Doodle Language

Final Report

COMS W4115: Programming Language and Translator
Professor Stephen A. Edward

Yusr Yamani

Fall 2009

Table of Content:

Chapter 1

Introduction

“Doodle” is a programming Language, created using Ocaml. It is designed to help software developers create unfocused sketches in a few simple steps. Due to its simple syntax, “Doodle” is suitable for beginners. Programmers who are familiar with other languages such as C will find Doodle easy to understand.

A full Doodle consists of three primary sections: Declaration, Window, and Object. The Declaration section is where variables and function are declared, defined and introduced. The Window section sets the size and color of the output window. The Object section is where shape object functions, and user defined functions are called. In addition, Doodle supports simple shapes, including ellipses, rectangles, line and text. Flow controls such as iteration statements, conditionals statements are added to Doodle’s language to give it more flexibility. Moreover, this language allows users to define their own functions to avoid code redundancy. Further descriptions of the language are introduced throughout the report.

Chapter 2

Language Tutorial

Two simple examples are being introduced in this chapter to illustrate the overall features of Doodle language:

2.1 Example 1

```
Declare  [ Int  x = 50 ;  ]
Window [  ( 200 , 200 )  ; Red ]
Object [  Rectangle ( x ,  x , 60 , 20 ) ; ]
```

Doodle Code 2.1

2.1.2 Output window:

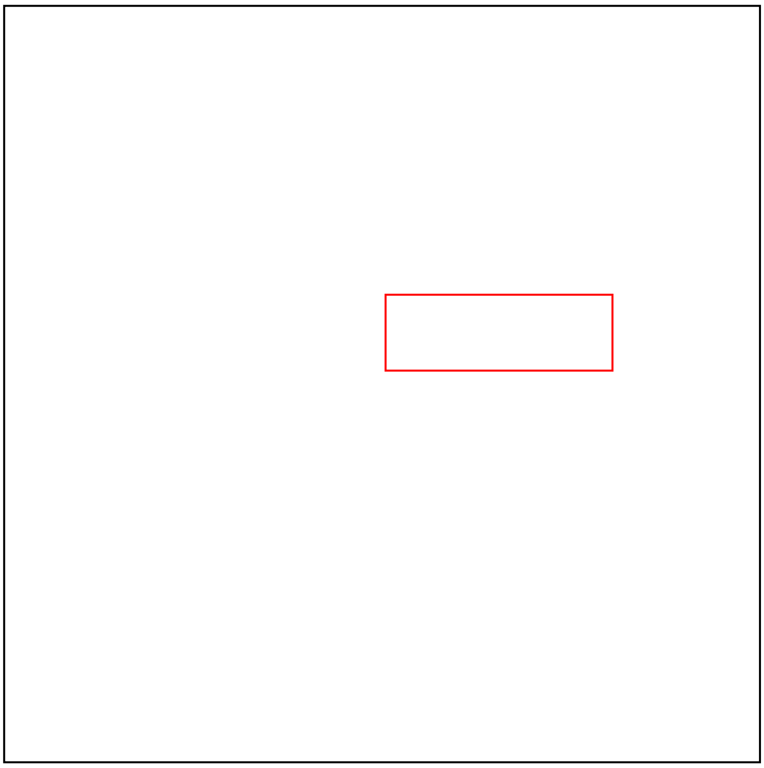


Figure 2.1

Figure 2.1 illustrates the output window of executing code 2.1 . The first line in this code defines an integer variable x, and set it to 50 . The second line sets the output window dimensions to 200pixels X 200pixels. The third line, draws a rectangle of 30w X 20h, with the left bottom corner of it in point (50, 50) on the output window.

The syntax of Doodle program consists of three sequentially executed sections:

<u>Declare Section</u>: It is used to define variables and user functions, which are declared between brackets. Variables should be set to an initial value. Though declare section is optional, it should be included at the beginning of the program file

<u>Window Section:</u> Three parameters state the window specification. The first two parameters are numbers; they set the size of the output window in pixels. The third sets the shape color.

Ex: *Window[(200, 200) ; Red]* sets the output window domains to 200 pixelsX 200pixels, and shapes colors to red.

<u>Object Section:</u> This section is the main section of the program, it includes all the running code. Code varies form simple shape call , user function call , to flow control code. All statements are included within object section brackets. Calling a function or a variable is permitted, though they cannot be declared in this section.

In this example a rectangle is displayed on the output window, using the following statement:

Rectangle(30, 20, 50, 50) It has the form Rectangle (x, y, w, h) which means draw a rectangle with width w, height h, and the lower corner at point (x, y) on the output window.

