                      | REALV
93                      | TRUE
94                      | FALSE
```

4 Project Plan

4.1 Team Responsibilities

Every team member will be given primary responsibility for certain project goals, clear and definite roles will make sure that every team member pull his own weight. The fundamental tasks of each team member are shown in Table 3.

Lixing Dong	Scanner, parser, and code generator
Chao Song	Semantic analysis
Dongyang Jiang	Test and code coverage
Siyuan Lu	AST printer
Zhou Ma	Error Recovery

Table 3: Project Responsibilities

4.2 Project Timeline

The deadlines were set for key project development goals as Table 4 shows

09-12-2011	Language whitepaper, core language features defined
09-28-2011	Language proposal, core language features defined
10-31-2011	Language reference manul, grammar complete
11-07-2011	Scanner, parser complete
11-14-2011	Code generation complete
11-31-2011	Semantic Analysis complete
12-10-2011	Error recovery complete
12-17-2011	Code freeze, project feature complete

Table 4: Project Timeline

4.3 Software Development Environment

This project is developed on Mac using Ocaml 3.11.0. The parser will be developed using ocalmlyacc and the scanner is developed using ocamllex. The whole project is tested using bisect to test the code coverage. Source code is controlled using a distributed revision control system, Git. Makefiles is created in source directory. No files will be checked in to Git unless it makes without error. The proposal, reference manual, and the final report will be written in Google docs by all the members. Final report is generated by latex.

4.4 Project Log

The actual lists of significant project milestones are shown as Table 5.

09-12-2011	Project initiated
09-28-2011	Language white paper complete
10-07-2011	Code conventions, first draft
10-10-2011	Development environment configured
10-20-2011	Grammar first draft
10-28-2011	Language reference manual
11-10-2011	Scanner, parser complete
11-21-2011	Code generator finished
11-28-2011	Semantic analysis finished
12-10-2011	Error Recovery finished
12-22-2011	Final report complete

Table 5: Project Log

4.5 Programming Style

The programming style guide is modified based on here:
http://www.seas.upenn.edu/ cis341/programming_style.html

File Submission Requirements:
Code must compile
80 column limit

Commenting:
Comments go above the code they reference
Avoid useless comments
Avoid over-commenting
Line breaks
Proper multi-line commenting

Naming and Declarations:
Use meaningful names
Naming conventions

Indentation:
Indenting nested let expressions
Indenting match expressions
Indenting if expressions
Indenting comments

Using Parentheses:
Parenthesize to help indentation
Wrap match expressions with parenthesis

Over parenthesizing

Pattern Matching:
No incomplete pattern matches
Pattern match in the function arguments when possible
Function arguments should not use values for patterns
Avoid using too many projections
Pattern match with as few match expressions as necessary

Code Factoring:
Don't let expressions take up multiple lines
Breakup large functions into smaller functions
Over-factoring code

Verbosity:
Don't rewrite existing code
Misusing if expressions
Misusing match expressions
Other common misuses
Avoid computing values twice

5 Architectural Design

5.1 Architecture

The Cologo compiler consists of several major blocks which are common in compiler designs: abstract syntax tree, scanner, parser, printer, semantics, code generator. The relationship between these components is demonstrated in Figure 1. The input to the compiler are Pencil specification files (which have, by convention, the suffix .cl) and the final output from the compiler is translated HTML code. The compiler takes one cologo file once a time, and translates it to HTML. The scanner is implemented in OCamllex, a version of lex for Ocaml. The parser is generated by ocamlyacc, a version of yacc for Ocaml.

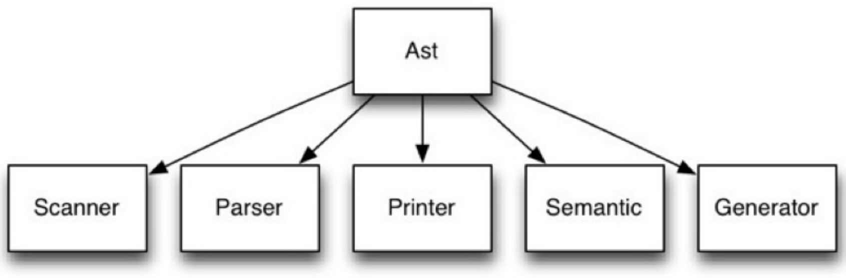


Figure 1

5.2 Work Flow

When compiling a COLOGO program, scanner will first read the code and turn them into tokens. Then parser will parse the token and convert them to AST. Then semantic analysis is applied to AST to check whether the code is semantically correct. Finally, code generator is used to generate target code from AST. Or, if '-a' is used, then instead of generating html code, the structure of AST will be printed. Figure 2 shows how our compiler works.

5.3 Work Distribution

The actual work distribution of our project is similar to the states in chapter 4, table 6 shows the details of it.

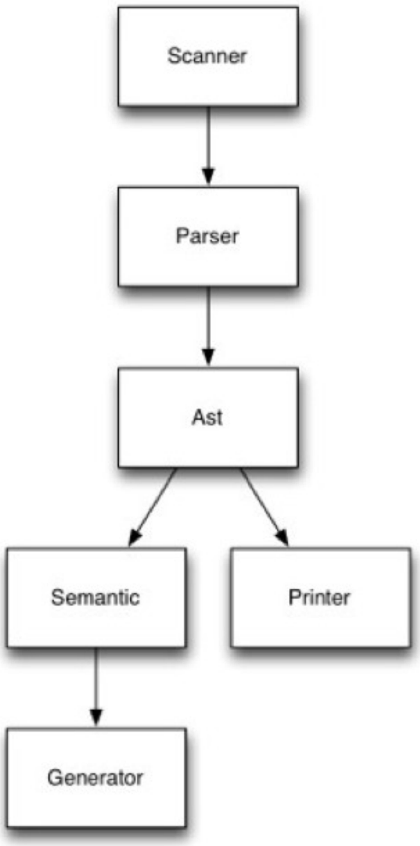


Figure 2

Scanner.mll	Implemented by Lixing Dong
Parser.mly	Originally implemented by Lixing Dong Zhou Ma add some work to support naive code recovery
semantic.ml	Implemented by Chao Song
ast.ml	Discussed by all team member, modified as project progresses
astprinter.ml	Implemented by Siyuan Lu
generator.ml	Implemented by Lixing Dong
Test cases	Contributed by Lixing Dong, Chao Song and Dongyang Jiang
Code coverage	Implemented by Dongyang Jiang

Table 6: Distribution

6 Test Plan

We designed a variety of test cases to test our compiler. These test cases are located in /test/ folder. And we have a shell script to compile them together.

6.1 A Simple Example of Test Programs

First let's see a simple example of our test program.

This is a very simple program to draw a line of length 5.

