Reference Manual

VideO Processing Language (VOPL)

Oct.19th, 2008

Baolin Shao （ bs2530 ）
Xuyang Shi （xs2137 ）
Huning Dai (hd2210)
Jia Li （ jl3272 ）

Table of Content

1. Introduction 4
2. Document Convention 4
3. Lexical Conventions 4
3.1 Comments 5
3.2 Identifiers 5
3.3 Keywords 5
3.4 Operators 5
3.4.1 Binary Operator 5
3.4.2 Unary Operator 6
3.4.3 Comparison Operator 6
3.4.4 Equality Operator 7
3.4.5 Logical Operator 7
3.5 String Literals 8
3.6 Constants 8
3.6.1 Integer 8
3.6.2 Floating 8
3.6.3 Character 9
4. Types 9
4.1 Basic Types 9
4.2 Video Types 9
4.3 Void 9
5. Declarations 9
5.1 Variable Declaration 10
5.2 Array Declaration 11
6. Expression 11
6.1 Primary Expression 12
6.1.1 lvalue 12
6.1.2 Constant 12
6.2 Assignment Expression 13
6.3 Binary Expression 13
6.4 Unary Expression 13
6.5 Parenthesis Expression 14
6.6 This Expression 14
6.7 Function Call 14
6.8 Expression List 14

7. Statement **15**
7.1 Expression Statement **15**
7.2 Compound Statement **15**
7.3 Selection Statement **16**
7.4 Iteration Statement **16**
7.4.1 While loop 16
7.4.2 For loop 16
7.5 Jump Statement **17**
7.5.1 Continue 17
7.5.2 Break 17
7.5.3 Return 17
7.6 Video Statement **17**
8. Scope **19**
9. Program **20**
9.1 Function **20**
9.2 Main Function **20**
10. Grammar **21**

1. Introduction

Nowadays, multi-media are dominating our everyday life, and we are used to digital cameras and photo editing tools, such as PhotoShop. From a programmer's perspective, many languages provide libraries and tools to reduce programmers' burden in photo-editing application development. However, we have not seen a popular language aimed for designing video-editing programs, so programmers have to spend much time on understanding format of videos and carefully manipulating frames in a video and pixels in a frame, rather than designing the core processing algorithms. Therefore, we propose to develop a VideO Processing Language (VOPL) by which programmers can easily manipulate videos. 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.

2. Document Convention

In the following sections, Italic letters represent non-terminals in grammar and **bold letters** represent **keywords or terminals**.

3. 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. From section 3.1 to section 3.6, we will explain every kind of token and comments in detail.

3.1 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.

3.2 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, otherwise a compile-time error will be reported.

3.3 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**	

3.4 Operators

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

3.4.1 Binary Operator

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

3.4.1.1 expression + expression

The binary operator + indicates addition. The result gives the sum of the two expressions. The type of the two operands can be int, float and double. 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.

3.4.1.2 expression – expression

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

3.4.1.3 expression * expression

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

3.4.1.4 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 +.

3.4.1.5 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.

3.4.2 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 ints and chars. 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.

3.4.3 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 the two expressions. When the specified relation is false, the result gives 0; when the specified relation is true, the result gives 1.

3.4.4 Equality Operator

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

3.4.4.1 expression == expression

3.4.4.2 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.

3.4.5 Logical Operator

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

3.4.5.1 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.

3.4.5.2 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.

3.5 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.

3.6 Constants

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

3.6.1 Integer

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

3.6.2 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 keywords is **float**.

3.6.3 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**.

4. Types

4.1 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.

4.2 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.

4.3 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.

5. 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:

*/*empty*/*

| *declaration*

| *declaration_list declaration*

declaration:

variable_declaration ***SEMICOLON***

| *array_declaration* ***SEMICOLON***

5.1 Variable Declaration

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

variable_declaration:

type ID_list

ID_list:

ID *opt_init*

| *ID_list* **COMMA ID** *opt_init*

opt_init:

*/*empty*/*

| ***ASSIGN*** *constant*

type 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 opt_init. The constant value in opt_init must have the same type of the variable.

5.2 Array Declaration

Arrays can be declared as follow:

array_declaration:

type ***ID LBRACKET INTCONST RBRACKET SEMICOLOUM***

VOPL only supports one-dimention 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 a 0 as their initial values.

6. Expression

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

expression:

primary_expression

| *assignment_expression*

| *binary_expression*

| *unary_expression*

| ***LPAREN*** *expression* ***RPAREN***

| *functioncall*

6.1 Primary Expression

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

primary_expression:

lvalue

| *constant*

6.1.1 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-negtive integer, otherwise an error will occur. this_expr (See 6.6) 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.