2.2 Example 2

```
Declare [ Int i=2;
        Func draw2shapes
                If ( i == 1)
                   Ellipse(10, 10, 60, 20) ;
                Else
                    Rectangle(10, 40, 60, 20);
                Endif
         Endfunc]

Window[ (200, 100) ; Red]

Object[ Loop (2)
         Callf draw2shapes ;
         i= i-1;
        Endloop
       ]
```

Doodle Code 2.2

The Declare part defines an integer variable i, and sets it to 2. It also defines a user function *draw2shapse*

```
Func draw2shapes
          If ( i == 1)
             Ellipse(50, 25, 100, 200) ;
          Else
              Rectangle(25, 0, 100, 200);
          Endif
      Endfunc]
```

Inside the body there is an if statement , that draws an Ellipse if i is equal to 1, and draws a rectangle otherwise. The Window section sets the output window as discussed in example 1. The loop in object section executes function draw2shapes twice. Ellipse (x, y, rx, ry) draws an ellipse with horizontal radius rx, vertical radius ry and center at point(x, y)

2.2.1 Output window

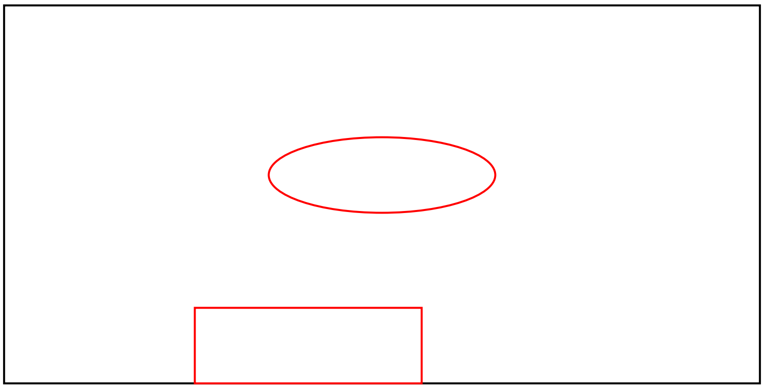


Figure 2.2

Chapter 3

Language Manual

3.1 Syntax Notation

Regular expressions are used for the syntax notation in this manual. No terminals categories are indicated by *italic* style. Quoted or bold style symbols are all terminals. Alternative categories are separated by '|'. An optional category ends with '?' . 'a*' indicates that 'a' may occur zero or more times. 'a+' indicates that 'a' may occur one or more times. '(a|b)' denotes a choice between the categories 'a' and 'b'. Parentheses are used to group symbols with respect to '?', '*', and '+'.

3.2 Lexical Conversions

3.2.1 Comments

Comments enhance programs readability. In Doodle, comments begin with '<*' and ends with '*>'. Any statement in between those symbols is ignored by the compiler.

3.2.2 Whitespace

Whitespaces, including ASCII space, carriage return and horizontal tab separate tokens. Any sequence of whitespaces is ignored by the compiler except spaces within a string.

3.2.3 Tokens

There are six classes of tokens: identifiers, keywords, integer constants, string literals, operators, and separators.

Tokens are case sensitive. Upper and lower case of a letter are different .

3.2.3.1 Identifiers

An identifier is any sequence of letters and digits that begins with a letter. Doodle is case sensitive; two identifiers are the same if they have the same Unicode character for every letter and digit.

$$Identifier \rightarrow Letter(Letter \mid Digit)*$$

$$Letter \rightarrow ['a'-'z' \quad 'A'-'Z']$$

$$Digit \rightarrow ['0'-'9']$$

3.2.3.2 Keywords

The following terms are keywords of the language that may not be used otherwise

Declare
Window
Object
Int
Rectangle
Ellipse
Line
Text
Red
Blue
White
Black
Loop
Endloop
Func
Endfunc
If
Else
Endif
String

2.3.3 Integer Constants

An Integer constant is a sequence of ASCII digits that represents a decimal number. Integers are positive.

$$IntegerConstant \rightarrow digit +$$

3.2.3.4 String Literal

A String literal is a sequence of one or more character, letter or digit, enclosed in double quotes. White spaces are allowed within the string. However, newline, double quote or any other character is not allowed.

$$StringLiteral \rightarrow \ \text{' " '}\ (Letter \mid digit \mid \text{' '})+\ \text{' " '}$$

3.2.3.5 Operators

Operators are: plus '+' minus '-' times '*', divide '/', Assignment '=', Equal "=="

3.2.3.6 Separators

The following symbols are separators in Doodle:

[ ] ; ( ) ,

3.3 The structure of Doodle:

A Doodle program consists of three main parts: Declaration, WindowSpecification, and ObjectSection

3.3.1 Declaration:

This section is optional. It contains identifiers and functions being declared, which should only be declared in this section of the program.

$$Declaration \rightarrow \ \boldsymbol{Declare}\ '['DeclSpecification * ']'$$

3.3.1.1 DeclSpecification:

$$DeclSpecification \rightarrow IdentifierDec \mid FunctionDec$$

3.3.1.2 IdentifierDec:

There are 2 types of variables: Integer and String. The following expression shows how an integer, and a string identifier are declared. It should be always initiated to a value.

$$IdentifierDec \rightarrow \boldsymbol{Int}\ \ Identifier \quad ' =' \ IntegerConstant;$$

$$\mid \boldsymbol{String}\ \ Identifier \quad ' =' \ StringLiteral;$$

3.3.1.3 identifier

Check section 3.3.1

3.3.1.4 functionDec:

