EZIP: Easy Image Processing

Kevin Chiu, Tejas Nadkarni, Swati Kumar, Avanti Dharkar

{kgc2113, tgn2104, sak2144, ad2518}@columbia.edu

October 18, 2007

Contents

1 Introduction

Ezip stands for Easy Image Processing. The purpose of this language is to allow its users to easily manipulate images using various image processing techniques. Ezip syntax is designed to be intuitive and easier for developers to understand compared to C, C++ or Java syntax. Using a few lines of codes the user can perform complex Image operations which would take many more lines of code in any other general purpose coding language.

Generally image processing algorithms have an image and a kernel that works on that image. Images can be loaded as a matrix of integers. A kernel can be defined as a 3x3 matrix. Ezip also allows for Image addition, subtraction and other simple Image operations. Ezip is written in Java, and works by converting the Ezip code into Java code using a compiler generated through Antlr.

2 Lexical conventions

2.1 Comments

EZIP follows Cs's syntax for comments. Multiline comments start with `/*` and end with `*/`. Single line comments start with `//`.

2.2 Identifiers

An identifier is a sequence of alphanumeric characters and underscores. Identifiers must start with a letter. The identifiers are case sensitive. Identifiers have unlimited length.

2.3 Keywords

The following identifiers are used as keywords in EZIP. They may not be used for any other purpose.

`for`	`image`	`function`	`continue`	`in`
`if`	`kernel`	`for`	`return`	`to`
`else`	`int`	`forall`	`AND`	`true`
`true`	`double`	`while`	`OR`	`false`
`false`	`boolean`	`break`	`NOT`	

2.4 Constants

Ezip has `integer`, `double`, and String constants. `Integers` are a sequence of digits. `Doubles` are a sequence of digits followed by a period, followed by another sequence of digits. Strings are a sequence of characters surrounded by double quotes.

3 Types

- `int` values are integers from -2147483648 to 2147483647.
- `double` values are floating point numbers expressed in the format `(DIGIT)+.(DIGIT)+`. A `DIGIT` is a natural number between 0 and 9 inclusive.

- `boolean` values are either `true` or `false`.
- `image` is a proprietary type that is internally stored as a matrix, but is exposed as a primitive data type. Individual pixels are accessible using bracket annotation, ID[x][y], to access to specific pixels at the representative x and y coordinates of the image.
- `kernel` is a proprietary type that is internally stored as a matrix, but is exposed as a primitive data type. Individual kernel entries are accessible using bracket annotation, ID[x][y], to access individual values in the matrix.

4 Statements

4.1 Expression

A statement is an expression having the form

```
expression;
```

4.2 Block

A block is a type of statement that consists of several statements grouped together to form a block having the form

```
{ statement* }
```

4.3 Conditional

The format of the conditional statement is

```
if (expression) statement
if (expression) statement else statement
```

In both cases, `statement` is executed when `expression` is `true`. The `else` is always associated with the last encountered `else`-less `if`.

4.4 While Loops

The format of the `while` statement is

```
while ( expression ) statement
```

The `statement` is executed repeated while the `expression` is `true`. The `expression` is checked before each execution of the `statement`.

4.5 For Loops

The format of the `for` statement is

```
for ( expression; expression; expression ) statement
```

The first `expression` is executed when the program enters the loop. The `statement` inside the loop continues to run as long as the second `expression` holds true. The value of the second `expression` before the `statement` is executed in each run of the loop. At the end of each loop, the third `expression` is evaluated.

4.6 ForAll Loops

The format of the `forall` statement is

```
forall ( ID in INT to INT ) statement
```

The `ID` is instantiated and assigned the value of the first `INT`. The loop executes the `statement` as long as the `ID` is no greater than the second `INT`. The value of the `ID` is incremented after each execution of the `statement` block.

4.7 Break

The format of the `break` statement is

```
break
```

The `break` statement can only be used inside loops, such as `for` or `while` statements. It terminates the innermost loop in which it is placed.

