Final Report

VideO Processing Language

(VOPL)

Baolin shao (bs2530)

Xuyang Shi (xs2137)

Huning Dai (hd2210)

Jia Li (jl3272)

1. Introduction

1.1 Motivation

Thanks to the information technology, now we are living a digital life, and gone are the days, when people must have their pictures developed and put into albums. Instead, we began to use digital cameras and to save pictures in a computer years ago, which led to an increasing popularity of picture-editing tools. Currently, however, booming of multi-media requires us move one-step further: we not only want to easily process images but also to manipulate videos. Unfortunately, there are few languages that can help programmers get rid of miscellaneous routines in video processing and can let them only focus on developing core algorithms for videos. For example, C and C++ are effective, but they require experienced programmers to do everything from scratch, such as designing a video class or carefully calculating pixels' positions. Compared with C and C++, Matlab is more productive, because it provides mechanisms to directly manipulate images so that complicated image processing becomes simply calling Matlab’s build-in functions. It is, however, still impossible to directly work on videos. Therefore, we propose to develop a VideO Processing Language (VOPL) by which programmers can easily edit videos.

1.2 Description

VOPL is designed to be a language for simple processing and manipulating videos which has format of avi or yuv. It essentially allows any programmer with the knowledge of videos to process any data relevant to his purpose. VOPL is a C-like language. But what makes VOPL unique is that it facilitates video programming by incorporating English-like statements into the language, much as SQL does for databases. Besides, VOPL provides basic sequential and flow-control operations with a C-like semantics.

1.3 Language Features

1.3.1 Basic Data Type

There are 3 basic data types in VOPL. They are int, float and char, whose definition is the same as that in C.

1.3.2 Key Words

All the key words are written in the lower case. The complete listing of the keywords is given in the language reference manual.

1.3.3 Comparative and Mathematical Operators

Operators are essential for condition comparison and calculation in video processing language like VOPL. There are many such operators in VOPL: +, -, *, /, %, ==, !=, >, >=, <, <= and so on.

1.3.4 Special Features of VOPL

There is a new data type in VOPL. It is video. Video type is used in the declaration of a variable which is used to store a video that will be processed with the program. A declaration of a video type variable can be like this:

***video** v1;*

VOPL also has some English-like statements that have something to do with the manipulation of a video:

```
load v1 from "/film1.yuv" with 128 and 128;
store v1 to "/film1.avi";
insert v1 to v2 from 15;
delete v1 from 1 to 4;
copy v1 from 5 to 8 to v2;
update v1 from 5 to 10 {…}
```

2. Language Tutorial

2.1 VOPL Functionality

- Comments - /*   */
- Types - int, float, char
- Declaration - variable, array
- Expression - primary, assignment, binary, unary, (expression), functioncall
- Statements - expression statement, compound, selection, iteration, jump,

video statement

2.2 Compile VOPL

The VOPL code is stored as *.v file. To compile this file, call vopl *.v and the output is a *.m file which contains intermediate matlab code. Then run this *.m file in matlab. The output is the final result.

2.3 Sample Program

Suppose we have three videos news.avi (352*288), news2.yuv (176*144) and coastguard.avi (352*288) each of which has 300 frames. Given below is a simple program in VOPL to firstly smooth news2.yuv from frame 50 to 100, delete news.avi from 1st frame to 150th frame, and then connect coastguard.avi to the end of the segment of news2.yuv from 51st frame to 100th frame, and finally insert the connected segment to the modified news.avi from 76th frame.

```
void main()
{ video v1,v2;
   video v3,v4;
   load v1 from "news.avi" with 352 and 288;    /* the original video can be .avi or .yuv*/
   load v2 from "news2.yuv" with 176 and 144;
   load v3 from "coastguard.avi" with 352 and 288;
update v2 from 50 to 100
{ int i,j;
   for (i=1;i<144;i=i+1)
   for (j=1;j<176;j=j+1)
   this(i,j)=(this(i,j)+this(i+1,j)+this(i,j+1)+this(i+1,j+1))/4;
}
delete v1 from 1 to 150;
copy v2 from 51 to 100 to v4;
insert v3 to v4 from 50;
insert v4 to v1 from 76;

store v1 to "news-composed.avi";      /* the modified video can be stored as .avi or .yuv*/

}
```

3. Language Reference Manual

3.1 Lexical Conventions

The first compilation phase is lexical analysis. In this phase, a program is seen as a stream of input characters in which some are discarded, such as white spaces and tabs, and others form different tokens. In VOPL, there are 5 kinds of tokens which are identifiers, keywords, operators, constants and string literals. Besides, blanks, horizontal and vertical tabs, newlines, formfeeds, and comments are ignored except they separate tokens.

3.2 Comments

Comments begins with /* and ends with */, including everything in between. These characters will be discarded after lexical analysis. Comments do not nest, and they do not appear in any string literal.

3.3 Identifiers

An identifier is a sequence of digits and letters. But the first character must be a letter. Identifiers cannot be distinguished by uppercase or lowercase. Length of an identifier cannot exceed 32 bits. Otherwise a compile-time error will be reported.

3.4 Keywords

The following identifiers are exclusively reserved as keywords which can not be used otherwise:
int float char if for else while void return break
continue load store insert delete copy update from
to with video and this

3.5 Operators

There are several kinds of operators as following. For all kinds of operations, there is no type conversion in VOPL.

3.5.1 Binary Operator

VOPL provides some fundamental algebraic operations such as plus, minus, times, divide and mod. The corresponding binary operators are +, -, *, / and %. +, – associate from left to right. So do * and /. But * and / have higher priority level.

3.5.1.1 expression + expression
The binary operator + indicates addition. The result gives the sum of the values of two expressions. The type of the two operands can be int and float. But both should be the same type. And the result of the operation will give the same type as operands. Other types of operands are illegal.

3.5.1.2 expression – expression
The binary operator – indicates subtraction. The result gives the difference between the values of two expressions. The type consideration for operands is the same as +.

3.5.1.3 expression * expression
The binary operator * indicates multiplication. The type consideration for operands is the same as +.

3.5.1.4 expression / expression
The binary operator / indicates division. When division is applied for two integers, remainder is always discarded. The type consideration for operands is the same as +.

3.5.1.5 expression % expression
The binary operator % indicates delivery. The result gives the remainder from the division of the first expression to the second. The type of the two operands must be both int. And the type of result is int.

3.5.2 Unary Operator

There are two unary operators in VOPL. One is logical negation "!". It is used in the form of !expression, which will give a result of 1 when the expression has a value 0, and give a result of 0 when the expression has a non-zero value. This operator is applicable only to int and char. And the type of result is int. The other is minus "-". It is used in the form of –expression, which will give the negative of the expression with the same type of the expression. This operator is applicable to int, float and char.

3.5.3 Comparison Operator

The comparison operators in VOPL include less than, greater than, less equal and greater equal, which are presented as <, >, <=, >= associating from left to right.

3.4.3.1 expression < expression
3.4.3.2 expression > expression
3.4.3.3 expression <= expression
3.4.3.4 expression >=expression
The above four comparison operations are used to show the relation between two expressions. When the specified relation is false, the result gives 0; when the specified relation is true, the result gives 1.

3.5.4 Equality Operator

Equality operators include equal (==) and not equal (!=).

3.5.4.1 expression == expression
3.5.4.2 expression != expression
The comparison operators == and != have lower priority level than >, <, <= and >=. == and != are used to show the logical relation between the two expressions is equal or not equal. It has the same rule as showed in comparison operator.

3.5.5 Logical Operator

Logical operators include logicaland (&&) and logicalor (||) associating from left to right.

3.5.5.1 expression && expression
The result of the && operation is 0 when one or both of the expressions equal to 0; and the result will be 1 when both of the expressions equal to 1. If the first expression is evaluated to be 0, the second one will not be evaluated, and result gives out 0 directly. If the first expression is not equal

to 0, the second one will be evaluated. The result gives 0 when the second expression is equal to 0, otherwise 1. The type of the result is int. And the two operands may not have the same type but must have the same arithmetic type.

3.5.5.2 expression || expression

The result of the || operation is 1 when one or both of the expressions equal to 1; and the result will be 0 when both of the expressions equal to 0. If the first expression is evaluated to be 1, the second one will not be evaluated, and result gives out 1 directly. If the first expression is not equal to 1, the second one will be evaluated. The result gives 1 when the second expression is equal to 1, otherwise 0. The type of the result is int. And the two operands may not have the same type but must have the same arithmetic type.

3.6 String Literals

A string literal is a sequence of characters surrounded by “ and ”. It can only appear in load and store statements as file names. Using string literals in any other places will cause errors. It cannot be modified, or assigned to a character array. String literals are different even when they have the same value.

3.7 Constants

There are three kinds of constants: integer constant, floating constant and character constant.

3.7.1 Integer

An integer constant is a sequence of digits. All the integer constants are viewed as decimal. The associated keyword is int.

3.7.2 Floating

A floating constant consists of an integer part, a decimal point and a fraction part. Both integer part and fraction part consist of a sequence of digits. The associated keyword is float.

3.7.3 Character

A character constant is a sequence of one or more characters in a pair of single quotes as ‘x’. The associated keyword is char.

3.8 Types

3.8.1 Basic Types

There are several basic types in VOPL as following:

int is a type for an integer.
float is a type for a single precision floating number.
char is a type for a sequence of one or more characters.

3.8.2 Video Types

Video can only be type of a variable, not function. A video variable cannot be an lvalue, and thus cannot be assigned. Video variables can be modified only by video statements.

3.8.3 Void

Void can only be type of a function. In this case, this function does not return any value to its caller function. A function with void type cannot constitute a right value of an assignment.

3.9 Declarations

Variable declaration states type and names of variables. There is no difference between declaration and definition, and the scope of a variable starts from the next statement/declaration of its own declaration. Declarations must be written before any statement of the same block, and a declaration has two variations, variable declaration and array declaration.

declaration_list_opt:

*/*empty*/*
|declaration_list

declaration_list:
declaration
| declaration_list declaration

declaration:
variable_declaration SEMICOLON
| array_declaration SEMICOLON

3.9.1 Variable Declaration

In general, a variable declaration is in the following form:

variable_declaration:
dtype ID_list

ID_list:
ID_Init
| ID_list COMMA ID_Init

ID_Init:
ID
| ID ASSIGN constant

dtype is one of the four pre-defined types in VOPL and ID_list is a list of variable names with the same type, separated by comma. The type of a variable cannot be void, because in VOPL void is not a type, instead it is a function specifier. Variable can optionally be initialized in declaration by ID_Init. In variable declaration, the variable can only be assigned constant as shown in ID_Init. The constant value must have the same type of the variable.

3.9.2 Array Declaration

Arrays can be declared as follow:

array_declaration:
dtype ID LBRACKET INTCONST RBRACKET

VOPL only supports one-dimension arrays, because most of operations on video need "matrix-like" data structures and we provide this_expression to meet this end. Requirements on type and ID are the same as variable declaration. INTCONST, a constant integer, specifies the length of an array. Unlike variables, programmers cannot initialize arrays, instead all arrays will default 0 as their initial values.

3.10 Expression

There are several kinds of expressions in VOPL such as primary expression, assignment expression, binary expression, unary expression, expression in parentheses, and function call.

```
expression:
     primary_expression
  |  assignment_expression
  |  binary_expression
  |  unary_expression
  |  LPAREN expression RPAREN
  |  functioncall
```

3.10.1 Primary Expression

Primary expressions are lvalues and constants. Value of a primary expression is that of its sub-expression.

```
primary_expression:
  lvalue
| constant
```

3.10.1.1 lvalue

An lvalue is something that can appear on the left side of an assignment. It represents memory locations in the running environment. As an expression, the value of an lvalue is that of its sub-expressions. In VOPL, an lvalue is defined by:

```
lvalue:
  ID
| ID LBRACKET expression RBRACKET
| this_expr
```

ID can only be the name of a declared variable or array. The value of the array index must be non-negative integer, otherwise an error will occur. this_expr (See 3.10.5) is also an lvalue, but it can only be used in an update statement. This syntax provides a secure way to revise the content of a video.