A programmer is able to define his own functions. A function declarations starts with the reserved word Func followed by a function name, then an optional set of arguments, followed by statements of the function body, and ends with the Endfunc keyword

$$functionDec \rightarrow \boldsymbol{Func}\ \ FuncName\ \ FuncBody\ \boldsymbol{Endfunc}$$

3.3.1.5 FuncName:

$$FuncName \rightarrow identifier$$

3.3.1.6 FuncBody:

Here all statements of a function is specified

$$FuncBody \rightarrow statement *$$

For *statement*, check section 3.3.3.1

3.3.2 WindowSpecification

This section is mandatory. It starts with " Window [" and ends with "]". It sets the general parameters of the output graphics window

$$WindowSpecification \rightarrow \boldsymbol{Window}\, '['\ WindowSize \quad ';'\ ObjectColor';'\ ']'$$

3.3.2.1 WindowSize:

WindowSize sets the size of the output windows in pixels. It is the first parameter in windowSpecification ***(****width in pixels, height in pixels****)***

$$WindowSize \rightarrow\ '('\ integer \quad ','\ integer\ ')'$$

3.3.2.2 ObjectColor*:*

the third parameter is ObjectColor. It sets the color of the drawings

$$ObjectColor \rightarrow Color$$

3.3.2.3 Color:

A window background or a shape color could be one of the following:

Black, White, Blue, Red

$$Color \rightarrow \boldsymbol{Black} \mid \boldsymbol{White} \mid \boldsymbol{Blue} \mid \boldsymbol{Red}$$

3.3.3 ObjectSection

This section is mandatory. It starts with " Object [" and ends with "]". Statements are added to this section.

$$ObjectSection \rightarrow \boldsymbol{Object}\, '['\ Statement * \ ']'$$

3.3.3.1 Statement:

There are 3 kinds of statements. They are executed in sequence according to their appearance in the object section.

$$Statement \rightarrow ConditionalStatement | IterationStatement | Expression$$

3.3.3.2 ConditionalStatement

Conditional statements is represented by If , else clause. In the first expression, If the the equality test is true, the first statement is executed, otherwise the second statement is executed. In the second if statement, if the equality test is true, then the statement is executed otherwise nothing happens.

$$ConditionalStatement \rightarrow \boldsymbol{If} \quad '(\quad 'EqualityTest \quad ')'\ statement * \ \ \boldsymbol{else}\ statement * \ \ \boldsymbol{Endif}$$

$$| \boldsymbol{If} \quad '(\quad 'EqualityTest \quad ')'\ statement * \quad \boldsymbol{Endif}$$

3.3.3.3EqualityTest

Equality test returns 1 if both expressions are equal, 0 otherwise

$$EqualityTest \rightarrow ArithExp \quad '=='\ ArithExp$$

3.3.3.4 IterationStatement

In this iteration statement, The integer between parentheses represents the number of times the list of statements is executed.

$$IterationStatement \rightarrow \boldsymbol{Loop} \quad '(' \ integer \quad ')' \ statement * \ \boldsymbol{Endloop}$$

3.3.3.5 Expressions

There are three kinds of expressions: Assignment Expressions, ObjectCalls, and FunctionCalls

$$Expression \rightarrow AssingExp \quad ';' \,|ObjectCall \quad ';' \,|FunctionCall';'$$

3.3.3.6 Assignment Expression:

$$AssingExp \rightarrow identifier \quad '=' \ (ArithExp|StringLiteral)$$

3.3.3.7 ArithExp:

Operations are left-associative. '/' and '*' have higher precedence than '+' and '-'

$$\begin{aligned} ArithExp \rightarrow & '(' ArithExp ')' \\ & |ArithExp \quad '+' ArithExp \\ & |ArithExp \quad '-' \ ArithExp \\ & |ArithExp \quad '*' \ ArithExp \\ & |ArithExp \quad '/' \ ArithExp \\ & |Identifier \\ & |IntegerConstant \end{aligned}$$

3.3.3.8 Function calls:

$$FunctionCall \rightarrow \ \boldsymbol{Callf} \ FuncName \quad '(' \ actualArguments *')'$$

$$actualArguments \rightarrow ArithExp \mid actualArguments \quad ',' \ ArithExp$$

this language supports Applicative-order evaluation which means that the function arguments are evaluated first from left to right before executing the body of the function.

3.3.3.9 ObjectCall:

To draw shapes, we can use object calls to draw a specific shape. We have 4 main shapes:

Ellipse: To draw an ellipse we need to call the Ellipse functions: Ellipse (rx, ry, x, y)
this function draws an ellipse with horizontal radius rx, vertical radius ry and center at point(x, y)

Rectangle: To draw a rectangle, we need to call: Rectangle (w, h, x, y)
This function draws a rectangle with width w, height h, and the lower corner at point (x, y)

Line: to draw a line, we need to call: Line (x1, y1, x2, y2)
This function Draws a line from point (x1, y1) to point (x2, y2)

Text: to type a test in the output window, we need to call Text (" string", x, y);
This function Prints a string starting from point (x, y).

$$ObjectCall \rightarrow \textbf{\textit{Ellipse}} \quad '(\quad 'ArithExp \quad ',' ArithExp \quad ',' \quad ArithExp \quad ',' \quad ArithExp)$$

$$| \textbf{\textit{Rectangle}} \quad '(' ArithExp \quad ',' ArithExp \quad ',' \quad ArithExp \quad ',' \quad ArithExp)$$

$$|\textbf{\textit{Line}} \quad '(' ArithExp \quad ',' ArithExp \quad ',' \quad ArithExp \quad ',' \quad ArithExp)$$

$$| \textbf{\textit{Text}} '(\quad 'StringLiteral|identifier \quad ',' \quad ArithExp \quad ',' \quad ArithExp)$$

3.4 Lexical Scope:

Identifiers, objects and keywords all fall into the same name space. If a there are two functions or identifiers of the same name, then the second one overwrites the first one.

