IPL Language Reference Manual

Jianning.Yue
Wookyun.Kho
Youngjin.Yoon

1. General Information

IPL is a computer language that enables users to easily describe and show the pictures and decent animations, and create those images by using the functions such as image displaying, rotation, color modification, or repeat of images.

Moreover, users easily can make simple animations such as movement of images from location A to B, and making an animation with many images.

By using IPL, users can define (or import) images, and with these images, users can display and modify their own image very easily.

2. Lexical Conventions

There are six kinds of tokens: identifiers, keywords, constants, strings, expression operators, and other separators. In general blanks, tabs, newlines, and comments as described below are ignored except as they serve to separate tokens. At least one of these characters is required to separate otherwise adjacent identifiers, constants, and certain operator-pairs.

If the input stream has been parsed into tokens up to a given character, the next token is taken to include the longest string of characters which could possibly constitute a token.

2.1 Comments

The characters /* introduce a comment, which terminates with the characters */.

2.2 Identifiers (Names)

An identifier is a sequence of letters and digits; the first character must be alphabetic. The underscore „„_“ counts as alphabetic. Upper and lower case letters are considered different.

2.3 Keywords

The following identifiers are reserved for use as keywords, and may not be used otherwise:

```
break
continue
if
while
return
defunc
```

2.4 Constants

There is only one kind of constants: integer constants. An integer constant is a sequence of digits.

2.5 Strings

A string is a sequence of characters surrounded by double quotes " `` ". A string is only used as a filename. (In the future, displaying strings with images might be possible)

3. Syntax notation

In the syntax notation used in this manual, syntactic categories are indicated by *italic* type, and literal words and characters in gothic. Alternatives are listed on separate lines. An optional terminal or nonterminal symbol is indicated by the subscript "opt," so that

{ $expression_{opt}$ }

would indicate an optional expression in braces.

4. What's in a Name?

IPL bases the interpretation of an identifier upon contents of the identifier: image, constants, coordination, or list. If variables are invoked inside of a function, variables are local to each invocation of a function, and are discarded on return. Independently of invocations of the function; external variables are independent of any function.

IPL supports four fundamental types of objects: integers, strings, images, and coordinations,

Besides the four fundamental types there is a conceptually infinite class of derived types constructed from the fundamental types in the following ways:

lists of objects of most types;

functions which return objects of a given type;

In general these methods of constructing objects can be applied recursively.

5. Objects and lvalues

An object is a manipulatable region of storage; an lvalue is an expression referring to an object. An obvious example of an lvalue expression is an identifier. The name "value" comes from the assignment expression ''E1 = E2'' in which the left operand E1 must be an lvalue expression. The discussion of each operator below indicates whether it expects lvalue operands and whether it yields an lvalue.

6. Conversions

In IPL, conversion is not available.

7. Expressions

The precedence of expression operators is the same as the order of the major subsections of this section (highest precedence first). Thus the expressions referred to as the operands of + (§7.5) are those expressions defined in §§7.1_7.4. Within each subsection, the operators have the same precedence. Left- or right-associativity is specified in each subsection for the operators discussed therein. The precedence and associativity of all the expression operators is summarized in an appendix.

Otherwise the order of evaluation of expressions is undefined. In particular the compiler considers itself free to compute subexpressions in the order it believes most efficient, even if the subexpressions involve side effects.

7.1 Primary Expressions

Primary expressions involving subscripting, and function calls group left to right.

7.1.1 *identifier*

An identifier is a primary expression. Its type is specified at the first time when it is used.

7.1.2 *constant*

A decimal, octal, or floating constant is a primary expression whose type is integer.

7.1.3 *string*

A string is primary expression. It is formed by a single or a number of characters. String is used for denoting the path name when importing an image object.

7.1.4 *coordination*

A coordination is a primary expression whose type is two identifiers of integer type or constant with brackets, such as [3, a]. It denotes the coordination information on the screen.

7.1.5 *primary-expression* ($primary\text{-}list_{opt}$)

A function call is a primary expression followed by parentheses containing a possibly empty, comma-separated list of expression which constitute the actual argument to the function. The primary expression is of type the function call returns.

7.1.6 *primary-expression* [*primary*]

A primary expression followed by an expression within square brackets is a primary expression. It denotes a list of integers, images or coordinates. The type of expression is determined by what type of the object it stores. The primary-expression is an identifier and the type of primary expression in the bracket should only be arithmetic expression of integers or integer identifiers.

7.1.7 *primary coordination*

A primary expression followed by a coordinates is a primary expression. Its type is image which has been modified with the coordinate operand. The type of the primary should only be image type.