4.8 Continue

The `continue` statement can only be used inside loops. It skips the evaluation of any following statements

4.9 Return

The format of the `return` statement is

```
return expression;
return;
```

The `return` statement can only be used inside function definitions. It returns the value of the expression to the containing function's caller or exits the function if there is nothing to return.

4.10 Escape

The `escape` statement can only be used inside loops. Calling `escape` inside a loop breaks out of all enclosing loops.

4.11 Assignments

The format of an assignment is

```
ID = expression;
ID[x,y] = expression;
```

The expression on the right is evaluated, and the value resolved from that evaluation is bound to the ID on the left. In the case that the ID is an array, the array element associated with the provided index(es)

4.12 Function Calls

The format of an assignment is

```
ID( expression list )
```

The expressions inside the comma-separated `expression list` are passed to the function defined by the `ID` for execution. Functions are considered expressions. When a functions return type is `void` the value of the expression is `null`.

4.13 Declarations

The format of a declaration is

```
TYPE ID;
```

Declarations and instantiations can be combined. The format for the combined statement is

```
TYPE ID = expression;
```

where `TYPE` is `int`, `double`, `boolean`, `kernel`, or `image`. The `expression` on the right is instantiated and the result is bound to the `ID` on the left.

Declarations can also be chained in interesting ways. Instantiations can be chained and include an optional assignments.

```
TYPE (ID = expression,)* ID = expression;
```

5 Expressions

5.1 Primary Expressions

A primary expression can be an identifier (`ID`), constant (`image`, `kernel`, `integer`, `double`), boolean (`true`, `false`), an expression contained in parenthesis, or an element (`image[x,y]`). All expressions evaluate to a primary expression.

5.2 Unary Expressions

There is one unary operator, `-`, which negates the expression. The actual results of this operation are different for different types. It is not legal to negate a boolean value.

- `int` values are multiplied by -1.
- `double` values are multiplied by -1.
- `image` values are inverted. We take the maximum of the color space of the image, typically 255, and subtract the current pixel values in the image from the maximum. This gives us a visually inverted image.
- `kernel` values are inverted. We take the maximum of the kernel and subtract the current kernel values from the maximum to find the inverted kernel.

5.3 Arithmetic Expressions

5.3.1 Integers and Doubles

The order of precedence from highest to lowest for operators on `integers` and `doubles` is

- `*` and `/`
- `+` and `-`

When `integers` and `doubles` are mixed in an expression, the expression is evaluated as a `double`.

5.3.2 Images and Kernels

The following list gives the order of precedence from highest to lowest for operators on `images` as well as explanations for their use.

- `**` takes an `image` on the left and a `kernel` on the right. Evaluating an expression containing a convolution involves executing a sum of products calculation over an image to produce a new image.
- `*` and `/` operators can be used in two ways with images. First, they can be used between a integer or double value and an image to multiply or divide the individual pixel entries in the image by the scalar value to produce a new image. Second, they can be used between two images to multiply or divide the corresponding pixel values in each image to produce a new image.
- `+` and `-` can be used in two ways with images. First, they can be used between a integer or double value and an image to add or subtract the individual pixel entries in the image to or from the scalar value to produce a new image. Second, they can be used between two images to add or subtract the corresponding pixel values in each image to produce a new image. In the case of subtraction, the image or scalar on the right is subtracted from the image or scalar on the left.

5.4 Logical Expressions

Logical expressions use the following operators to take pairs of boolean expressions and evaluate to boolean expressions, or in the case of `not`, a boolean to another boolean.

- `and` evaluates to `true` when both input expressions are `true` and `false` otherwise.
- `or` evaluates to `true` when either input expression is `true` and `false` otherwise.
- `not` evaluates to `true` when the input expression is `false` and evaluates to `true` otherwise.

5.5 Relational Expressions

The format of a relational expression is

```
EXPRESSION (relational operator) EXPRESSION
```

Both `EXPRESSION` statements must evaluate to the same type. The relational operators are as follows.