Doodle uses static scoping. All variables are global. their lives begin where it is declared in Declare section, and ends at the end of Object section.

3.5 Associatively and Precedence of operator:

the following table demonstrates the precedence of operators, starting from the highest to the lowest precedence.

( )
/ *
+ -

'/', '*', '+', '-', are left associative

'=', and '==' are right associative

Chapter 4 Project Plan

Project Plan

4.1 Project Timeline

The following deadlines were set for this project at the beginning of the semester

Date	Module Completed
February 10, 2009	Language Proposal
March 10, 2009	LRM
March 26, 2009	Lexer
April 2, 2009	Parser
April 16, 2009	AST
April 23, 2009	Code Generation
May 7, 2009	Testing
May 14, 2009	Project Report

4.2 Software Development Environment

Operating system

Both Windows and Linux were used for this project. Windows were used to test each step of the project. Linux was used for the last part of testing.

Language Used

Ocamllex was used for the scanner, Ocamlyacc was used to the parser, and ocaml was used for the rest of the code.

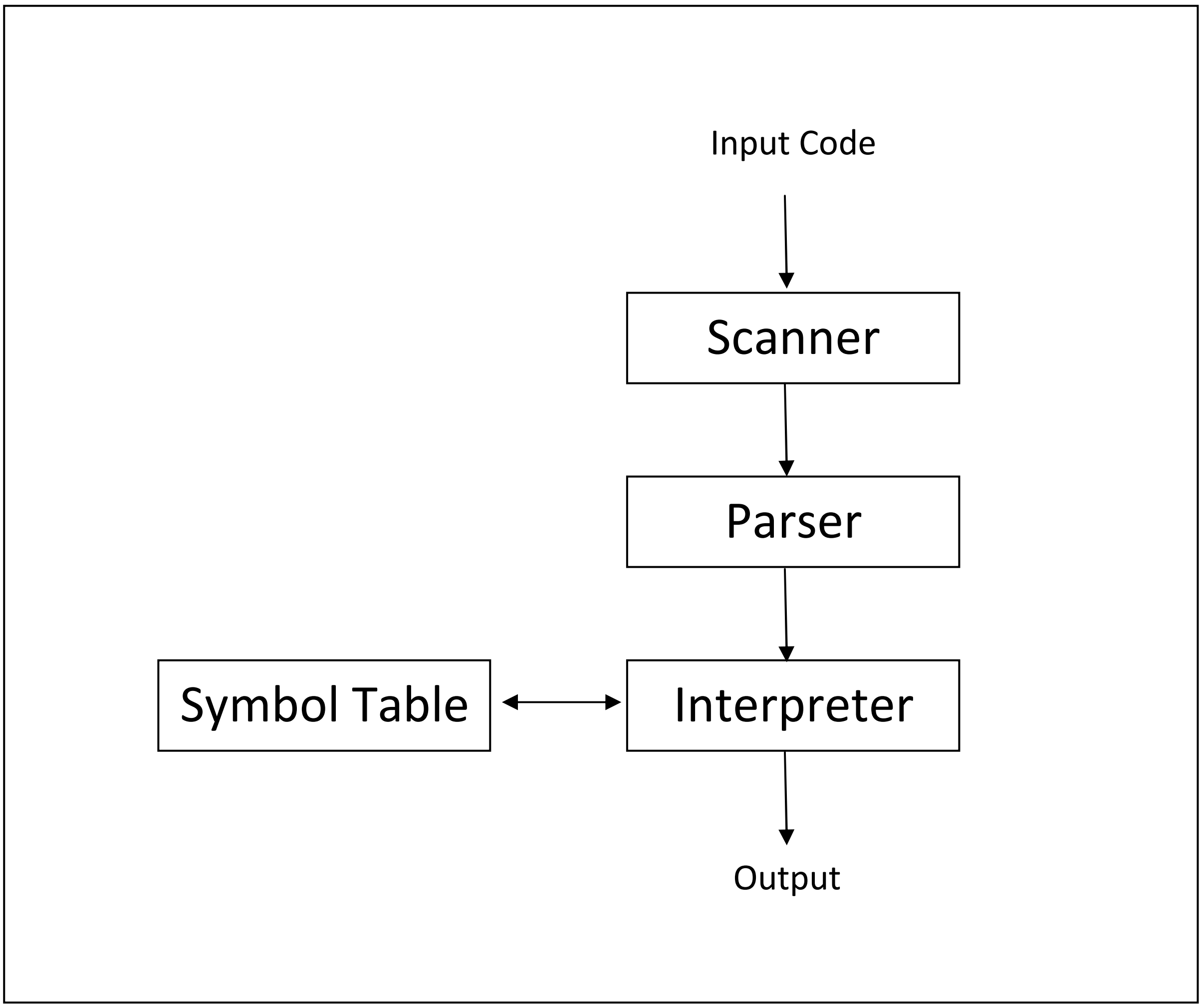
4.3 Project Log

Date	What I did
April 18, 2009	Working on make file, and scanner
April 25, 2009	Running a complete scanner with a simple parser and interpreter
April 29, 2009	Testing the graphic library
May 3, 2009	Scanner completed Working with parser, and AST in parallel
May 6, 2009	Creating Test cases
May 8, 2009	Parser AST completed
May 12, 2009	Final draft of project report Working on interpreter
May 16, 2009	Interpreter completed
May 18, 2009	Test case completed
May 18, 2009	Final Project completed

Chapter 5

Architectural Design

5.1 Block Diagram



5.2 Doodle Architecture

Doodle was implemented using Ocaml. It consists of several parts: laxer, parser and AST, and interpreter. The source code of this language has a " .d" extension, and it's out put is the graphics window.

5.2.1 Lexer:

A lexer reads an input file, converts characters and symbols into token. In this stage white spaces and comments are removed. Then, tokens are passed to the parser.

5.2.2 Parser and AST:

The parser gets the tokens from the lexer and creates an abstract syntax tree(AST) After assuring that the sequence of tokens doesn't violate the grammar rules of the language.

5.2.3 Interpreter:

The interpreter is mainly responsible for:

1. Walking through the created AST
2. Creating a symbol table and adding identified variables into it
3. Type checking
4. Evaluating arithmetic and Boolean expressions
5. Executing statements such as iteration, case, function statements, and drawing shapes

Chapter 6

Test Plan