3.10.1.2 Constant

A constant can be a primary expression. Its type is as showed in 3.8.1

```
constant:
      INTCONST
  | CHARCONST
  | FLOATCONST
```

3.10.2 Assignment Expression

An assignment expression is a kind of expression with the form of ‘lvalue=expression’. Lvalue is as showed in 3.10.1.1. Only lvalue can appear at the left side of assignment operator =, and others are not allowed. Assignment operator = associates from right to left. When the assignment takes place, the value of expression on the right will be stored in the operand at left. The operands on both sides must have the same arithmetic type. Value of an assignment expression is that of the expression on the right side of “=”.

```
assignment_expression:
lvalue ASSIGN expression
```

3.10.3 Binary Expression

A binary expression is a kind of expression with the form of expression operator expression. Value of a binary expression is the result of the operation specified by operator, on the value of two sub-expressions.

```
binary_expression:
  expression PLUS expression
|expression MINUS expression
|expression TIMES expression
|expression DIVIDE expression
|expression MOD expression
```

|expression LESSTHAN expression
|expression GREATERTHAN expression
|expression LESSEQUAL expression
|expression GREATEREQUAL expression
|expression EQUAL expression
|expression NOTEQUAL expression
|expression LOGICALAND expression
|expression LOGICALOR expression

3.10.4 Unary Expression

Unary expression has the form:

unary_expression:
NEGATION expression
| MINUS expression %prec UMINUS

Its value is the result of the unary operation on its sub-expression.

3.10.5 this_expr

In an update statement, to symbolically manipulate each frame, VOPL provides a dummy frame variable, called this. An expression, this(x,y), has the value of the pixel at column x and row y in some specific frame. The indexes in this_expr must be non-negative integers. The syntax of this expression is:

this_expr:
THIS LPAREN expression COMMA expression RPAREN

3.10.6 Function Call

A function call is also a kind of expression. It has a function designator (an identifier) followed by parentheses containing an empty or a list of comma-separated expressions. Before using function call, the function should be defined in front of the main function.

functioncall:
ID LPAREN expression_list_opt RPAREN

3.10.7 Expression List

Expression list is a series of expressions, separated by commas. The value of an expression list is that of the last expression in this list.

expression_list_opt
*/*empty*/*
| expression_list

expression_list:
expression
| expression_list COMMA expression

3.11 Statement

There are several kinds of statements in VOPL such as expression statement, compound statement, selection statement, iteration statement, jump statement and video statement.

3.11.1 Expression Statement

Most statements in VOPL are expression statements having the form as:

expression_statement:
expression SEMICOLON

the expressions inside are showed as 3.10. But most expressions in expression statement are assignments and function calls. In an expression statement in VOPL, the expression cannot be empty. If so, the compiler will report an error.

3.11.2 Compound Statement

Compound statement is consisted with several statements in a pair of braces, which is usually used in the body of a function definition. Its form is as following:

compound_statement:
    LBRACE declaration_list_opt statement_list_opt RBRACE

```
statement_list_opt:
     /*empty*/
  |statement_list

statement_list:
     statement
  |   statement_list statement
```

All declarations must be written before all statements.

3.11.3 Selection Statement

Selection statements have a form as following:

```
selection_statement:
     IF LPAREN expression RPAREN statement else_opt

else_opt:
     %prec NOELSE
  |   ELSE statement
```

The expression in the parentheses must have arithmetic type. When it equals to 1, the first statement is executed; when it equals to 0, the statement following ELSE will be executed if there is an else. If there is no else, selection statement will finish without any choice when the expression equals to 0.

3.11.4 Iteration Statement

Iteration statement has a form as following:

```
iteration_statement:
  WHILE LPAREN expression RPAREN statement
| FOR LPAREN expression SEMICOLON expression SEMICOLON expression RPAREN
statement
```

3.11.4.1 While loop

In WHILE statement, if the expression remains equal to 1, the substatement is executed repeatedly until the expression equals to 0. The expression must have an arithmetic type.

3.11.4.2 For loop
In FOR statement, the first expression is performed as initialization and is only executed once at the beginning of the loop. The second expression is evaluated before each iteration. If it equals to 1, the loop continues; if it equals to 0, the loop ends. The type of the second expression must be arithmetic type. The third expression is evaluated after each iteration. It performs as re-initialization.

3.11.5 Jump Statement

Jump statement transfer control unconditionally. It has a form as following:

```
jump_statement:
     CONTINUE SEMICOLON
  |   BREAK SEMICOLON
  |   RETURN expression SEMICOLON
```

3.11.5.1 Continue
A continue statement is only used in an iteration statement. It passes control to the loop-continuation portion of the smallest enclosing statement.

3.11.5.2 Break
A break statement is only used in an iteration statement. It can terminate the execution of the smallest enclosing statement no matter whether the ending condition of the iteration is reached or not. Control will pass to the statement following the terminated statement.

3.11.5.3 Return
Return statement terminates a function and passes the value of the expression followed by “return” to its caller function.

3.11.6 Video Statement

3.11.6.1 Load
The load statement will put data into a video variable. This instruction frees programmers from worrying how data are organized in a file.

LOAD ID FROM STRINGCONST WITH expression AND expression SEMICOLON

ID must be a variable defined with a build-in type video and the string constant indicates the path of the file. Expressions constitute the width and height of this video, so they must be they type of int.

3.11.6.2 Store

As its name tells, store statement will put the processed data back to file whose format can be .avi or .yuv.

STORE ID TO STRINGCONST SEMICOLON

The semantics of ID and STRINGCONST are the same as those in LOAD

3.11.6.3 Insert

Insert statement puts one video into another in a user-specified position.

INSERT ID TO ID FROM expression SEMICOLON

The first two IDs must be video type variables, but the expression must be a type of int, indicating one of the first video's frames after which the second video will be inserted. The statement returns the second video.

3.11.6.4 Delete

Delete statement deletes all frames in between a user-specified range.

DELETE ID FROM expression TO expression SEMICOLON

The first ID must be a video variable and the type of two expressions must be int.

3.11.6.5 Copy

Copy statement inserts a range of one video's frames into the end of the other one.

COPY ID FROM expression TO expression TO ID SEMICOLON

This statement finishes two tasks. First, it will read some data from the first video. Then it inserts this chunk of data into the end of the second video. The statement returns the second video.

3.11.6.6 Update

Update is the most powerful and complex instruction in VOPL. It allows programmers to design any algorithm and iteratively apply it to a sequence of frames.

UPDATE ID FROM expression TO expression compound_statement

The compound_statement contains any statement, except video statement. Function calls are prohibited either. The code in this compound_statement will be repeatedly applied to each frame between the specified range. During each iteration, the current frame is represented by a this_expr.

3.12 Scope

Programs usually are not written in one chunk of code. Instead, programmers divide a program into different parts, each of which does not conflict. We call every part a scope. Here, every function constitutes a scope. Identifiers in one scope do not conflict with those in another, even if they have the same lexical representation.

3.13 Program

A program is a compilation unit, and VOPL does not support multiple files. Each program consists of one main function, which is compulsory, and multiple functions, which is optional, but each must be written before main.

```
program:
     func_list_opt EOF

func_list_opt:
   /*empty*/
  | func_list

func_list:
  |    func
  |    func_list func
```

3.14 Function

A function is a portion of code within a larger program, which performs a specific task and can be relatively independent of the remaining code. A function can be defined as follow:

```
func:
dtype ID LPAREN argument_list_opt RPAREN compound_statement

argument_list_opt:
   /*empty*/
  | argument_list

argument_list:
     argument
  |   argument_list COMMA argument

argument:
   dtype ID
```

dtype tells what a function will return to its caller function. If specified by basic type, the function will return a value of that type, otherwise, specified by VOID, the function will return nothing. Argument_list_op defines a function's parameters. All the parameters are local variable of this function. Each argument declaration must have a dtype and an ID, and multiple arguments are separated by comma. It is also legal that a function does not have any parameter.

3.15 Grammar

```
program:
     func_list_opt EOF

func_list_opt:
   /*empty*/
  |func_list

func_list:
  |   func
  |   func_list func

func:
  dtype ID LPAREN argument_list_opt RPAREN compound_statement

argument_list:
      argument
  |    argument_list COMMA argument

argument_list_opt:
    /*empty*/
  | argument_list

argument:
    dtype ID

declaration_list:
|    declaration
|    declaration_list declaration

declaration_list_opt:
    /*empty*/
  |declaration_list

declaration:
      variable_declaration SEMICOLON
  |    array_declaration SEMICOLON

variable_declaration:
      dtype ID_list

ID_list:
ID_Init
|    ID_list COMMA ID_Init

ID_Init:
    ID
| ID ASSIGN constant

array_declaration:
      dtype ID LBRACKET INTCONST RBRACKET

constant:
      INTCONST
  |    CHARCONST
  |    FLOATCONST

dtype:
    VIDEO
  | INT
  | FLOAT
  | CHAR
  | VOID

expression_list_opt:
    /*empty*/
  | expression_list

expression_list:
      expression
  |    expression_list COMMA expression


expression:
      primary_expression
  |    assignment_expression
  |    binary_expression
  |    unary_expression
  |    LPAREN expression RPAREN
  |    functioncall

functioncall:
      ID LPAREN expression_list_opt RPAREN

primary_expression:
        lvalue
  |    constant

lvalue:
      ID
  |    ID LBRACKET expression RBRACKET
  |    this_expr

this_expr:
    THIS LPAREN expression COMMA expression RPAREN

assignment_expression:
      lvalue ASSIGN expression

binary_expression:
  expression PLUS expression
|expression MINUS expression
|expression TIMES expression
|expression DIVIDE expression
|expression MOD expression
```

```
|expression LESSTHAN expression
|expression GREATERTHAN expression
|expression LESSEQUAL expression
|expression GREATEREQUAL expression
|expression EQUAL expression
|expression NOTEQUAL expression
|expression LOGICALAND expression
|expression LOGICALOR expression

unary_expression:
      NEGATION expression
|    MINUS expression %prec UMINUS


statement:
    expression_statement
|    compound_statement
|    selection_statement
|    iteration_statement
|    jump_statement
|    video_statement

expression_statement:
      expression SEMICOLON

compound_statement:
      LBRACE declaration_list_opt statement_list_opt RBRACE

statement_list:
      statement
  |    statement_list statement

statement_list_opt:
      /*empty*/
  |statement_list

selection_statement:
      IF LPAREN expression RPAREN statement else_opt

else_opt:
      %prec NOELSE
  |    ELSE statement

iteration_statement:
      WHILE LPAREN expression RPAREN statement
  | FOR LPAREN expression SEMICOLON expression SEMICOLON expression RPAREN
```

```
statement

jump_statement:
      CONTINUE SEMICOLON
  |   BREAK SEMICOLON
  |   RETURN expression SEMICOLON

video_statement:
  |   LOAD ID FROM STRINGCONST WITH expression AND expression SEMICOLON
  |   STORE ID TO STRINGCONST SEMICOLON
  |   INSERT ID TO ID FROM expression SEMICOLON
  |   DELETE ID FROM expression TO expression SEMICOLON
  |   COPY ID FROM expression TO expression TO ID SEMICOLON
  |   UPDATE ID FROM expression TO expression compound_statement
```

4. Project Plan

4.1 Processes

There were some processes involved in the project. They are planning, Lexer & Syntax Specification, Code Developing and Testing. The four team members meet one or two times a week to check every progress, discuss problems and solve them.

4.1.1 Planning

At first, our team came up an idea that we can create a language to show dynamically the traversal of trees in data structure and associate with animation, which can be showed in html. But later we found there had already been many given instances and the source code to realize the traversal of trees can even be found via internet, which would decrease the practicability of our language creation. And, most of us are not familiar with animation production, which would certainly increase the difficulties of our project. Then, considering there were few succinct and easy-understanding video processing languages, we decided to make a C-like language whose goal is to let programmers manipulate videos easily on their own purpose.