- `==` evaluates to `true` if both expressions resolve to equivalent values, and `false` otherwise. This operator can also be used to compare images. If both images have equal pixel values for every pixel defined in each image, then this evaluates to true, otherwise, it evaluates to `false`.
- != evaluates to `true` if both expressions resolve to unequal values, and `false` otherwise.
- `<` evaluates to `true` if the left expression resolves to a value less than the right expression, and `false` otherwise.
- `>` evaluates to `true` if the left expression resolves to a value greater than the right expression, and `false` otherwise.
- `<=` evaluates to `true` if the left expression resolves to a value less than or equal to the right expression, and `false` otherwise.
- `>=` evaluates to `true` if the left expression resolves to a value greater than or equal to the right expression, and `false` otherwise.

6 Function Definitions

The format of a function definition is

```
function (TYPE)? ID (((TYPE ID)(, TYPE ID)?)*) statement
```

A function begins with the keyword `function` followed by an optional `TYPE` specifier describing the return type of the function. The remainder of the function includes the `ID`, argument list, and function body.

7 Scope

7.1 Default Scope

If a variable is declared outside of a block it is assigned the default scope. The scope of this variable extends from where it is declared to the end of the file.

7.2 Local Scope

If a variable is declared inside of a block its scope is local to that block. The scope of this variable extends from where it is declared within the block to the end of the block.

7.3 Scope Conflicts

- A variable cannot be declared more than once in the current scope, not including surrounding or surrounded scopes.
- If a variable has been declared in a surrounding scope, that variable is accessible in the current scope.
- If a variable has been declared in a surrounding scope, redeclaring the variable in the current scope hides the variable that has been declared in the surrounding scope until execution leaves the bounds of the current scope.

A Lexer

```
class EzipLexer extends Lexer;
options{
     testLiterals = false;
     charVocabulary = '\3'..'\177';
     k = 2;
}

protected ALPHA     : ('a'..'z'|'A'..'Z'|'_') ;
protected DIGIT     : ('0'..'9') ;

ID options { testLiterals = true ;}
       : (ALPHA) ( ALPHA | DIGIT )*
       ;

NUM    : (DIGIT)+ ( '.' (DIGIT)+ )?;

WS     : (' ' | '\t')+ { $setType(Token.SKIP); } ;

NL     : ('\n' | ('\r' '\n') => '\r' '\n' | '\r')
            { $setType(Token.SKIP); newline(); }
       ;

STRING
     : '"'!
          ( '"' '"'!
          | ~('"'|'\n'|'\r')
          )*
```

```
        '"'!
      ;

LPAREN   : '(';
RPAREN   : ')';
MULT     : '*';
CONV     : "**";
PLUS     : '+';
MINUS    : '-';
DIV      : '/';
SEMI     : ';';
LBRACE   : '{';
RBRACE   : '}';
LBLK     : '[';
RBLK     : ']';
ASGN     : '=';
COMMA    : ',';
GE       : ">=";
LE       : "<=";
GT       : '>';
LT       : '<';
EQUI     : "==";
NEQUI    : "!=";

MULTI_COMMENT
         :   "/*" ( options {greedy=false;} : (NL) {newline();}
         | ~( '\n' | '\r' ) )* "*/" {$setType(Token.SKIP); }
         ;

LINE_COMMENT
         : "//" ( options {greedy=false;} : . )* '\n'
              {$setType(Token.SKIP); newline(); }
         ;
```