Testing started from early stages of this project, where I tried to check that all files in a simple interpreter were executed successfully. The second testing was to ensure that the Graphic library worked on this interpreter. The third part, the main one, was after completing the whole interpreter. I tested all cases of the language to make sure that all parts of the interpret works successfully by taking inputs from standard in. The last part was by reading inputs from files.

There are 15 files, each one test a part of the language. The following table represents each input file.

File description	**File name**
No declare part, no global variables	Test_nodecl.txt
With declare section, and global variables	Test_decl.txt
With declare section No global variables	Test_emptydecl
Drawing an ellipse	Test_ellipse.txt
Drawing a rectangle	Test_rectangle.txt
Drawing a line	Test_line.txt
Drawing a text	Test_text.txt
If statement	Test_if1.txt Test_if2.txt Test_ifelse1.txt Test_ifelse2.txt
Arithmetic operation	Test_arith
Loop	Test_loop.txt
User defined function without arguments(local variables)	Test_func.txt

Chapter 7

Lesson learned

I had learned several programming languages in the past, and always wondered who were behind those languages, and why was I always learning how to use a language instead of learning how to create one. This project gave me the opportunity to work with languages from different aspect, which turned out to be both interesting and challenging at the same time.

The interesting part was having the freedom to choose my own syntax, and the level of complexity in my language. The challenging part was working with Ocaml. It took me some time to get used to its new style, and to understand its semantic. But even though I prefer other languages, I'm glad that I was exposed to a different way of programming. I got the chance to think differently, and code less.

I also learned how time is valuable. If I had the chance to start all over again, I would spend less time in the scanner part, more time in the interpreter, and testing part.

Chapter 8

Appendix

8.1 Doodle Grammar :

The following is a list of Doodle grammar . The start symbol is ***DoodleProgram***

Identifier → Letter(Letter | Digit)*

IntegerConstant → digit +

StringLiteral → ' " ' (Letter | Digit | ' ')+ ' " '

Letter → ['a'–'z' 'A' – 'Z']

Digit → ['0' – '9']

DoodleProgram → Declaration? WindowSpecification ObjectSection

Declaration → **Declare** '['DeclSpecification ∗ ']'

DeclSpecification → IdentifierDec
| functionDec

IdentifierDec → **Int** Identifier '=' IntegerConstant ';'
|**String** Identifier '=' StringLiteral';'

functionDec → **Func** FuncName FuncBody **Endfunc**

FuncName → identifier

FuncBody → statement ∗

WindowSpecification → **Window** '[' WindowSize ';' ObjectColor';' ']'

WindowSize → '(' integer ',' integer ')'

ObjectColor → Color

Color → **Black** | **White** | **Blue**| **Red**

ObjectSection → **Object** [' Statement ∗ ']'