4.1.2 Lexer & Syntax Specification

Since VOPL is a C-like language, most of its lexer and syntax is similar to C. But VOPL has its own special features. It has a new data type video and many English-like video manipulating statements. The specifications are nearly integrated in the first Language Reference Manual. And we revised it while project developed.

4.1.3 Code Developing

There were two stages in code developing. First, lexer, parser, AST definition and main program vopl.ml were developed, and a testing script pr_st.ml was also developed to print out AST in order to prove whether VOPL code can be parsed correctly and form accurate syntax tree. Meanwhile, code generator and semantic checking were developed. And code generator was tested to ensure it could generate correct intermediate code. Then the second stage was to integrate all the sectional programs.

4.1.4 Testing

Testing happens in different stages. As mentioned above, we test lexer and parser. The testing way was to write some test cases and print out AST by calling pr_st.ml in the main program vopl.ml. In this way, we could intuitionally see the form of ast and judge whether our lexer and parser had some problem. The script of pr_st.ml is as that in 8.2. We also tested the correctness of semantic analyzer by testing some cases to see whether the incorrectness can be reported. And we also tested our entire program after integrating all the sectional parts. Test cases and sample program can be seen in 8.3 and 2.3.

4.2 Project Timeline

S#	Milestones	Deliverables	Milestone Date
01	Completion of Project plan	Project Proposal	2008-9-24
02	Completion of Language Reference Manual	Language Reference Manual	2008-10-10
03	Completion of Coding + Unit Testing	Code, Unit test results	2008-11-23
04	Completion of Integrated testing	Code, Integrated test results	2008-12-1
05	Completion of Presentation	Presentation of the language	2008-12-6
06	Completion of Final Report	Final Project Report	2008-12-15

4.3 Roles and Responsibilities

- Baolin Shao: Code Generator
- Xuyang Shi: Semantic Analyzer, Semantic Test
- Huning Dai: AST, Matlab Library, Integrated Test
- Jia Li: Lexer, Parser, Syntax Test, Integrated Test, Documents

4.4 Software Development Environment

4.4.1 IDE

The team wrote the compiler for VOPL through IDE eclipse by utilizing Ocaml plug-in. Working with Ocaml plug-in for eclipse has advantages because it has many convenient features such as source editor for modules (ml files), interfaces (mli files), parsers (mly files) and lexers (mll files), syntax coloring, automatic indentation while typing, integrated debugger and so on.

4.4.2 Version Control System

The team used SVN connecting to google code project hosting server to which we upload our source code in order to allow team members work concurrently and collaborate on the same project. This way makes it easy for every member in the team to check the latest code modified by others and commit his own code.

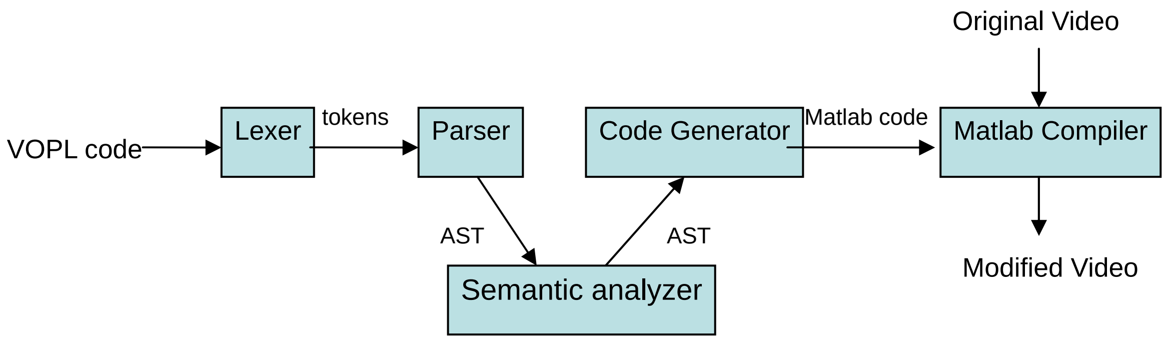
4.5 Project Log

S#	Tasks	Deliverables	Date
01	Completion of Project plan	Project Proposal	2008-9-24
02	Completion of Language Reference Manual	Language Reference Manual	2008-10-10
03	Completion of Coding + Unit	Lexer	2008-11-10

	Testing	Parser, define AST	2008-11-23
		Code Generation	2008-11-29
		Semantic Checking	2008-12-15
04	Completion of Integrated test	Code, Integrated test results	2008-12-17
05	Completion of Final Report	Final Project Report	2008-12-18
06	Completion of Presentation	Presentation of the language	2008-12-19

5. Architectural Design

VOPL code is read by the Lexer and converted into tokens. These tokens are sent to the parser which generates an abstract syntax tree based on the VOPL grammar. The semantic analyzer traversal the abstract syntax tree, and then intermediate matlab code is then generated as a *.m file by code generator. When matlab compiler executes this *.m file, an original video is loaded into the compiler and the output is the modified video which is stored to a file appointed by VOPL code.



Baolin Shao: Code Generator
Xuyang shi: Semantic analyzer
Huning Dai: Matlab library
Jia Li: Lexer, Parser

6. Test Plan

6.1 Syntax Test

We wrote a testing script called pr_st.ml to display the output of the abstract syntax tree that was generated by the parser. The syntax test script is showed in 8.3.1

6.2 Semantic Analyzer Test

We wrote test cases that cover possible mistakes to the greatest extent and see whether semantic mistakes will be reported while compiling.

6.3 Integrated Test

We wrote some program samples in VOPL and compile them with our compiler. And then check whether the output is what we expect or whether any on purpose mistake can be reported.

6.4 Test Cases

Test cases are chosen according to the syntaxes we defined before. Test cases used in syntax test and semantic analyzer are showed in 8.3.

6.5 Responsibility

- Syntax Test: Jia Li
- Semantic Test: Xuyang shi
- Integrated Test: Jia Li, Huning Dai

7. Lessons Learned

7.1 Baolin Shao

It is 3 months since the first time we sit together and proposed for developing a new programming language to ease video editing application development. After this 3-month project, we finally, for the first time in our life, accomplish a language compiler from scratch, although it is small. During this project, we finished most tasks we planed, but we still made some tradeoff to handle problems we did not expect at the beginning. Therefore, I would like to share some of my experience learned in this project.

As Prof. Stephen Edwards “warned” us in the class that we must be careful with dealing with the other three “enemies” in the group, the most valuable experience for me, a project leader, is how to fill in the knowledge gap among different members. In our group, we have various backgrounds (compiler for them is a magic). So, in our first meeting, I gave them a presentation about what a compiler looks like, what we need to do and how to do it. Even so, I found that I spent most of time in making sure everyone really understands what his or her own task.

Another valuable lesson is that designing a language is far more important and complicated than implementing it. Before taking this course, I was involved in some compiler-like project. But in those projects what I did was nothing but programming. This project is different that we have to decide where to go and how far we will be. This requires much language design experience, which we did not have at that time. So, there is no surprise that we made some designs that was found to be not practical later in implementation. For example, initially, our language has scopes, similar to C’s scoping rules. But when we were going to generate Matlab code, we found that Matlab does not have scoping rules. It would beyond our capability to use a non-scoping language to support a scoping one, because it requires sophisticated variable renaming. Therefore, we had to make a compromise by giving up scoping rules, which makes our language less beautiful.

Besides, I learnt Ocaml and improved my engineering skills. But, at the same time, I also find that there is something that I should have done better. For example, it is my fault to make the project lasting too long, although sometimes we have to compromise between the coming finals and the project. Second, I do not think I discover all of my group members’ potential in this project. As I stated before, most of the time, they did what I wanted. I should encourage them more to actively contribute to this project.

7.2 Xuyang Shi

Doing this project deepens my understanding in how complier work and the structure of all kinds of language. As for me, I found communication between team members is extremely important. The weekly meeting we hold helped us through almost all problems.

I was in charge of semantic analysis in our project, and I found Ocaml is really good at doing this kind of job.

My team used SVN with IDE in Eclipse at the very beginning of the project. Though it seems trivial to update and commit frequently, it did prevent lots of disasters.

Advices for future teams:
1. Should pay more time on the "Dragon Book".
2. Try to start the project ASAP.

7.3 Huning Dai

It is my very first time building a language from nothing, and using Ocaml, which I had never heard of before. I have to say it was a great time that I and three of my teammates spent together.

The two-third of all the time that we spent in discussing and deciding the whole aspect of VOPL is quite worthy because after that everything went smoothly.

Using the IDE on Eclipse along with SVN was a wise idea. If we didn't use them, we must be died by now.

Planning things ahead is quite useful. After thanksgiving, every team member was suffering from massive attacks from all subjects. We were lucky enough to have most of the work done before that time.

And what's more, I learned how to use Ocaml, how to process video using matlab...and so many more.

Advices for future teams:

1. Spend more time in designing the language before coding. Try not to waste time on redesigning the language.
2. An old phrase: Use SVN or Die.
3. Plan thing ahead................................or Die.

7.4 Jia Li

From the project I learnt that it is worth-while to spend much time on the plan of the project at the beginning. By discussing and figuring out the whole structure and the way to implement every step in the project, such as the special features in the language and the intermediate code, all the team members can get a microscopy of the project, which can be helpful for the future development.

Once the syntax is confirmed, we had better not to make big changes to it while developing the project, otherwise it will cause a lot of extra labor since the parser will generate new AST which will also require changes to the back-end of the compiler. Thus, testing the parser and trying to cover all the syntaxes is important. Making sure the correctness of every segmental part of the project can make the future development smooth.

And SVN is convenient for group developing. It is easy for every member in the team update and commit code when they working separately by using SVN. It is a powerful tool for team collaborating.

Advice for future teams:
Start as early as possible. Communicate and discuss any unsure problems with teammates as much as possible. Spend more time on planning and testing.

8. Appendix

8.1 Source Code

scanner.mll

```
{ open Parser }

let string_set=['a'-'z' 'A'-'Z']| ['0'-'9']| '.'|'/'|'\\'|'~'|'-'
```

```
rule token=
parse
[' ' '\t'] { token lexbuf }
| ("\r\n"|'\r'|'\n') { token lexbuf }
| "/*" { comments lexbuf }
| '(' { LPAREN }
| ')' { RPAREN }
| '[' { LBRACKET }
| ']' { RBRACKET }
| '{' { LBRACE }
| '}' { RBRACE }
| ',' { COMMA }
| ';' { SEMICOLON }
| '+' { PLUS }
| '-' { MINUS }
| '*' { TIMES }
| '/' { DIVIDE }
| '%' { MOD }
| '!' { NEGATION }
| '=' { ASSIGN }
| '<' { LESSTHAN }
| '>' { GREATERTHAN }
| "==" { EQUAL }
| "!=" { NOTEQUAL }
| "<=" { LESSEQUAL }
| ">=" { GREATEREQUAL }
| "&&" { LOGICALAND }
| "||" { LOGICALOR }

| "if" { IF }
| "else" { ELSE }
| "for" { FOR }
| "while" { WHILE }
| "break" { BREAK }
| "continue" { CONTINUE }
| "return" { RETURN }
| "void" { VOID }
| "video" { VIDEO }
| "int" { INT }
| "float" { FLOAT }
| "char" { CHAR }
| "this" { THIS }
| "load" { LOAD }
| "from" { FROM }
| "with" { WITH }
| "and" { AND }
| "store" { STORE }
| "insert" { INSERT }
| "delete" { DELETE }
| "copy" { COPY }
| "update" { UPDATE }
| "to" { TO }

| eof { EOF }
| ['0'-'9']+ as intconst { INTCONST(int_of_string intconst) }
| "'" _ "'" as charconst { CHARCONST(String.get charconst 1) }
| ['0'-'9']+ '.' ['0'-'9']* as floatconst { FLOATCONST(float_of_string floatconst) }
| ['a'-'z''A'-'Z']['a'-'z''A'-'Z''0'-'9''_']* as id { ID(id) }
| '"' string_set* '"' as stringconst { STRINGCONST(stringconst) }

and comments = parse
   | "*/"    {token lexbuf }
   | ("\r\n"|'\n'|'\r') { comments lexbuf }
   | eof { EOF }
   | _ { comments lexbuf }
```