6.1.2 Constant

A constant can be a primary expression. Its type is as showed in 4.

constant:

INTCONST

| ***CHARCONST***

| ***FLOATCONST***

6.2 Assignment Expression

An assignment expression is a kind of expression with the form of ‘lvalue=expression’. Lvalue is as showed in 6.1.1. 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*

6.3 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. Operator inside is as showed in 3.4.

binary_expression:

expression operator expression

6.4 Unary Expression

Unary expression has the form

unary_expression:

***NEGATION** expression*

| ***MINUS** expression*

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

6.5 Parenthesis Expression

A parenthesis expression has a form that expression in parentheses as '(expression)'. Its value is that of its sub-expression.

6.6 This Expression

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 row x and colum y in some specific frame. The indexes in this_expression must be non-negative integers. The syntax of this expression is

this_expression:

THIS LPAREN *expression* ***COMMA*** *expression* ***RPAREN***.

6.7 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* ***RPAREN***

6.8 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*

7. Statement

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

7.1 Expression Statement

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

expression_statement:
    expression ***SEMICOLON***

the expressions inside are showed as 6. 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.

7.2 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 statement_list RBRACE

statement_list:
    statement
| *statement_list* ***SEMICOLO****N statement*
| */*empty*/*

All declarations must be written before all statements.

7.3 Selection Statement

Selection statements have a form as following:

selection_statement:

IF LPAREN *expression* ***RPAREN*** *statement else_opt*

else_opt:

*/*empty*/*

| ***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 else_opt is not empty. If else-opt is empty, selection statement will finish without any choice when the expression equals to 0.

7.4 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*

7.4.1 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.

7.4.2 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.

7.5 Jump Statement

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

jump_statement:

CONTINUE SEMICOLON

| ***BREAK SEMICOLON***

| ***RETURN*** *expression* ***SEMICOLON***

7.5.1 Continue

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

7.5.2 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.

7.5.3 Return

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

7.6 Video Statement

7.6.1 Load

The load statement will put data into the 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.

7.6.2 Store

As its name tells, store statement will put the processed data back to file

STORE ID TO STRINGCONST SEMICOLON

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

7.6.3 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.

7.6.4 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.

7.6.5 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.

7.6.6 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_expression.

8. 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 compound_statement constitutes a scope and one scope can be nested in another. Identifiers in one scope do not conflict with those in another, even if they have the same lexical representation.

9. 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:

function_list main_function

9.1 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:

```
function:
    type|VOID ID    LPAREN argument_list RPAREN compound_statement

argument_list_opt
   /*empty*/
  | argument_list

argument_list:
    argument
  |   argument_list COMMA argument

argument:
   variable_declaration
```

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

9.2 Main Function

Main function is the starting point of every program. Its type is void and it does not carry any parameter.

main_function:

***VOID MAIN LPAREN RPAREN** compound_statement*

10. Grammar

program:
    function_list main_function

fuction_list:
    */*empty*/*
  | *function*
  | *fuction_list function*

function:
  type|VOID ID   LPAREN argument_list RPAREN compound_statement

argument_list_opt
   */*empty*/*
  | *argument_list*

argument_list:
    argument
  | *argument_list COMMA argument*

argument:
   variable_declaration

declaration_list:
   */*empty*/*
| *declaration*
| *declaration_list declaration*

declaration:
    variable_declaration SEMICOLON
  | *array_declaration SEMICOLON*

variable_declaration:
    type ID_list

```
ID_list:
ID opt_init
|   ID_list COMMA ID opt_init

opt_init:
/*empty*/
|   ASSIGN constant

array_declaration:
    pre_type ID LBRACKET INTCONST RBRACKET

constant:
    INTCONST
 |   CHARCONST
 |   FLOATCONST

main_function:
    VOID MAIN LPAREN RPAREN compound_statement

type:
    pre_type
   | VIDEO

pre_type:
     INT
 |   FLOAT
 |   CHAR

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 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 operator expression

operator:
    binary_operator
  |  comparison_operator
  |  equality_operator
|  logical_operator

binary_operator:
    PLUS
  |  MINUS
  |  TIMES
  |  DIVIDE
  |  MOD

comparison_operator:
    LESSTHAN
  |  GREATERTHAN
  |  LESSEQUAL
  |  GREATEREQUAL

equity_operator:
  |  EQUAL
```

```
 |  NOTEQUAL

logical_perator:
    LOGICALAND
 |  LOGICALOR

unary_expression:
    NEGATION expression
  | MINUS expression

comma_expression:
    expression COMMA expression

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

expression_statement:
    expression SEMICOLON

compound_statement:
    LBRACE declaration_list statement_list RBRACE

statement_list:
    statement
 |  statement_list SEMICOLON statement
 |  /*empty*/

selection_statement:
    IF LPAREN expression RPAREN statement else_opt

else_opt:
    /*empty*/
 |  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
```