7.2 Unary operator

Expressions with unary operators group right-to-left

7.2.1 ! *expression*

The result of the logical operator ! is 1 if the value of the expression is 0, 0 if the value of the expression is non-zero. The type of the result is integer. This operator is applicable only to integer.

7.3 Image operators

Image operators are used for transforming the show of pictures. It provides the user with easy and direct ways to modify the picture appearance with high flexibility. The operators will return the type image after being transformed according the operands. The image operators includes ^ operator used for scaling pictures, and @ operator used for rotating pictures by how many degrees.

7.3.1 *expression* ^ *expression*

The ^ operator indicates scaling. The first operand must be image type which is the picture needed scaling, and second operand must be integer type denotes how many times the picture will be scaled. Whether enlarging or shrinking the image is determined by the value of the second operand. The result is the image type object that has been scaled.

7.3.2 *expression* @ *expression*

The @ operator indicates rotating. The first operand must be image type which is the picture needed rotating, and second operand must be integer type denotes how degrees the picture will be rotated. The result

is the image type object that has been scaled.

7.4 Multiplicative operators

Arithmetic Operator is used for arithmetic calculation of integers. The binary operators *, /, %, +, - group left-to-right.

7.4.1 *expression* * *expression*

The * operator indicates arithmetic multiplication.

For *expression* * *expression*, the result type is an integer or an integer list. If the operands are both integer type, the result type is integer. If one of the operands is integer, and the other is image, the return type is image.

7.4.2 *expression* / *expression*

The / operator indicates arithmetic division. The same type considerations as for multiplication apply.

7.4.3 *expression* % *expression*

The binary % operator yields the reminder from the division of the first expression by the second. The same type considerations as for multiplication apply.

7.5 Additive operators

The additive operators + and - group left-to-right.

7.5.1 *expression* + *expression*

The + operator indicates arithmetic addition. The result is the sum of the expressions. The same type considerations as for multiplication apply.

7.5.2 *expression* – *expression*

The result is the difference of the expressions. The same type considerations as for addition apply.

7.6 Display operators

Display operators group left-to-right.

7.6.1 *expression* $ *expression*

$ operator indicates the horizontal combination of the two operands. Both operands must be of type image. The operator returns an image. The result will look like first image combine the second abreast.

7.6.2 *expression* # *expression*

operator indicates the vertical combination of the two operands. Both operands must be of type image. The operator returns an image. The result will look like that the first image stands above the second.

7.7 Assignment operators

Assignment operator groups right-to-left. It requires a lvalue as their left operand, and the type of an assignment expression is that of its left operand. The value is the value stored in the left operand after the assignment has taken place.

7.7.1 *lvalue* = *expression*

The assignment operator requires a *lvalue* as its left operand, and the type of an assignment expression is that of its left operand. The value is the value stored in the left operand after the assignment has taken place.
lvalue is an expression referring to an object which is a manipulatable region of storage. The value of the expression replaces that of the object referred to by the *lvalue*. An obvious example of an *lvalue* expression is an identifier.

The = operator supports the automatic type transformation, which means that the *lvalue* type will automatically change to the type of the right operand if the two are different types, which is a default operation.

7.8 Relational operators

The relational operators group left-to-right, the following are the relational operators:

7.8.1 *expression* > *expression* (larger)
7.8.2 *expression* >= *expression* (larger than equal)
7.8.3 *expression* < *expression* (smaller)
7.8.4 *expression* <= *expression* (smaller than equal)

The relational operators all yield 0 if the specified relation is false and 1 if it is true.

7.9 Equality operators

7.9.1 *expression* == *expression*
7.9.2 *expression* != *expression*

The == (equal to) and the != (not equal to) operators are exactly analogous to the relational operators except for their lower precedence. (Thus ''a<b == c<d'' is 1 whenever a<b and c<d have the same truth value).

7.10 Logical operators

7.10.1 *expression* | *expression*

The | operator groups left-to-right. The operands must be integer type. The result is an integer which is the bit-wise inclusive or of its operands.

7.10.2 *expression* & *expression*

The & operator groups left-to-right. Both operands must be integer type. The result is an integer which is the bit-wise logical and of the operands.