parser.mly

```
%{ open Ast %}

%token VOID INT FLOAT CHAR VIDEO
%token PLUS MINUS TIMES DIVIDE MOD EOF
%token LOGICALAND LOGICALOR
%token EQUAL NOTEQUAL
%token LESSTHAN GREATERTHAN LESSEQUAL GREATEREQUAL
%token NEGATION
%token COMMA SEMICOLON ASSIGN
%token IF ELSE WHILE FOR CONTINUE BREAK RETURN
%token LOAD FROM WITH AND STORE TO INSERT DELETE COPY UPDATE
THIS
%token LPAREN RPAREN LBRACE RBRACE LBRACKET RBRACKET
%token <string> ID
%token <int> INTCONST
%token <char> CHARCONST
%token <float> FLOATCONST
%token <string> STRINGCONST

%nonassoc NOELSE
%nonassoc ELSE
%left LOGICALOR
%left LOGICALAND
```

```
%left OR
%left AND
%left EQUAL NOTEQUAL
%left LESSTHAN GREATERTHAN LESSEQUAL GREATEREQUAL
%left COMMA
%right ASSIGN
%left PLUS MINUS
%left TIMES DIVIDE MOD
%nonassoc UMINUS NEGATION

%start program
%type <Ast.program> program

%%


program:
    func_list_opt EOF {Pro($1)}

func_list_opt:
   /*empty*/           {[]}
  |func_list     {List.rev $1}

func_list:
  |    func              {[$1]}
  |    func_list func    {$2::$1}

func:
  dtype ID LPAREN argument_list_opt RPAREN compound_statement
{Func($1,$2,$4,$6)}

argument_list:
    argument    {[$1]}
  |    argument_list COMMA argument    {$3::$1}

argument_list_opt:
   /*empty*/      {[]}
  | argument_list    {List.rev $1}

argument:
   dtype ID    {Arg($1,$2)}

declaration_list:
|    declaration      {[$1]}
|    declaration_list declaration    {$2::$1}
```

```
declaration_list_opt:
   /*empty*/    {[]}
  |declaration_list    {List.rev $1}

declaration:
     variable_declaration SEMICOLON    {VarDecl($1)}
  |    array_declaration SEMICOLON      {ArraDecl($1)}

variable_declaration:
     dtype ID_list    { Ddecl ($1,$2)}

ID_list:
ID_Init      {[$1]}
|    ID_list COMMA ID_Init    {$3::$1}

ID_Init:
   ID    { Init($1,None)}
| ID ASSIGN constant { Init($1,$3)}

array_declaration:
     dtype ID LBRACKET INTCONST RBRACKET    {Adecl($1,$2,$4)}

constant:
     INTCONST       {Consopt(Intc($1))}
  |    CHARCONST    {Consopt(Charc($1))}
  |    FLOATCONST {Consopt(Floatc($1))}

dtype:
     VIDEO       {Video}
  | INT         {Int}
  |    FLOAT       {Float}
  |    CHAR         {Char}
  | VOID {Void}

expression_list_opt:
   /*empty*/       {[]}
  | expression_list      {List.rev $1}

expression_list:
     expression       {[$1]}
  |    expression_list COMMA expression      {$3::$1}


expression:
     primary_expression      {$1}
  |    assignment_expression    {$1}
```

```
 |    binary_expression    {$1}
 |    unary_expression      {$1}
 |    LPAREN expression RPAREN    {$2}
 |    functioncall    {$1}


functioncall:
     ID LPAREN expression_list_opt RPAREN { Funcall($1,$3)}

primary_expression:
       lvalue    { Lval($1)}
 |    constant    {Const($1)}

lvalue:
     ID    {Variable($1)}
 |    ID LBRACKET expression RBRACKET      {Array($1,$3)}
 |    this_expr            { Thisl($1)}

this_expr:
   THIS LPAREN expression COMMA expression RPAREN    {This($3,$5)}

assignment_expression:
     lvalue ASSIGN expression    { Assign ($1,$3)}

binary_expression:
  expression PLUS expression    { Binop($1,Plus,$3)}
|expression MINUS expression    { Binop($1,Minus,$3)}
|expression TIMES expression    { Binop($1,Times,$3)}
|expression DIVIDE expression    { Binop($1,Divide,$3)}
|expression MOD expression    { Binop($1,Mod,$3)}
|expression LESSTHAN expression    { Binop($1,Lessthan,$3)}
|expression GREATERTHAN expression    { Binop($1,Greaterthan,$3)}
|expression LESSEQUAL expression    { Binop($1,Lessequal,$3)}
|expression GREATEREQUAL expression    { Binop($1,Greaterequal,$3)}
|expression EQUAL expression    { Binop($1,Equal,$3)}
|expression NOTEQUAL expression    { Binop($1,Notequal,$3)}
  |expression LOGICALAND expression    { Binop($1,Logicaland,$3)}
  |expression LOGICALOR expression    { Binop($1,Logicalor,$3)}

unary_expression:
     NEGATION expression    {Unaryop(Negation,$2)}
|    MINUS expression %prec UMINUS {Unaryop(Ninus,$2)}

statement:
    expression_statement    {$1}
|    compound_statement      {$1}
|    selection_statement    {$1}
|    iteration_statement    {$1}
|    jump_statement             {$1}
|    video_statement            {$1}

expression_statement:
     expression SEMICOLON    { Expression($1)}

compound_statement:
     LBRACE declaration_list_opt statement_list_opt RBRACE    { Compound($2,$3)}

statement_list:
     statement      {[$1]}
  |    statement_list statement    {$2::$1}

statement_list_opt:
     /*empty*/    {[]}
  |statement_list    {List.rev $1}

selection_statement:
     IF LPAREN expression RPAREN statement else_opt    {If($3,$5,$6)}

else_opt:
     %prec NOELSE    {Compound([],[])}
  |    ELSE statement    {$2}

iteration_statement:
     WHILE LPAREN expression RPAREN statement    {While($3,$5)}
  | FOR LPAREN expression SEMICOLON expression SEMICOLON expression
RPAREN statement    {For($3,$5,$7,$9)}

jump_statement:
     CONTINUE SEMICOLON    {Continue}
  |    BREAK SEMICOLON        {Break}
  |    RETURN expression SEMICOLON    {Return($2)}

video_statement:
  | LOAD ID FROM STRINGCONST WITH expression AND expression SEMICOLON
{Load($2,$4,$6,$8)}
  |     STORE ID TO STRINGCONST SEMICOLON                    {Store($2,$4)}
  |    INSERT ID TO ID FROM expression SEMICOLON
{Insert($2,$4,$6)}
  |    DELETE ID FROM expression TO expression SEMICOLON
{Delete($2,$4,$6)}
  |    COPY ID FROM expression TO expression TO ID SEMICOLON
{Copy($2,$4,$6,$8)}
```

```
 |    UPDATE ID FROM expression TO expression compound_statement
{Update($2,$4,$6,$7)}
```

ast.mli

```
type operator=
  Plus
|Minus
|Times
|Divide
|Mod
|Lessthan
|Greaterthan
|Lessequal
|Greaterequal
|Equal
|Notequal
|Logicaland
|Logicalor

type unary_operator=
  Negation
|Ninus

type id=string

type expression=
  Lval of lvalue
|Const of constant_option
|Assign of lvalue*expression
|Binop of expression*operator*expression
|Unaryop of unary_operator*expression
|Comma of expression*expression
|This of expression*expression
|Funcall of id*expression list

and lvalue=
  Variable of id
|Array of id*expression
|Thisl of expression



and constant=
  Intc of int
```

```
|Charc of char
|Floatc of float

and constant_option=
  None
|Consopt of constant


type dtype=
  Int
|Char
|Float
|Video
|Void

type i_init=
Init of id*constant_option


type declaration=
  VarDecl of variable_declaration
|ArraDecl of array_declaration

and variable_declaration=
  Ddecl of dtype*(i_init list)

and array_declaration=
  Adecl of dtype*id*int

type argument=
Arg of dtype*id

type statement=
  Expression of expression
|Compound of declaration list*statement list
|If of expression*statement*statement
|While of expression*statement
|For of    expression*expression*expression*statement
|Continue
|Break
|Return of expression
|Load of id*string*expression*expression
|Store of id*string
|Insert of id*id*expression
|Delete of id*expression*expression
|Copy of id*expression*expression*id
```

```
|Update of id*expression*expression*statement

type func=
  Func of dtype*id*argument list*statement

type program=
  Pro of func list
```