Statement → ConditionalStatement
|IterationStatement
|Expression

ConditionalStatement → **If** ' ('EqualityTest ')' statement ∗ **Else** statement ∗ **Endif**
|**If** ' ('EqualityTest ')' statement ∗ **Endif**

$$
\begin{aligned}
IterationStatement \rightarrow\ & \boldsymbol{Loop}\quad '('\ integer\quad ')'\ statement * \ \boldsymbol{Endloop}\\
Expression \rightarrow\ & idenfitier\quad '='\ (ArithExp|StringLiteral);\\
& |ObjectCall;\\
& |FunctionCall;\\
ArithExp \rightarrow\ & ‘('ArithExp')'\\
& |ArithExp\quad '+'ArithExp\\
& |ArithExp\quad '-'\ ArithExp\\
& |ArithExp\quad '*'\ ArithExp\\
& |ArithExp\quad '/'\ ArithExp\\
& |Identifier\\
& |IntegerConstant
\end{aligned}
$$

$$
EqualityTest \rightarrow\ ArithExp\quad '=='\ ArithExp
$$

$$
FunctionCall \rightarrow\ \boldsymbol{Callf}\ FuncName\quad '('\ \ actualArguments\ *')'
$$

$$
\begin{aligned}
actualArguments \rightarrow\ & ArithExp\\
& |actualArguments\quad ','\ ArithExp
\end{aligned}
$$

$$
\begin{aligned}
ObjectCall \rightarrow\ & \boldsymbol{Ellipse}\quad '(\quad 'ArithExp\quad ','\ ArithExp\quad ','\ \ ArithExp\quad ','\ \ ArithExp)\\
& |\ \boldsymbol{Rectangle}\quad '('\ ArithExp\quad ','\ ArithExp\quad ','\ \ ArithExp\quad ','\ \ ArithExp)\\
& |\boldsymbol{Line}\quad '\big('ArithExp\quad ','ArithExp\quad ','\ \ ArithExp\quad ','\ \ ArithExp\big)\\
& |\ \boldsymbol{Text}\ '\big(\quad 'StringLiteral|identifier\quad ','\ \ ArithExp\quad ','\ \ ArithExp\big)
\end{aligned}
$$

8.2 Doodle Code

8.2.1 scanner.mll

```
{ open Parser }                         (* Get the token types *)

rule token = parse
[' ' '\t' '\r' '\n'] { token lexbuf }    (* Whitespace *)
| "<*" { comment lexbuf }               (* Comments *)
| '(' { LPAREN }                        (* separators*)
| ')' { RPAREN }
| '[' { LBRACK }
| ']' { RBRACK }
| ';' { SEMI }
| ',' { COMMA }

| '+' { PLUS }                           (* operators *)
| '-'  { MINUS }
| '*' { TIMES }
| '/' { DIVIDE }
| '=' { ASSIGN }
| "==" { EQ }

| "If" { IF }                            (* keywords *)
| "Else" { ELSE }
| "Endif" { ENDIF }
| "Loop" { LOOP }
| "Endloop" { ENDLOOP }
| "Func" { FUNC }
| "Endfunc" { ENDFUNC }
| "Callf" { CALLF }
| "Red" { RED }
| "Blue" { BLUE }
| "White" { WHITE }
| "Black" { BLACK }
| "Green" { GREEN }
| "Rectangle" {RECTANGLE }
| "Ellipse" { ELLIPSE }
| "Line" { LINE }
| "Text" { TEXT }
| "String" { STRING }
| "Int" { INT }
| "Declare" { DECLARE }
| "Window" { WINDOW }
| "Object" { OBJECT }

| eof { EOF }                           (* Endoffile*)
| ['0'-'9']+ as lxm { LITERAL(int_of_string lxm) }        (* integers *)
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9']* as lxm { ID(lxm) }(*idenfitiers*)
| '"' ['a'-'z' 'A'-'Z' '0'-'9' ' ' ]* '"' as lxm { ST(lxm) }
| _ as char { raise (Failure("illegal character " ^ Char.escaped char))}

and comment = parse
"*>" { token lexbuf } (* End of comment*)
| _ { comment lexbuf } (* Eat everything else *)
```