B Parser

```
class EzipParser extends Parser;

options{
    k = 2;
    buildAST = true;
    exportVocab = ezipVocab ;
}
```

```
tokens{
     DECLARATION;
     STATEMENT;
     FOR_CON;
     FORALL_CON;
     FUNC_CALL;
     VAR_LIST;
     MATRIX;
     ROW;
     EXPR_LIST;
     POSITIVE;
     NEGATIVE;
     INDEX;
}
program
        : ( statement | func_def )* EOF!
            {#program = #([STATEMENT,"PROG"], program); }
        ;

func_def
        : "function" (type)? ID LPAREN! var_list RPAREN! func_body
        ;

var_list
        : type ID ( COMMA! type ID )*
            {#var_list = #([VAR_LIST,"VAR_LIST"], var_list); }
        |
            {#var_list = #([VAR_LIST,"VAR_LIST"], var_list); }
        ;

func_body
        : LBRACE! (statement)* RBRACE!
            {#func_body = #([STATEMENT,"FUNC_BODY"], func_body); }
        ;

statement
          : decl_stmt
          | if_stmt
          | for_stmt
          | forall_stmt
          | while_stmt
          | break_stmt
          | continue_stmt
          | return_stmt
          | assign_stmt
          | func_call_stmt
          | block
             {#statement = #([STATEMENT,"STATEMENT"], statement); }
          ;

type
          : "image"
          | "kernel"
          | "int"
          | "double"
          | "boolean"
          ;

decl_stmt
          : type ID^ ( ASGN! expression )? ( COMMA! ID ( ASGN expression )? )* SEMI!
             {#decl_stmt = #([DECLARATION,"DECLARATION"], decl_stmt); }
          ;


block     : LBRACE! (statement)* RBRACE! ;

if_stmt
          : "if" LPAREN! expression RPAREN! statement
            (options {greedy = true;}: "else"! statement )?
          ;

for_stmt
          : "for" for_con statement
          ;

for_con
          : LPAREN! expression SEMI! expression SEMI! expression RPAREN!
             {#for_con = #([FOR_CON,"FOR_CON"], for_con); }
          ;

forall_stmt
          : "forall" forall_con statement
          ;

forall_con
          : LPAREN! ID "in"! NUM "to"! NUM RPAREN!
             {#forall_con = #([FORALL_CON,"FORALL_CON"], forall_con); }
          ;

while_stmt
```

```
          : "while" LPAREN! expression RPAREN! statement
          ;

break_stmt
          : "break" SEMI!
          ;

continue_stmt
          : "continue" SEMI!
          ;

return_stmt
          : "return" (expression)? SEMI!
          ;

assign_stmt
          : element ASGN expression
          ;

element   : ID^ ( LBLK! NUM COMMA! NUM RBLK! )?
          ;

func_call_stmt
          : func_call SEMI!;

func_call
          : ID LPAREN! expr_list RPAREN!
              {#func_call = #([FUNC_CALL,"FUNC_CALL"], func_call); }
          ;

expr_list
          : ( expression (COMMA! expression)* )?
              {#expr_list = #([EXPR_LIST,"EXPR_LIST"], expr_list); }
          ;

expression
          : not_expr ( ( "AND"^ | "OR"^ ) not_expr )*
          ;

not_expr
          : ("NOT"^)? rltn_expr
          ;

rltn_expr
          : add_expr ( ( GE^ | LE^ | LT^ | GT^ | EQUI^ | NEQUI^ ) add_expr )?
          ;

add_expr
          : mult_expr ( ( PLUS^ | MINUS^ ) mult_expr )* ;

mult_expr
          : conv_expr ( ( MULT^ | DIV^ ) conv_expr )* ;

conv_expr
          : sign_atom ( (CONV^) sign_atom )*
          ;

sign_atom
          : PLUS! atom
              {#sign_atom = #([POSITIVE,"POSITIVE"], sign_atom); }
          | MINUS! atom
              {#sign_atom = #([NEGATIVE,"NEGATIVE"], sign_atom); }
          ;

atom
          : element
          | NUM
          | func_call
          | matrix
          | "true"
          | "false"
          | LPAREN! expression RPAREN!
          ;

matrix    : LBLK! row (COLON! row)* RBLK!
              {#matrix = #([MATRIX,"MATRIX"], matrix); }
          ;

row       : NUM (COMMA! NUM )*
              {#row = #([ROW,"ROW"], row); }
          ;
```

C References

- Antlr 2.0 Documentation *http://www.antlr2.org/doc/index.html*
- Antlr 2.0 Tutorial *http://www.javadude.com/articles/antlrtut/*
- MX Source Code *http://www1.cs.columbia.edu/ sedwards/classes/2003/w4115/mx030521.zip*

- C Reference Manual *http://cm.bell-labs.com/cm/cs/who/dmr/cman.pdf*
- ANSI C grammar for ANTLR v3 http://www.antlr2.org/grammar/1153358328744/c.g