semantic_analysis.ml

```
open Ast

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

type sub = {
      is_loop : bool;
      current_type : Ast.dtype;
      is_update : bool;
      current_function_type: Ast.dtype;
      is_main:bool;
}
let type_to_string t =
      match t with
            |Int -> "int"
            |Char -> "char"
            |Float -> "float"
            |Video -> "video"
            |Void -> "void"

let get_type env =
      let (sub_env,_,_) = env in sub_env.current_type

let get_function env =
      let (sub_env,_,_) = env in sub_env.current_function_type

let get_loop env =
      let (sub_env,_,_) = env in sub_env.is_loop

let set_type env t =
      let (sub_env,v,f) = env in
      let sub_env={sub_env with current_type = t} in
      let env=(sub_env,v,f) in env
let set_loop env t =
        let (sub_env,v,f) = env in
        let sub_env={sub_env with is_loop = t} in
        let env=(sub_env,v,f) in env
let set_update env t =
        let (sub_env,v,f) = env in
        let sub_env={sub_env with is_update = t} in
        let env=(sub_env,v,f) in env


let check_constopt c=
          let t = (match c with
                Intc(_)-> Int
                |Floatc(_)-> Float
                |Charc(_)->Char ) in t



let check_const env cons_opt=
        let (sub_env, var_map, func_map) = env in
        match cons_opt with
                None -> env
                |Consopt(c) ->
                        let t = check_constopt c in
                        let sub_env = {sub_env with current_type=t} in
                        let env = (sub_env, var_map, func_map) in
        env

let check_i_init env init =
        match init with
                | Init (vid,cons_op) ->
                             let (s,var_map,f) = env in
                            let t = get_type env in
                            let p=StringMap.mem vid var_map in
                            let _=if(p=true) then let err=    "This variable "^vid^" has been defined
before\n"in

                                 print_string err    in
                            let env=check_const env cons_op in
                            let ctype = get_type env in
                            let _=if(ctype!=t) then let err = "This variable "^vid^" has type "
                            ^(type_to_string t)^"but here used with"^(type_to_string ctype)^"\n" in

                            print_string err       in
                            let var_map = StringMap.add vid t var_map in
                            let env=(s,var_map,f) in env
let check_vardecl env dec_list=
    match dec_list with
                Ddecl(dectype,ilist) ->
        let (sub_env,var_map,func_map) = env in
        let sub_env = {sub_env with current_type=dectype} in
        let env = (sub_env,var_map,func_map) in
        let env = List.fold_left check_i_init env ilist in
        env

let check_arrdecl env dec_list=
        match dec_list with
                | Adecl(dectype,aid,length) ->
                          let (sub_env,var_map,func_map) = env in
                         let sub_env = {sub_env with current_type=dectype} in
                         let arr_exist = StringMap.mem aid var_map in
                         let _=if(arr_exist==true) then let err =    "This array "^aid^" has been
defined before\n" in

                             print_string err in
                         let _=if(length<=0) then let err="This array "^aid^"'s length should be
greater than 0\n" in

                            print_string err      in
                          let var_map = StringMap.add aid dectype var_map in
                         let env =(sub_env,var_map,func_map) in env

let check_declaration env dec_list=
        match dec_list with
                |VarDecl(var_decl) -> let env = check_vardecl env var_decl in env
                |ArraDecl(arry_decl) -> let env = check_arrdecl env arry_decl in env



let rec check_expression env expr =
        match expr with
                Lval (lval)-> let env = check_lval env lval in env
                |Const (c_option)-> let env = check_const env c_option in env
                |Assign (lval,expr)-> let env = check_assign env lval expr in env
                |Binop (expr1,oper,expr2)-> let env = check_binop env expr1 oper expr2 in env
                |Unaryop (uoper,expr)-> let env = check_unaryop env uoper expr in env
                |Comma (expr1,expr2)-> let env = check_comma env expr1 expr2 in env
                |This (expr1,expr2)-> let env = check_this env expr1 expr2 in env
                |Funcall (fid,expr_list)-> let env = check_funcall env fid expr_list in
        env
```

```
and check_unaryop env uoper expr=

        let env = check_expression env expr in
        let t =get_type env in
        let _=if(t=Char) then let err= "unary operation cannot be applied to char type\n" in
print_string err in
        let env = set_type env t in
        env

and check_comma env expr1 expr2=

        let env = check_expression env expr1 in
        let t1 = get_type env in
        let env = check_expression env expr2 in
        let t2 = get_type env in
        let _=if(t1!=t2) then let err = "two expression in a comma expression should have
the same type, but here used with "
          ^(type_to_string t1)^" and "^(type_to_string t2) in
        print_string err in
        let (sub_env,var_map,func_map) = env in
        let sub_env = {sub_env with current_type=t1} in
        let env = (sub_env,var_map,func_map) in
        env

and check_this env expr1 expr2=
        let (sub_env,var_map,func_map) = env in
        let env = check_expression env expr1 in
        let t1=get_type env in
        let env = check_expression env expr2 in
        let t2=get_type env in
        let _=if((t1!=Ast.Int)||(t2!=Ast.Int)) then print_string "This operation requires integer
type of operands\n";      in
        let _=if(sub_env.is_update==false) then print_string " \"this\" operator should be
used in update statement\n";      in
        env

and match_argu env arg expr =
        let env=check_expression env expr in
        let (sub_env,var_map,func_map) = env in
        let t1=sub_env.current_type in
        match arg with
                | Arg (t2,_) ->
                    let _=if(t1!=t2) then print_string "Function's parameters are not the same\n";
in
                    env
```

```
and check_funcall env fid expr_list=
	let _ = if(fid ="main") then print_string "function main cannot be called\n" in
	let (sub_env,var_map,func_map) = env in
	let fun_exist = StringMap.mem fid func_map in
	let _=if(fun_exist==false) then let err = "Function "^fid^" not defined\n" in
print_string err	in
	let arg=StringMap.find fid func_map in
	(match arg with
		| hd::tl ->

			let env=List.fold_left2 match_argu env tl expr_list in
			(match hd with
			| Arg(t,_)->
				let env = set_type env t in	env)
		|[]->env
	)
and check_binop env expr1 oper expr2=
	let (sub_env,var_map,func_map) = env in
	let env = check_expression env expr1 in
	let t1 = get_type env in
	let env = check_expression env expr2 in
	let t2 = get_type env in
	let _=if(t1!=t2) then let err="Binary operation requires same type of operands, but
here used with " ^(type_to_string t1)^" and "^(type_to_string t2)^"\n" in

	print_string err	in
	let _=if((t1==Ast.Float)&&(oper==Ast.Mod)) then print_string "Floating point
MOD operation requires integer operands\n";	in
	let sub_env = {sub_env with current_type=t1} in
	let env = (sub_env,var_map,func_map) in
	env

and check_assign env lval expr=
	let (sub_env, var_map, func_map) = env in
	let env = check_lval env lval in
	let t1 = get_type env in
	let env = check_expression env expr in
	let t2 = get_type env in
	let _=if(t1!=t2) then let err="assignment requires same type of operands, but here
used with"^(type_to_string t1)^" and "^(type_to_string t1)^"\n" in print_string err	in
	let sub_env = {sub_env with current_type=t1} in
	let env = (sub_env, var_map, func_map) in
	env


and check_array env aid expr=
        let (sub_env,var_map,func_map) = env in
        let env = check_expression env expr in
        let t = get_type env in
        let _= if(t!=Int) then print_string "array index should be used with type int\n";      in
        let t = StringMap.mem aid var_map in
        let _= if(t!=true) then let err="array "^aid^" not declared\n" in print_string err    in
        let vartype = StringMap.find aid var_map in
        let env = set_type env vartype in env


and check_lval env lval =
        let (_,var_map,_)=env in
        match lval with
                |Array (id,expr)-> let env = check_array env id expr in env
                |Thisl (expr)-> let env = check_expression env expr in env
                 |Variable(var) -> let t = StringMap.mem var var_map in
                                                let _= if(t!=true) then let err ="variable "^var^" not
declared" in print_string err      in
                                                let vartype = StringMap.find var var_map in
                                                let env = set_type env vartype in env


let rec check_statement env stmts =
        match stmts with
        |Expression(expr) -> let env = check_expression env expr in env
        |Compound (decl,stmt) -> let env = check_compound env decl stmt in env

        (* IF: check 3 conditions:*)
        (* (1) if expression is right*)
        (* (2) if expression's type is int, we don't have bool in our language*)
        (* (3) check the two statements if right *)
        |If (expr,stmt1,stmt2) -> (match expr with
        |Binop(_,op,_) -> (match op with

        |Lessthan

        |Greaterthan

        |Lessequal

        |Greaterequal

        |Equal

        |Notequal
        |Logicaland

        |Logicalor -> let env = check_expression env expr in

                                let (sub_env,var_map,func_map) = env in

                                let _ = if(sub_env.current_type != Int) then

                                let err="if condition requires type of int, but here used with
"^(type_to_string sub_env.current_type)^"\n" in print_string err      in

                                let env = check_statement env stmt1 in

                                let env = check_statement env stmt2 in env

        |_-> let _= print_string "if condition requires logical operation";

                in env )
         |Unaryop(op,_) -> (match op with

        |Negation ->let env = check_expression env expr in

                              let (sub_env,var_map,func_map) = env in

                              let _ = if(sub_env.current_type != Int) then

                              let err="if condition requires type of int, but here used with
"^(type_to_string sub_env.current_type)^"\n" in print_string err      in

                              let env = check_statement env stmt1 in

                              let env = check_statement env stmt2 in env

        |Ninus -> let _= print_string "if condition requires logical operation";

                    in env )
          |_-> let _= print_string "if condition requires logical operation";

                   in env )

        (* While: check 3 conditions *)
        (* (1) chekc expression *)
        (* (2) chekc expression's type is int*)
        (* (3) update env as is_loop = true and check the body of loop as compound
statement *)
```

```
    |While (expr,stmt) ->(match expr with
                                    |Binop(_,op,_) -> (match op with

                                        |Lessthan

                                        |Greaterthan

                                        |Lessequal

                                        |Greaterequal

                                        |Equal

                                        |Notequal

                                        |Logicaland

                                        |Logicalor ->

                                     let env = check_expression env expr in

                                     let (sub_env,var_map,func_map) = env in

                                     let sub_env={sub_env with is_loop = true} in

                                     let _ = if(sub_env.current_type != Int) then

                                        let err="while condition requires type of int,

                                            but here used with "^(type_to_string
sub_env.current_type)^"\n" in print_string err      in

                                         let env = check_statement env stmt in

                                         let env = set_loop env false in env

                                       |_-> let _= print_string "while condition requires
logical operation";

                                        in env )
        |Unaryop(op,_) -> (match op with

                                      |Negation ->let env = check_expression env expr in

                                          let (sub_env,var_map,func_map) = env in
                                        let sub_env={sub_env with is_loop = true} in

                                        let _ = if(sub_env.current_type != Int) then

                                                    let err="while condition requires type of int,
but here used with "^(type_to_string sub_env.current_type)^"\n" in print_string err      in

                                                      let env = check_statement env stmt in

                                                      let env = set_loop env false in env

                                     |Ninus -> let _= print_string "while condition requires
logical operation";

                                            in env )
      |_-> let _= print_string "while condition requires logical operation";

              in env )
   (* For: check conditions:*)
   (* (1) expr1 must be ass expr with int*)
   (* (2) expr2 must be binary expr with int*)
   (* (3) expr3 must be assg expr with int*)
   (* (4) update env's is_loop to be true and check statement*)

   |For (expr1,expr2,expr3,stmt) -> let env= (match expr1 with

      | Assign(_) -> let env =check_expression env expr1 in env

      | _ -> let _ = print_string "the first expression in for loop requires to be
assignment";

                in env ) in
                let t = get_type env in
                let _ = if(t != Int) then

                  print_string "for loop's first expression requires type of int, but here
                  used with conflict";      in


               let env= (match expr2 with

      | Binop(_,_,_) -> let env =check_expression env expr2 in env

      | _ -> let _ = print_string "the second expression in for loop requires to be binary
expression";
```

```
                    in env ) in
                    let t = get_type env in
                    let _ = if(t != Int) then

                        print_string "for loop's second expression requires type of int, but here
                        used with conflict";      in



                      let env= (match expr3 with

            | Assign(_) -> let env =check_expression env expr3 in env

            | _ -> let _ = print_string "the third expression in for loop requires to be
assignment";

                     in env ) in
                     let t = get_type env in
                     let _ = if(t != Int) then

                         print_string "for loop's third expression requires type of int, but here
                         used with conflict";      in
                     let env = set_loop env true in
                     let env = check_statement env stmt in
                     let env = set_loop env false in env
           |Continue -> let t = get_loop env in
                              let _ = if(t != true) then print_string "continue statement must be
used in loop";      in env
           |Break ->let t = get_loop env in
                        let _ = if(t != true) then print_string "break statement must be used in
loop";      in env

     (*check if return has the same type has funciton *)
        |Return (expr) -> let env = check_expression env expr in
                                    let t1 = get_type env in
                                 let t2 = get_function env in
                                 let _ = if(t1 != t2) then print_string "return statement must
return the same type with its function,                                   but here
used with conflict"; in env
     (*Load:*)
     (*(1) if id is a declared variable*)
     (*(2) expr1 & expr2 must be int*)
     |Load (var,_,expr1,expr2) -> let (sub_env,var_map,func_map) = env in
                                          let t = StringMap.mem var var_map in
                                      let _ = if(t!=true) then let err= ("variable
"^(var)^"is not declared") in print_string err in
```

```
                                                  let t = StringMap.find var var_map in
                                                  let _ = if(t!=Video) then print_string "variable in
load must be video, but here used with conflict";    in
                                                   let env = check_expression env expr1 in
                                                    let t = get_type env in
                                                   let _ = if(t != Int) then

print_string " expression used in load requires type of int, but here used with conflict";
in
                                                   let env = check_expression env expr2 in
                                                   let t = get_type env in
                                                   let _ = if(t != Int) then

print_string " expression used in load requires type of int, but here used with conflict";
in env


    |Store (var,_) -> let (sub_env,var_map,func_map) = env in
                                          let t = StringMap.mem var var_map in
                                      let _ = if(t!=true) then print_string    ("variable "^(var)^"is not
declared");      in
                                      let t = StringMap.find var var_map in
                                      let _ = if(t!=Video) then print_string "variable in store must be
video, but here used with conflict";    in env

    |Insert (var1,var2,expr) -> let (sub_env,var_map,func_map) = env in
                                                  let t = StringMap.mem var1 var_map in
                                                  let _ = if(t!=true) then print_string    ("variable
"^(var1)^"is not declared");      in
                                                  let t = StringMap.find var1 var_map in
                                                  let _ = if(t!=Video) then print_string "variable in
Insert must be video, but here used with conflict";      in
                                                  let t = StringMap.mem var2 var_map in
                                                  let _ = if(t!=true) then print_string    ("variable
"^(var2)^"is not declared");      in
                                                  let t = StringMap.find var2 var_map in
                                                  let _ = if(t!=Video) then print_string "variable in
Insert must be video, but here used with conflict";    in
                                                  let env = check_expression env expr in
                                                  let t = get_type env in
                                                  let _ = if(t != Int) then

print_string " expression used in insert requires type of int, but here used with conflict";
in env
```

```
        |Delete (var,expr1,expr2) -> let (sub_env,var_map,func_map) = env in
                                        let t = StringMap.mem var var_map in
                                        let _ = if(t!=true) then print_string    ("variable
"^(var)^"is not declared");     in
                                        let t = StringMap.find var var_map in
                                        let _ = if(t!=Video) then print_string "variable in
Delete must be video, but here used with conflict";    in
                                        let env = check_expression env expr1 in
                                        let t = get_type env in
                                            let _ = if(t != Int) then

print_string " expression used in Delete requires type of int, but here used with conflict";
in
                                        let env = check_expression env expr2 in
                                        let t = get_type env in
                                        let _ = if(t != Int) then

print_string " expression used in Delete requires type of int, but here used with conflict";
in env


        |Copy (var1,expr1,expr2,var2) -> let (sub_env,var_map,func_map) = env in
                                            let t = StringMap.mem var1 var_map in
                                            let _ = if(t!=true) then print_string
("variable "^(var1)^"is not declared");      in
                                            let t = StringMap.find var1 var_map in
                                            let _ = if(t!=Video) then print_string
"variable in Copy must be video, but here used with conflict";      in
                                            let t = StringMap.mem var2 var_map in
                                            let _ = if(t!=true) then print_string
("variable "^(var2)^"is not declared");      in
                                             let t = StringMap.find var2 var_map in
                                            let _ = if(t!=Video) then print_string
"variable in Copy must be video, but here used with conflict";      in
                                            let env = check_expression env expr1 in
                                            let t = get_type env in
                                            let _ = if(t != Int) then

print_string " expression used in Copy requires type of int, but here used with conflict";
in
                                            let env = check_expression env expr2 in
                                            let t = get_type env in
                                                                        let _ = if(t !=
Int) then

print_string " expression used in Copy requires type of int, but here used with conflict";
```

```
in env


        |Update (var,expr1,expr2,stmt) -> let (sub_env,var_map,func_map) = env in
                                                      let env = set_update env true in
                                                     let t = StringMap.mem var var_map in
                                                      let _ = if(t!=true) then print_string
("variable "^(var)^"is not declared");      in
                                                       let t = StringMap.find var var_map in
                                                       let _ = if(t!=Video) then print_string
"variable in Delete must be video, but here used with conflict";      in
                                                       let env = check_expression env expr1 in
                                                      let t = get_type env in
                                                      let _ = if(t != Int) then

print_string " expression used in Delete requires type of int, but here used with conflict";
in
                                                       let env = check_expression env expr2 in
                                                       let t = get_type env in
                                                       let _ = if(t != Int) then

print_string " expression used in Delete requires type of int, but here used with conflict";
in
                                                       let env = check_statement env stmt in
                                                       let env = set_update env false in env



and check_compound env dec_list stat_list=
        let env = List.fold_left check_declaration env dec_list in
        let env = List.fold_left check_statement env stat_list in
        env

let check_argument env arg =
        match arg with
                | Arg(t,var) -> let _ =if(t==Void) then print_string ("function cannot have type
\"void\" as its arguement");      in
                                       let (s,var_map,f) = env in
                                       let p=StringMap.mem var var_map in
                                       let _=if(p==true) then let err="This variable "^var^" has
been defined before" in print_string err      in
                                      let var_map = StringMap.add var t var_map in
                                      let env=(s,var_map,f) in env

let check_function env function_list =
        let (sub_env,var_map,func_map) = env in
        let var_map = StringMap.empty in
```

```
    let env=(sub_env,var_map,func_map) in
    match function_list with
        | Func(t, f_name, args, stmts) ->
                let _ = if(t=Video) then print_string ("function "^(f_name)^"cannot
return video type") in
                let env = List.fold_left check_argument env args in
                let (sub_env,var_map,func_map) = env in
                let tt = StringMap.mem f_name func_map in
                let _=if(tt=true) then print_string ("function "^(f_name)^" has been
defined") in
                let args=Arg(t,"null")::args in
                let func_map = StringMap.add f_name args func_map in
                let sub_env = {sub_env with current_function_type = t} in

                    if(f_name="main") then let sub_env = {sub_env with
                    is_main=true} in

                let env = (sub_env,var_map,func_map) in

                let env = check_statement env stmts in env
                    else let env = (sub_env,var_map,func_map) in

                   let env = check_statement env stmts in env

let check_program env p=
    match p with
        | Pro(prog) ->
  let env = List.fold_left check_function env prog in
    let (sub_env,_,_) = env in
    let _ =if(sub_env.is_main=false) then    print_string ("there is no main function in
the program") in
    p

let translate prog =
    (* Initialize the semantic analysis environment and call main program
check_program *)
    let fun_map = StringMap.empty in
    let var_map = StringMap.empty in
    let sub_env = {
        is_loop = false;
        current_type = Ast.Int;
        is_update = false;
        current_function_type = Void;
        is_main=false;
        }in
    let env = (sub_env,var_map,fun_map) in
```

```
        let p=check_program env prog in p
```