8.2.2 parser.mly

```
%{ open Ast %}

%token LPAREN  RPAREN  LBRACK  RBRACK SEMI  COMMA
%token PLUS  MINUS  TIMES  DIVIDE  ASSIGN  EQ
%token IF  ELSE  ENDIF  LOOP  ENDLOOP  FUNC  ENDFUNC CALLF
%token RED  BLUE  WHITE  BLACK GREEN
%token RECTANGLE  ELLIPSE  LINE  TEXT
%token STRING  INT  DECLARE  WINDOW  OBJECT  EOF

%token <int> LITERAL
%token <string> ID
%token <string> ST

%nonassoc ELSE
%left  ASSIGN
%left  EQ
%left PLUS  MINUS
%left TIMES  DIVIDE

%start doodle_program      /* entry point */

%type <Ast.doodle_program> doodle_program

%%


doodle_program:                    /*3 main sections of the program*/
decl_sec window_sec object_sec {( fst $1), ($3 :: snd $1) }
;


decl_sec:                              /*this section is optional */
/*nothing*/ {[],[]}
|DECLARE LBRACK declist RBRACK {$3}
;


declist:                               /*list of declarations*/
/*nothing*/ {[],[]}
|declist id_dec  {($2::fst $1), (snd $1) }   /*add integer,string variables declaration to the first list*/
|declist fun_dec { fst $1, ($2 :: snd $1) }   /*add function declaration to the second list*/
;

id_dec:                              /*declaring an identifier*/
INT ID ASSIGN LITERAL  SEMI   /*integer variable*/
{{ vname= $2; vtype="int" ; ivalue=$4; svalue="none"}}

|STRING ID  ASSIGN ST  SEMI   /*string variable*/
{{ vname= $2; vtype="string" ; ivalue=0; svalue=$4}}
;

fun_dec:
FUNC  ID   stmt_list ENDFUNC
{{ fname= $2;
   fbody= List.rev $3}} /*function declaration*/
;
```

```
window_sec:   /*winodw section*/
WINDOW  LBRACK   LPAREN LITERAL COMMA LITERAL RPAREN SEMI  color RBRACK
{Graphics.open_graph " string_of_int($4) ^ 'x' ^ string_of_int($6) ";Graphics.set_color $9}
;

color:
RED  {0xff0000}
|BLUE {0x0000ff}
|WHITE {0xffffff}
|BLACK {0x000000}
|GREEN {0x00ff00}
;


object_sec:   /*object section */
OBJECT LBRACK stmt_list  RBRACK
{{ fname="Object";
   fbody= List.rev $3}}
;

stmt_list:
/*nothing*/ {[]}
|stmt_list stmt {$2::$1}
;


stmt:
exp SEMI {Exp($1)}
|IF  LPAREN eqtest RPAREN stmt_list ELSE  stmt_list ENDIF {If($3, $5, $7)}
|IF  LPAREN eqtest RPAREN stmt_list ENDIF {If($3, $5,[ ] )}
|LOOP LPAREN LITERAL RPAREN stmt_list ENDLOOP{Loop($3,$5) }
|CALLF ID SEMI {Call_f($2)}
;

eqtest:
arith_exp EQ arith_exp {Eqtest($1, $3)}
;

exp:
ID ASSIGN arith_exp{Assign_i($1, $3)}
|ID ASSIGN  ST  {Assign_s($1, $3)}
|RECTANGLE  LPAREN arith_exp  COMMA arith_exp COMMA arith_exp  COMMA arith_exp  RPAREN
{Rec($3,$5,$7,$9)}
|ELLIPSE  LPAREN arith_exp  COMMA arith_exp COMMA arith_exp  COMMA arith_exp  RPAREN
{Elp($3,$5,$7,$9)}
|LINE     LPAREN arith_exp  COMMA arith_exp COMMA arith_exp  COMMA arith_exp  RPAREN
{Line($3,$5,$7,$9)}
|TEXT  LPAREN ST  COMMA arith_exp COMMA arith_exp  RPAREN {Txt_s ($3,$5,$7)}
|TEXT  LPAREN ID COMMA arith_exp COMMA arith_exp  RPAREN {Txt_id ($3,$5,$7)}
;

arith_exp:  /*arithmatic operations, done on integers and id of integers*/
ID {Id($1)}
|LITERAL {Literal($1)}
|LPAREN arith_exp RPAREN  {$2}
|arith_exp PLUS arith_exp {Arith($1,Add, $3)}
|arith_exp MINUS arith_exp {Arith($1,Sub, $3)}
|arith_exp TIMES  arith_exp {Arith($1,Mult, $3)}
|arith_exp DIVIDE arith_exp {Arith($1,Div, $3)}
```

8.2.3 ast.mli

```
type doodle_program= decl_sec

and decl_sec = declist

and declist=(id_dec list) * (fun_dec list )

and id_dec=
{ vname: string;
  vtype: string;
  ivalue: int;
  svalue: string;}

and fun_dec=
{ fname: string;
  fbody: stmt_list;}

and window_sec= unit

and color=int

and object_sec= fun_dec

and stmt_list= stmt list

and stmt=
Exp of exp
|If of eqtest  * stmt_list * stmt_list
|Loop of int * stmt_list
|Call_f of string

and eqtest=
Eqtest of arith_exp * arith_exp

and exp=
 Assign_i of string * arith_exp
|Assign_s of string * string
|Rec  of arith_exp * arith_exp * arith_exp * arith_exp
|Elp  of arith_exp * arith_exp * arith_exp * arith_exp
|Line   of   arith_exp * arith_exp * arith_exp * arith_exp
|Txt_s of string *  arith_exp * arith_exp
|Txt_id of string *  arith_exp * arith_exp


and arith_exp=
Id of string
|Literal of int
|Arith of arith_exp * op * arith_exp


and op= Add|Sub|Mult|Div
```