```
int a;
FD 5;
```

The following is its corresponding HTML code. Between the start point and the end point are the javascript translation code of our COLOGO language. Other lines could be considered as headers and tails.

```
<!DOCTYPE HTML>
<html>
    <head>
        <style>
            body {
                margin: 0px;
                padding: 0px;
            }

            #myCanvas {
                position: absolute;
            }
        </style>
<script src="http://code.jquery.com/jquery-latest.js"></script>
    </head>
    <body onmousedown="return false;">
        <canvas id="myCanvas">
        </canvas>
        <script>

        var _canvas = document.getElementById("myCanvas");
        var _context = _canvas.getContext("2d");
        var _stack = new Array;
        ResizeCanvas();
        _context.lineWidth = 2;

        var _currx = document.width / 2;
        var _curry = document.height / 2;
        var _tmp, _tmpx, _tmpy, _tmps, _i;

        var _r;
        var _g;
        var _b;
        var _bDraw = true;

        // in degree
        var _direction = 0;

        function drawLine(x1, x2, y1, y2){
            _context.beginPath();
            _context.moveTo(x1, x2);
            _context.lineTo(y1, y2);
            _context.stroke();
        }

        //start point

var a;
_tmp = _direction * Math.PI / 180;
_tmpx = Math.cos(_tmp) * ( 5 ) + _currx;
_tmpy = Math.sin(_tmp) * ( 5 ) + _curry;
if (_bDraw)
  drawLine(_currx, _curry, _tmpx, _tmpy);
_currx = _tmpx; _curry = _tmpy;

        //end point


        function ResizeCanvas()
        {
            _canvas.width = document.width;
            _canvas.height = document.height;
        }