codegen.ml

```
open Ast
(*change a file name from "abc" to 'abc' *)
let remove_qoute str =
        let len = String.length str in
        let _= String.set str 0 '\'' in
        let _= String.set str (len-1) '\'' in
        str

let build_binary_operator func_file op =
        match op with
                |Plus -> let _= output_string func_file " + " in func_file
                |Minus-> let _= output_string func_file " - " in func_file
                |Times-> let _= output_string func_file " * " in func_file
                |Divide-> let _= output_string func_file " / " in func_file
                |Mod->    func_file (*because mod in matlab is a funciton, generate the code
seepartely in build_expr *)
                |Lessthan-> let _= output_string func_file " < " in func_file
                |Greaterthan-> let _= output_string func_file " > " in func_file
                |Lessequal-> let _= output_string func_file " <= " in func_file
                |Greaterequal-> let _= output_string func_file " >= " in func_file
                |Equal-> let _= output_string func_file " == " in func_file
                |Notequal-> let _= output_string func_file " ~= " in func_file
                |Logicaland-> let _= output_string func_file " && " in func_file
                |Logicalor-> let _= output_string func_file " || " in func_file

let build_unary_operator func_file op =
        match op with
        |Negation ->let _= output_string func_file " ~ " in func_file
    |Ninus ->let _= output_string func_file " - " in func_file

let build_id func_file identifier =
        let i = identifier in
        let _ = output_string func_file i in
        func_file

let build_const func_file const =
        match const with
                |Intc(v) -> let _= output_string func_file (string_of_int v) in func_file
                |Charc(v) -> let _= output_string func_file "'" in
```

```
                              let _= output_string func_file (String.make 1 v) in
                             let _= output_string func_file "'" in
                                        func_file
          |Floatc(v) -> let _= output_string func_file (string_of_float v) in func_file

let build_init func_file init =
     match init with
          | Init (var_name,var_val) -> match var_val with
                                      |None -> func_file
                                      |Consopt(x) -> let _= build_id func_file var_name in

                                        let _= output_string func_file "=" in

                                        let _= build_const func_file x in

                                        let _= output_string func_file ";\n" in

                                        func_file


let rec build_expr func_file expr =
     match expr with
          |Lval (lval)-> build_lvalue func_file lval
          |Const(const)->( match const with
                                             | None -> func_file
                                             | Consopt(c) -> build_const func_file c )
          |Assign(lval,e)->      let _ = build_lvalue func_file lval in
                                let _= output_string func_file "=" in
                                     build_expr func_file e
          |Binop (e1,op,e2)->let _= output_string func_file "(" in
                              let _=
                                   ( match op with
                                 | Mod -> let _= output_string func_file "mod(" in
                                         let _= build_expr func_file e1 in
                                        let _= output_string func_file "," in
                                        let _= build_expr func_file e2 in
                                        let _= output_string func_file ")" in
                                              func_file
                                  | _ ->
                                         let _= build_expr func_file e1 in
                                        let _= build_binary_operator func_file op in

                                        build_expr func_file e2) in
                             let _= output_string func_file ")" in
                                             func_file
          |Unaryop (op,e1)->    let _= build_unary_operator func_file op in
                                let _= output_string func_file "(" in
                                let _ = build_expr func_file e1 in
                                let _= output_string func_file ")" in
                                       func_file
          |Comma (e1,e2)-> let _=build_expr func_file e1 in
                              let _= output_string func_file "," in
                                 build_expr func_file e2
          |This (e1,e2)-> let _= output_string func_file "this(" in
                          let _=build_expr func_file e1 in
                          let _= output_string func_file "," in
                          let _=build_expr func_file e2 in
                          let _= output_string func_file ",1)" in
                                 func_file
          |Funcall (f_name,e_list)-> let _=build_id func_file f_name in
                                 let _= output_string func_file "(" in
                          (* helper function to print arguments in a function call *)
                                 let build_expr2 func_file e =
                                                output_string func_file ",";
                                                build_expr func_file e
                                                          in
               (* if there is 0 argument passed, "_" case will be match so nothing printed *)
               (* if there is 1 argument passed, "hd::tl" will be matched and the first
expressino is printed normally *)
               (* "tl" is [], so List.fold_left will do nothing but return func_file *)
               (* if there is more than 1 arguments, "hd::tl" matched and "hd" printed
normally and before print each *)
               (* expression in "tl", an extra "," will be added*)
                                 let _=(match e_list with
                                   | hd::tl -> let _=build_expr func_file hd in

                                    List.fold_left build_expr2 func_file tl
                                   | _ -> func_file) in
                                       let _= output_string func_file ")" in
                                              func_file
and build_lvalue func_file lval =
     match lval with
          |Variable (var) -> build_id func_file var
          |Array(var,e) -> let _= build_id func_file var in
                                         let _=output_string func_file "(" in
                                         let _= build_expr func_file e in
                                         let _=output_string func_file ")" in
                                         func_file
          |Thisl(e) -> let _= build_expr func_file e in func_file (*(match e with
                                     |Comma(e1,e2) -> let _= output_string func_file
"this(" in
let _=build_expr func_file e1 in

let _= output_string func_file "," in

let _=build_expr func_file e2 in

let _= output_string func_file ")" in

  func_file
                                                | _-> func_file )*)
(* get all identifiers from video declaration so that we can initialize it later, which is
required by matlab lib*)
let get_id ids init =
       match init with
              | Init (var_name,_) -> var_name::ids

let build_video func_file identifier =
       let _=output_string func_file (identifier^".cdata=[];\n"^identifier^".colormap=[];\n")
in
       func_file

let build_decl func_file decl =
       match decl with
              |    VarDecl(var_decl) -> (match var_decl with
                                          | Ddecl(t,init_list) -> (match t with

                                              | Video -> let _ =List.fold_left build_init func_file
                                              init_list in

                                           let all_ids= List.fold_left get_id [] init_list in

                                           List.fold_left build_video func_file all_ids



                                       | _ -> List.fold_left build_init func_file init_list

                                     )
                                                                   )

       |    ArraDecl(_) -> func_file

let rec build_statement func_file stmt =
        match stmt with
              | Expression(expr) -> let _=build_expr func_file expr in
```

```
                                                    let _= output_string func_file ";\n" in
                                                         func_file
                        | Compound(decls,st) -> let _= List.fold_left build_decl func_file decls in
                                                     let _=List.fold_left build_statement func_file st in
                                                     let _= output_string func_file "\n" in
                                                               func_file
                      | If(expr,st1,st2) -> let _= output_string func_file "if " in (*FIXME: ELSE*)
                                                 let _= build_expr func_file expr in
                                                let _= output_string func_file "\n" in

                                                let _ = build_statement func_file st1 in
                                                let _= output_string func_file "else \n" in
                                                let _ = build_statement func_file st2 in
                                                let _= output_string func_file "\nend\n" in
                                                func_file
                        | While(expr,st) ->      let _= output_string func_file "while " in
                                                     let _= build_expr func_file expr in
                                                    let _= output_string func_file "\n" in
                                                    let _ = build_statement func_file st in
                                                    let _= output_string func_file "\nend \n" in
                                                         func_file
                        | For(expr1,expr2,expr3,st)    ->    let _= output_string func_file "for " in
                                                                  let _= build_expr func_file expr1 in
                                                               let _=    match expr3 with

                                                         | Assign(_,sub_e1) -> (output_string func_file ":" ;

match sub_e1 with (* for (i=1;i<10;i=i+1) is the only form that is allowed *)

                                                     | Binop(_,_,e2) -> build_expr func_file e2

                                                     | _ -> func_file )(*FIXME*)
                         | _ -> func_file  in
                                          let _= output_string func_file ":" in
                                          let _= match expr2 with
                                               | Binop(_,_,e) -> build_expr func_file e
                                               | _ -> func_file in
                                                        let _= output_string func_file "\n" in

                                                      let _= build_statement func_file st in

                                                         let _= output_string func_file "\nend \n" in
                                                                   func_file

                        | Continue-> let _= output_string func_file "continue \n" in func_file
                        | Break -> let _= output_string func_file "break; \n" in func_file
| Return(expr) -> let _= output_string func_file "ret_vopl_xxx=" in
                  let _= build_expr func_file expr in
                  let _= output_string func_file ";\n return \n" in
                         func_file
| Load(var,file,e1,e2) -> let file = remove_qoute file in
                          let _ = build_id func_file var in
                         let _= output_string func_file ("=load_video("^file^",") in
                         let _= build_expr func_file e1 in
                         let _= output_string func_file "," in
                         let _= build_expr func_file e2 in
                         let _= output_string func_file ");\n" in
                              func_file
| Store(var,file) -> let file = remove_qoute file in
                     let _= output_string func_file "store_video(" in
                     let _= build_id func_file var in
                     let _= output_string func_file (",\""^file^"\");\n")
                            in func_file
| Insert(var1,var2,e) ->
                       let _ = build_id func_file var2 in
                      let _= output_string func_file "=insert_video(" in
                      let _ = build_id func_file var2 in
                     let _= output_string func_file "," in
                      let _ = build_id func_file var1 in
                     let _= output_string func_file "," in
                     let _= build_expr func_file e in
                     let _= output_string func_file ");\n" in
                            func_file
| Delete(var,e1,e2) ->
                     let _ = build_id func_file var in
                     let _= output_string func_file "=delete_video(" in
                     let _ = build_id func_file var in
                     let _= output_string func_file "," in
                     let _= build_expr func_file e1 in
                     let _= output_string func_file "," in
                     let _= build_expr func_file e2 in
                     let _= output_string func_file ");\n" in
                            func_file
| Copy(var1,e1,e2,var2) ->
                      let _ = build_id func_file var2 in
                     let _= output_string func_file "=copy_video(" in
                     let _ = build_id func_file var1 in
                     let _= output_string func_file "," in

                     let _= build_expr func_file e1 in
                     let _= output_string func_file "," in
                     let _= build_expr func_file e2 in
```

```
                              let _= output_string func_file "," in
                              let _ = build_id func_file var2 in

                              let _= output_string func_file ");\n" in
                                   func_file
          (* update v from index1 to index2 { ....} *)
          (* ==>                                          *)
          (* for v_index = index1:index2                  *)
          (*      this=v(:,:,v_index);                    *)
          (*        .......                               *)
          (* end                                          *)

          | Update(var,e1,e2,st) ->
                                     let _= output_string func_file "for v_index = " in
                                      let _ = build_expr func_file e1 in
                                      let _= output_string func_file ":" in
                                      let _= build_expr func_file e2 in
                                      let _= output_string func_file "\nthis=" in
                                      let _= build_id func_file var in
                                      let _= output_string func_file "(v_index).cdata;\n" in
                                      let _= build_statement func_file st in
                                      let _= output_string func_file "\nend \n" in
                                           func_file



let build_function_type func_file t =
     match t with
          |Int
          |Char
          |Float
          |Video -> let _ =output_string func_file " ret_vopl_xxx=" in
                        func_file
          |Void    -> let _ =output_string func_file " " in
                        func_file

let build_arg func_file para =
     match para with
          | Arg(_, para_name) ->
     let func_file = build_id func_file para_name in
     func_file


let build_function func_file f =
     match f with
          | Func(t, f_name, args, stmts) ->
```

```
    let _= output_string func_file "function" in
    let func_file = build_function_type func_file    t in
    let func_file = build_id func_file f_name in
    let _ = output_string func_file "(" in
    (*let _ = print_int (List.length args) in*)
    (* helper function to print arguments in a function def *)
    let build_arg2 func_file ar =
            output_string func_file ",";
            build_arg func_file ar
    in
    let _=(match args with
            | hd::tl -> let _=build_arg func_file hd in
                    List.fold_left build_arg2 func_file tl
            | _ -> func_file) in
    let _= output_string func_file ")\n" in
    let func_file = build_statement func_file stmts in
    func_file


let build p filename=
    match p with
        | Pro(prog) ->
    let n = String.index filename '.' in
    let filename = String.sub filename 0 n in
    let function_list =List.rev prog in
    let main_channel =
        open_out ( (filename)^".m") in
  List.fold_left build_function main_channel function_list
```