8.2.4 interpret.ml

```
open Ast
open Graphics

module NameMap = Map.Make(struct
   type t = string
   let compare x y = Pervasives.compare x y   end)

exception ReturnException of int * int NameMap.t


(*     begin of run function  *)
let run (vars, funcs)=

(* Put function declarations in a symbol table *)
  let func_map = List.fold_left (fun fmap fun_declist ->NameMap.add fun_declist.fname fun_declist
fmap ) NameMap.empty funcs

in

 (*   begin of call function  *)
  let rec call fun_declist globals=

 (*function for evaluating arithmatic expression, returning the value and environment*)
    let rec eval_arith env=function
        Literal(i) -> i, env
      | Id(var)-> let globals= env
                  in
                  if NameMap.mem var globals then
                                begin
                                if ((NameMap.find var globals).vtype="int") then
                                       (NameMap.find var globals).ivalue, env
                                else
                                       raise (Failure ("this identifier is not of type int"))
                                end
                  else raise (Failure ("undeclared identifier " ^ var))

      | Arith(e1, op, e2) -> let v1, env = eval_arith  env e1
                               in
                               let v2, env = eval_arith env e2
                               in

                              (match op with
                                  Add -> v1 + v2
                                | Sub -> v1 - v2
                                | Mult-> v1 * v2
                                | Div ->  v1 / v2), env

    in
(* function for evaluating equality test part*)
let rec eval_equality env=function

Eqtest(ae1,ae2)->let v1, env = eval_arith env ae1
                         in
                         let v2, env = eval_arith env ae2
                         in
                         let boolean i = if i then 1 else 0
                         in boolean (v1 = v2), env
```

```
in
(*function for executing expressions*)
let rec exec_exp env=function
   Assign_i(var,ae)-> let v, globals = eval_arith env ae
               in
               let irecord ={ vname= var; vtype="int" ; ivalue=v ; svalue="none"}
               in
                 if NameMap.mem var globals then
                     begin
                     if (NameMap.find var globals).vtype="int" then
                         (NameMap.add var irecord globals)
                     else
                          raise(Failure("identifier is not of type int"))
                     end
                 else raise (Failure ("undeclared identifier" ^ var))


 |Assign_s(var, s)->let globals =env
              in
              let srecord={ vname= var; vtype="string" ; ivalue=0 ; svalue=s}
              in
                if NameMap.mem var globals then
                     begin
                     if (NameMap.find var globals).vtype="string"then
                         (NameMap.add var srecord globals)
                     else
                          raise(Failure("identifier is not of type string"))
                     end
                 else raise (Failure ("undeclared identifier" ^ var))


 |Rec(ae1,ae2,ae3,ae4)->let v1, env = eval_arith env ae1
                in
                let v2, env = eval_arith env ae2
                in
                let v3, env = eval_arith env ae3
                in
                let v4, env = eval_arith env ae4
                in
                Graphics.draw_rect v1 v2 v3 v4;
                env


 |Elp(ae1,ae2,ae3,ae4)->let v1, env = eval_arith env ae1
                in
                let v2, env = eval_arith env ae2
                in
                let v3, env = eval_arith env ae3
                in
                let v4, env = eval_arith env ae4
                in
                Graphics.draw_ellipse v1 v2 v3 v4;
                env

 |Line(ae1,ae2,ae3,ae4)->let v1, env = eval_arith env ae1
                in
                let v2, env = eval_arith env ae2
                in
```

```
                 let v3, env = eval_arith env ae3
                 in
                 let v4, env = eval_arith env ae4
                 in
                 Graphics.moveto v1 v2;
                 Graphics.lineto v3 v4;
                env

  |Txt_s(str,ae1,ae2)->let v1, env = eval_arith env ae1
                 in
                 let v2, env = eval_arith env ae2
                 in
                 Graphics.moveto v1 v2;
                 Graphics.draw_string str;
                 env

  |Txt_id(str,ae1,ae2)->let v1, env = eval_arith env ae1
                 in
                 let v2, env = eval_arith env ae2
                 in let st=
                   let globals = env
                   in
                   if NameMap.mem str globals then
                   begin
                     if (NameMap.find str globals).vtype="string" then
                         (NameMap.find str globals).svalue
                     else raise (Failure ("identifier is not a string"))
                   end
                   else raise (Failure ("undeclared identifier" ^ str))
                 in
                 Graphics.moveto v1 v2;
                 Graphics.draw_string st;
                 env

in
(*function for executing stmt part*)
 let rec exec_stmt env = function
    Exp(e)->let env=exec_exp env e
       in env
    |If(et, slist1, slist2)->let test, env= eval_equality env et
            in List.fold_left exec_stmt env (if test !=0 then slist1 else slist2)

|Loop(i, slist)-> let rec looping env i=
                  let j= i-1
                   in
                   if j!=0 then
                    begin
                    looping  ( List.fold_left exec_stmt env slist ) j
                    end
                   else  env
                in looping env (i+1)

    |Call_f(str)->if NameMap.mem str func_map
             then let frecord=NameMap.find str func_map
                 in
                 List.fold_left exec_stmt env frecord.fbody
             else raise (Failure ("undefine function" ^ str))
```

```
in
(*  body of call *)

List.fold_left exec_stmt globals  fun_declist.fbody; (*end of call function*)
in

(* body of run *)

(* Put variables declarations in a symbol table *)
let var_map = List.fold_left (fun vmap id_dec ->NameMap.add id_dec.vname id_dec vmap)

NameMap.empty vars
in

try
   call(NameMap.find "Object" func_map) var_map
with Not_found ->
raise (Failure ("did not find the Object section function"))
```

8.2.5 doodle.ml

```
let _ =
let lexbuf = Lexing.from_channel stdin in
let program= Parser.doodle_program  Scanner.token lexbuf in Interpret.run program; read_line ();
```