        </script>
    </body>
</html>
```

6.2 Test Suite

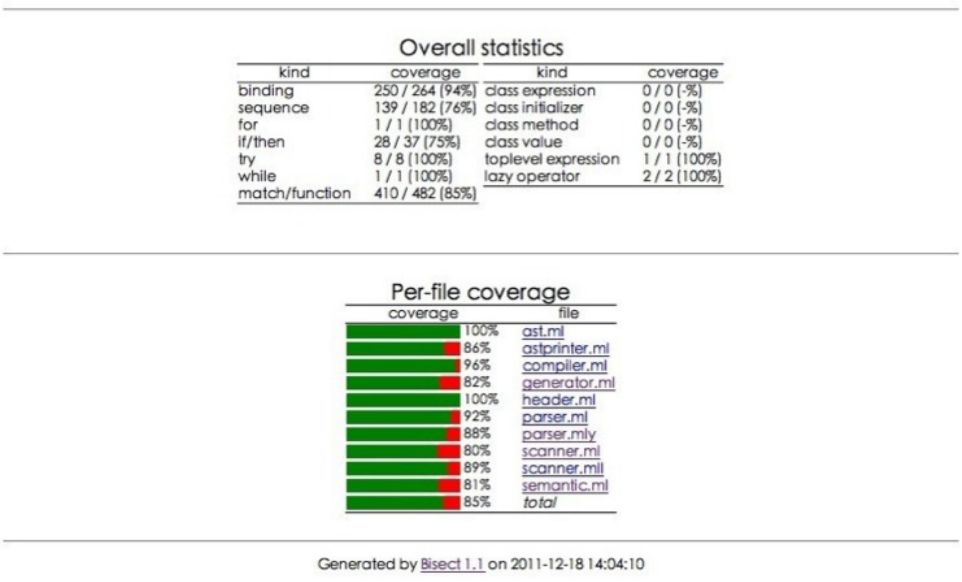
The entire test suite includes 44 test cases.

The way we create those test cases is that we would try to cover all the syntax and semantics we have implemented. For example, at the beginning, we created simple test cases as mentioned above with only variable declaration and FD. As we further developed our system, we design more complicated cases to check whether the new functionality we added works fine. A example would be a test cases using recursive function to draw lines with length corresponding to Fibonacci's number. Among our test cases, some are designed with correct syntax and semantics, thus are expected to work. The others are designed with wrong syntax or semantics, these are expected to fail. This method is usually used to test whether semantic analysis is working fine.

The compiler is auto-tested by test.sh file, it tests all the .cl files in the test folder and generate corresponding HTML file with javascript. Along with auto testing, we do code coverage with a professional code coverage tool – bisect. The code coverage is tested in a professional tool, bisect.

After build the source code , input the command line

```
bisect-report html report bisect*.out
```



Overall statistics

kind	coverage	kind	coverage
binding	250 / 264 (94%)	class expression	0 / 0 (-%)
sequence	139 / 182 (76%)	class initializer	0 / 0 (-%)
for	1 / 1 (100%)	class method	0 / 0 (-%)
if/then	28 / 37 (75%)	class value	0 / 0 (-%)
try	8 / 8 (100%)	toplevel expression	1 / 1 (100%)
while	1 / 1 (100%)	lazy operator	2 / 2 (100%)
match/function	410 / 482 (85%)		

Per-file coverage

coverage	file
100%	ast.ml
86%	astprinter.ml
96%	compiler.ml
82%	generator.ml
100%	header.ml
92%	parser.ml
88%	parser.mly
80%	scanner.ml
89%	scanner.mll
81%	semantic.ml
85%	*total*

Generated by Bisect 1.1 on 2011-12-18 14:04:10

Figure 3

it will display the html form report. As showed in Figure 3. With the details of the report, we can find out the code coverage of every part so that to add new test cases or delete the code would never be used to improve the code coverage and make our code more effective.

7 Lessons Learned

Lixing Dong Team work is very very important. We should regularly meet every week. I learned that a small, energetic team is much better than large team with someone who don't involve into the project much. Also, I found that a beautiful design at the beginning of the project is very helpful. It will reduce much coding time and will be easy for team module-orientated development.

Chao Song I think the greatest lesson I learned through this project is how to program using Ocaml. Before meeting Ocaml, I thought every programming look very similar to each other, but I was wrong. Ocaml uses a completely different idea of how to program. No loop, strong typed. I even had problem using 'if'. This is the only language that alway give me 'syntax error' when compiling. But after I have done the project, I think now I become much more familiar with programming using Ocaml.

Dongyang Jiang Through the project, I have a better understand on how a compiler works and the the structure of compilers and I learned how to write a compiler. The use of the combination of Lex and Yacc to generate scanner and parser makes writing a compiler easier. Besides, I pick up how to use bisect, a test tool, to test the code coverage, so that we can optimize our code. And of course, get familiar with programming with Ocaml.