copy_video.m

```
% initialize var2.cdata and var2.colormap before use.
%

function var2=copy_video(var1,btime,etime,var2)
cnt=1;
[d1,d2]=size(var2);
fprintf('Copying Video...\n');
if (d2==1)

for k=btime:etime
     var2(cnt)=var1(k);
     cnt=cnt+1;
end
```

```
else
        for k=btime:etime
        var2(d2+cnt)=var1(k);
        cnt=cnt+1;
        end
end
[d1,d2]=size(var2);
fprintf('The Number of Frames is Now %d\n',d2);
fprintf('Copying Finished\n');
```

delete_video.m

```
function var1=delete_video(var1,btime,etime)
fprintf('Deleting Video...\n');
for k=btime:etime
        var1(btime)=[];
end
[d1,d2]=size(var1);
fprintf('The Number of Frames is Now %d\n',d2);
fprintf('Deleting Finished\n');
```

insert_video.m

```
function var=insert_video(var1,var2,e)
fprintf('Inserting Video...\n');
[d1,d2,d3]=size(var1(1).cdata);
[d4,d5,d6]=size(var1(1).colormap);
if(d4>0)
for i=1:length(var2)
        frame(i)=var2(i);
        frame(i).cdata=imresize(frame(i).cdata,[d1 d2]);
        frame(i).colormap=gray(256);
        var2(i)=frame(i);
end
        else
                for i=1:length(var2)
        frame(i)=var2(i);
        frame(i).cdata=imresize(frame(i).cdata,[d1 d2]);
        var2(i)=frame(i);
                end
end
```

```
var=[var1(1:e) var2 var1(e+1:length(var1))];
[d1,d2]=size(var);
fprintf('The Number of Frames is Now %d\n',d2);
fprintf('Inserting Finished\n');
```

load_video.m

```
function mov=load_video(a,b,c)
global AVI;
[path,name,ext] = fileparts(a);
if(ext=='.avi')

    AVI=1;
    fprintf('Loading Video...\n');
    mov=aviread(a);
    info=aviinfo(a);
    fprintf('The number of frames of %s is: %d\n',a,info.NumFrames);
    fprintf('Loading Finished\n');
elseif(ext=='.yuv')

    AVI=0;
    fprintf('Please Wait For the Format Transforming...\n');
    yuv2avi(a,[b c],name,'none',30);
    fprintf('Loading Video...\n');
    mov=aviread(name);
    info=aviinfo(name);
    fprintf('The number of frames of %s.yuv is: %d\n',name,info.NumFrames);
    fprintf('Loading Finished\n');
end
```

store_video.m

```
function store_video(mov,b)
fprintf('Storing Video...\n');
[d1,d2,d3]=fileparts(b);
if(d3=='.avi')
movie2avi(mov,b,'fps',30,'quality',100,'compression','None');
else
mov2yuv(b,mov,'420');
end
fprintf('Storing Finished\n');
```

yuv2avi.m

```
function numfrm=yuv2avi(yuvfilename,dims,avifilename,compression,fps)
numfrm = seq_frames(yuvfilename,dims);
avi = avifile(avifilename,'fps',fps,'quality',100,'compression',compression);
h = waitbar(0,'Please wait While transforming... ');
for i=1:numfrm
        [Y, U, V] = yuv_import(yuvfilename,dims,1,i-1);
        yuv(:,:,1) = Y{1};
        yuv(:,:,2) = imresize(U{1},2,'bicubic');
        yuv(:,:,3) = imresize(V{1},2,'bicubic');
        rgb = yuv2rgb(Y{1},U{1},V{1});
        avi = addframe(avi,rgb);
        waitbar(i/numfrm,h);
end;
avi = close(avi);
close(h);
```

8.2 Syntax Test Script

pr_st.ml

```
open Ast

let print_unaryop unary_operator =
      match unary_operator with
Negation -> print_string "!"
|Ninus -> print_string "-"


let print_oper oper =
      match oper with
Plus -> print_string "+"
|Minus -> print_string "-"
|Times -> print_string "*"
|Divide -> print_string "/"
|Mod -> print_string "%"
|Lessthan -> print_string "<"
```

```
|Greaterthan -> print_string ">"
|Lessequal -> print_string "<="
|Greaterequal -> print_string ">="
|Equal -> print_string "=="
|Notequal -> print_string "!="
|Logicaland -> print_string "&&"
|Logicalor -> print_string "||"

let print_const constant =
      match constant with
  Intc(int) -> print_int int
|Charc(char) -> let _=print_string "'" in
                          let _=print_char char in
                          print_string "'"
|Floatc(float) -> print_float float

let print_constopt const_opt =
      match const_opt with
  None -> print_string ""
|Consopt(constant) ->print_const constant

let print_type t =
      match t with
|Int -> print_string "int"
|Char -> print_string "char"
|Float -> print_string "float"
|Video -> print_string "video"
|Void -> print_string "void"

let print_arraydecl arraydecl =
      match arraydecl with
Adecl(adecl_type,array_name,int)-> let _=print_type adecl_type in
                                                    let _=print_string " " in
                                                    let _=print_string array_name in
                                                    let _=print_string "[" in
                                                    let _=print_int int in
                                                    print_string "]"

let print_initlist1 init =
      match init with
Init (init_name,const_opt)-> let _=print_string init_name in
                                          (match const_opt with
                                            None-> print_constopt const_opt
                                          | _-> let _=print_string "=" in
                                                  print_constopt const_opt)
```

```
let print_initlist2 init =
  let _=print_string ","in
  print_initlist1 init

let print_initlist i_init_list =
       match i_init_list with
    hd::tl -> let _=print_initlist1 hd in
                 List.iter print_initlist2 tl
| _-> print_string " "

let print_vardecl vardecl =
       match vardecl with
  Ddecl(ddecl_type,i_init_list) -> let _=print_type ddecl_type in
                                    let _=print_string " " in
                                     print_initlist i_init_list

let print_delc decl =
       match decl with
VarDecl(variable_declaration)->let _=print_vardecl variable_declaration in
                                  print_string ";"
|ArraDecl(array_declaration) -> let _=print_arraydecl array_declaration in
                                  print_string ";"

let rec print_lv lv =
  match lv with
  Variable(variable) -> print_string variable
|Array(arrayname,expr) -> let _=print_string arrayname in
                           let _=print_string "[" in
                           let _=print_expr expr in
                           print_string "]"
|Thisl(expr) -> let _=print_string "this(" in
                 let _=print_expr expr in
                 print_string ")"

and
  print_expr expr =
       match expr with
  Lval(lvalue) ->print_lv lvalue
|Const(const_opt) -> print_constopt const_opt
|Assign(lvalue,expr) -> let _=print_lv lvalue in
                         let _=print_string "=" in
                        print_expr expr
|Binop(expr1,operator,expr2) -> let _=print_expr expr1 in
                                  let _=print_oper operator in
                                   print_expr expr2
|Unaryop(unary_operator,expr)-> let _=print_unaryop unary_operator in
```

```
                                              print_expr expr
|Comma(expr1,expr2) -> let _=print_expr expr1 in
                                  let _=print_string "," in
                                  print_expr expr2
|This(expr1,expr2) -> let _=print_string "this(" in
                             let _=print_expr expr1 in
                              let _=print_string "," in
                              let _=print_expr expr2 in
                              print_string ")"
|Funcall(f_name,expr_list) -> let _=print_string f_name in
                                       let _=print_string "(" in
                                       let _=print_expr1 expr_list in
                                       print_string ")"

and
print_expr1 expr_list =
match expr_list with
  hd::tl -> let _=print_expr hd in
                List.iter print_expr3 tl
|_-> print_string " "

and
print_expr3 e =
  let _=print_string ","in
  print_expr e

let rec print_stmt stmt =
       match stmt with
  Expression(expr)-> let _=print_expr expr in
                            print_string ";\n"
|Compound(decl_list,stmt_list)-> let _=print_string "{\n" in
                                          let _= List.iter print_delc decl_list in
                                          let _=List.iter print_stmt stmt_list in
                                         print_string "\n}\n"
|If(expr,stmt1,stmt2) -> let _= print_string "if(" in
                                let _= print_expr expr in
                                let _= print_string ")\n" in
                               let _= print_stmt stmt1 in
                                let _= print_string "\n else\n" in
                                let _= print_stmt stmt2 in
                                print_string "\n"
|While(expr,stmt) -> let _=print_string "while(" in
                           let _=print_expr expr in
                            let _=print_string ")\n" in
                            print_stmt stmt
|For(expr1,expr2,expr3,stmt) ->let _=print_string "for(" in
                                          let _=print_expr expr1 in
                                           let _=print_string ";" in
                                           let _=print_expr expr2 in
                                         let _=print_string ";" in
                                         let _=print_expr expr3 in
                                         let _=print_string ")\n" in
                                         print_stmt stmt
|Continue -> print_string "continue;\n"
|Break -> print_string "break;\n"
|Return(expr) -> let _=print_string "return " in
                  let _=print_expr expr in
                  print_string ";\n"
|Load(l_name,l_file,expr1,expr2)->let _=print_string "load " in
                                  let _=print_string l_name in
                                  let _=print_string " from " in
                                   let _=print_string l_file in
                                  let _=print_string " with " in
                                   let _=print_expr expr1 in
                                   let _=print_string " and " in
                                   let _=print_expr expr2 in
                                   print_string ";\n"
|Store(s_name,s_file) -> let _=print_string "store " in
                         let _=print_string s_name in
                         let _=print_string " to " in
                          let _=print_string s_file in
                         print_string ";\n"
|Insert(i_name1,i_name2,expr) -> let _=print_string "insert " in
                                 let _=print_string i_name1 in
                                 let _=print_string " to " in
                                  let _=print_string i_name2 in
                                  let _=print_string " from " in
                                  let _=print_expr expr in
                                 print_string ";\n"
|Delete(d_name,expr1,expr2) -> let _=print_string "delete " in
                                let _=print_string d_name in
                                let _=print_string " from " in
                               let _=print_expr expr1 in
                                let _=print_string " to " in
                                let _=print_expr expr2 in
                                print_string ";\n"
|Copy(c_name1,expr1,expr2,c_name2) -> let _=print_string "copy " in
                                       let _=print_string c_name1 in
                                       let _=print_string " from " in
                                        let _=print_expr expr1 in
                                       let _=print_string " to " in
                                       let _=print_expr expr2 in
```

```
                                                    let _=print_string " to " in
                                                    let _=print_string c_name2 in
                                                    print_string ";\n"
|Update(u_name,expr1,expr2,stmt) -> let _=print_string "update " in
                                               let _=print_string u_name in
                                                let _=print_string " from " in
                                                let _=print_expr expr1 in
                                                let _=print_string " to " in
                                                let _=print_expr expr2 in
                                                let _=print_string "\n" in
                                                print_stmt stmt

let print_arg1 arg =
      match arg with
      Arg(arg_type,arg_name)-> let _= print_type arg_type in
                                         let _=print_string " " in
                                          print_string arg_name
let print_arg2 arg =
  let _=print_string "," in
  print_arg1 arg

let print_arg f_arglist =
      match f_arglist with
hd::tl -> let _=print_arg1 hd in
            List.iter print_arg2 tl
| _-> print_string " "

let print_func func =
    match func with
    Func(f_type,f_name,f_arglist,f_body) -> let _= print_type f_type in
                                                     let _=print_string " " in
                                                      let _= print_string f_name in
                                                      let _=print_string "(" in
                                                      let _=print_arg f_arglist in
                                                      let _=print_string ")\n" in
                                                      let _=print_stmt f_body in
                                                      print_string "\n"
let print_program prog =
  match prog with
  Pro(func_list) -> List.iter print_func func_list
```

8.3 Test Cases

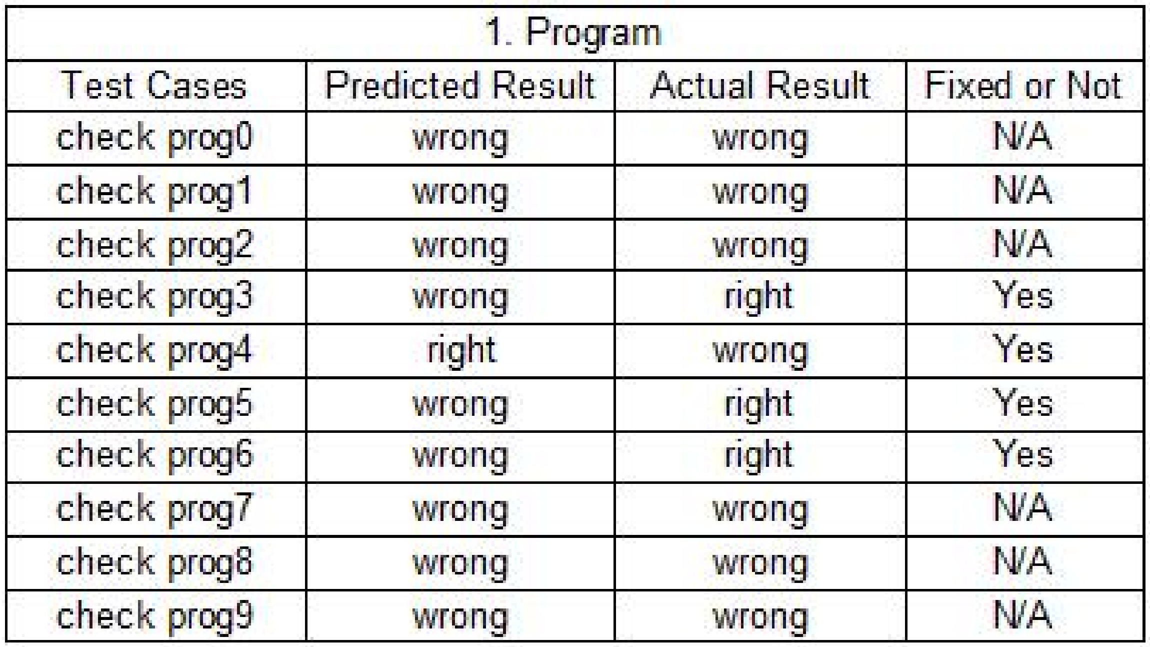
8.3.1 Syntax Test Cases

```
/* testprog1 */

void main( ){ }


/* testprog2 */

int foo( )
{ }
float erew1234( )
{ }
char dsr_srew_1343( )
{ }
void main( )
{ }

/* testexpr */

int foo( )
{k=k+1;
  s=m*k;
  a;
array[5];
this(3,4);
3;
'r';
}


/* testfunc */

int foo(int a,float t,char i,video p)
{int a; float f; char k; video v;
  int a=2; float l=5.56; char j='a';
  int p,b=5,y;
  int array[10];
}


/* testiterationstmt */

int foo( )
```

```
{while(a!=b)
  { k=k+a;
      a=3*4;
      p=f%2;
      c='s';}
  for(i=1;i<=10;i+1)
  { a[0]=1;
      k=5;}
}


/* testjumpstmt */

int foo( )
{break;
  continue;
  return a;}


/* testselectstmt */

int foo( )
{if(a<2)
  b=b+1;
  else
  b=-b;
  if(k>=a)
  {k=a;
    a=2;}
  else
  {k=b;
    b=a+3;}
}


/* testvideostmt1 */

int foo(int a,float t,char i,video p)
{load a from ".\afdsafas.mll" with 3 and 4;
  store a to "sadfasf";
}


/* testvideostmt2 */

int foo( )
```

```
{load a from "safda" with 3 and 4;
  store a to "asfasdf";
  insert a to b from 5;
  delete a from 6 to 8;
  copy a from 9 to 10 to b;
  update a from 11 to 12
  { }
}
```

8.3.2 Semantic Test Cases



1. Program			
Test Cases	Predicted Result	Actual Result	Fixed or Not
check prog0	wrong	wrong	N/A
check prog1	wrong	wrong	N/A
check prog2	wrong	wrong	N/A
check prog3	wrong	right	Yes
check prog4	right	wrong	Yes
check prog5	wrong	right	Yes
check prog6	wrong	right	Yes
check prog7	wrong	wrong	N/A
check prog8	wrong	wrong	N/A
check prog9	wrong	wrong	N/A

2. Expression			
Test Cases	Predicted Result	Actual Result	Fixed or Not
check prog0	wrong	wrong	N/A
check prog1	wrong	wrong	N/A
check prog2	wrong	wrong	N/A
check prog3	wrong	wrong	N/A
check prog4	wrong	wrong	N/A
check prog5	right	right	N/A
check prog6	wrong	wrong	N/A
check prog7	wrong	wrong	N/A
check prog8	wrong	wrong	N/A
check prog9	right	right	N/A
check prog10	right	right	N/A
check prog11	wrong	right	Yes
check prog12	wrong	wrong	N/A
check prog13	wrong	wrong	N/A
check prog14	right	right	N/A
check prog15	wrong	right	N/A
check prog16	wrong	wrong	N/A
check prog17	wrong	right	Yes
check prog18	right	right	N/A
check prog19	right	right	N/A

3. Statement			
Test Cases	Predicted Result	Actual Result	Fixed or Not
check prog0	wrong	wrong	N/A
check prog1	wrong	wrong	N/A
check prog2	wrong	wrong	N/A
check prog3	wrong	wrong	N/A
check prog4	right	right	N/A
check prog5	wrong	wrong	N/A
check prog6	wrong	wrong	N/A
check prog7	wrong	wrong	N/A
check prog8	wrong	wrong	N/A
check prog9	wrong	wrong	N/A
check prog10	wrong	wrong	N/A
check prog11	wrong	wrong	N/A
check prog12	wrong	wrong	N/A
check prog13	wrong	wrong	N/A
check prog14	wrong	right	Yes
check prog15	wrong	right	Yes
check prog16	wrong	wrong	N/A
check prog17	wrong	wrong	N/A
check prog18	right	right	N/A
check prog19	right	right	N/A
check prog20	right	right	N/A
check prog21	right	right	N/A
check prog22	right	right	N/A
check prog23	right	right	N/A
check prog24	right	right	N/A
check prog25	right	right	N/A
check prog26	right	right	N/A