8. Statements

Except as indicated, statements are executed in sequence. Statements can be categorized as defunc statements and others, named as normal statements. Normal statements tells what should be executed in IPL, while defunc statements describe what the specific function does in IPL. Defunc statements starts with defunc words, while normal statements does not. It is described as follows:

statement :
- *expression*
- { *statement-list* }
- `if` (*primary*) *statement*
- `while` (*primary*) *statement*
- `defunc` *func_identifier* (*param-list*$_{opt}$) *return-identifier statement*
- `break;`
- `continue;`

statement-list :
- *statement*
- *statement statement-list*

param-list :

identifier
identifier param-list

return-identifier :
identifier
VOID

8.1. Normal statements

Normal statements can be divided into expressions, conditional statements, loop statements, break statements and continue statements. Because each statement will be executed by sequence in IPL, it is important to control the various data using conditional statements and loop statements.

8.1.1. Expression statement

Most statements are expression statements, which have the form:

expression ;

It is defined in chapter 7. Usally expression statements are assignment or function calls.

8.1.2 Conditional statement

Conditional statement can be described as follows:

`if` (*primary*) *statement*

It executes one or more statements if the return value of *primary* in parenthesis is true. *statement* after close parenthesis describes what to do if the primary is true. It can be expressed as both single statement or, multiple statements surrounded by brace such as:

{ *statement-list* }

statement-list can be defined as:

statement-list :

statement

statement statement-list

8.1.3. Loop statement

Syntactically, there is only one loop statement in IPL. It is while loop expressed as:

`while` (*primary*) *statement*

It executes statements in braces iteratively until the return of *primary* in parenthesis becomes false. *statement* after close parenthesis describes what to do while the primary is true. It can be expressed as both single statement or, multiple statement surrounded by brace such as:

{ *statement-list* }

statement-list can be defined as:

statement-list :

statement

statement statement-list

As mentioned, the IPL does not support „for loop“ in C. However, anything with for loop can be done by while loop such as:

`for` (*i=0*; *i<3*; *i++*) {<*statements*>} - `for` *loop in C*

i=0; `while` (*i<3*) {<*statements*> *i++*;} - *in* IPL *with* `while` *loop.*

8.1.4. `break` statement

The statement

```
break;
```

causes termination of the smallest enclosing while statement; control passes to the statement follow the terminated statement.

8.1.5. `continue` statement

The statement

```
continue;
```

causes control to pass to the loop-continuation portion of the smallest elosing while statement; the is to the end of the loop.

8.2. `defunc` statement

Defunc statement starts with „`defunc`" reserved words which represents the definition of the function. It is free where you define function using `defunc` in IPL. However, the function should be defined before used in IPL. If you don"t, It will cause the error. The syntax of `defunc` can be described as:

`defunc` *func_identifier* (*param-list*$_{opt}$) *return-identifier statement*

It defines the *func_identifier* as a name of function which has *param-list* as an optional parameters with parenthesis and *return_value* as a return value. Moreover, *statement* after *return-identifier* represents what to do in the function. It should have *return_value* at least once so that the function returns the last updated value of the *return_value*.

8.3. Miscellaneous

param-list represents the parameter list for the function. It can be represented only one or more variables as follows:

param-list:

identifier

identifier param-list

return_value represents the return value of the function. It can be represented only one variable or empty if not needed defined as:

return-identifier:

identifier

VOID

9. Scope rules

Because IPL allows a block-structure surrounded by braces in case of `defunc`, `if` and `while` statements, it is important to define the scope of the variables. Because there is no declaration for variables, variables in braces is restricted as local variable in brace only. For example, following causes an error because imgB defined as a local scope:

```
imgA = "./bar.jpg";
if (numA > 3)
{
	imgB = "./foo.jpg";
}
print(imgB);
```

It also is applied between languages imported from file using a load function and languages written in interpreter. For example, suppose you type such as:

```
imgA = "./bar.jpg";
load("foo.ipl");
imgC = imgA + imgB;
```

While in foo.ipl:

```
if (numA > 3)
{
	imgB = "./foo.jpg";
}
```

As a result, it causes an error in line 3 because it is treated as same as follows:

```
imgA = "./bar.jpg";
if (numA > 3)
{
	imgB = "./foo.jpg";
}
imgC = imgA + imgB;
```

10. Boolean expressions

In several places in IPL requires expressions which evaluate true or false: in if statements and loop(while) statements. In IPL, constants and variables consisting only single integer are only allowed to compare with operator such as:

```
|   &   ==   !=   <   >   <=   >=
```

Or by the unary operators:

```
!
```

Even there is no such a boolean type in IPL, the result of those operation is both true or false. As a result, in IPL, those expression is allowed only in parenthesis of if statements and loop(`while`) statements.

Parenthesis can be used for grouping, but not for function calls.

11. Examples

11.1 Example 1 – `defunc` and `while`

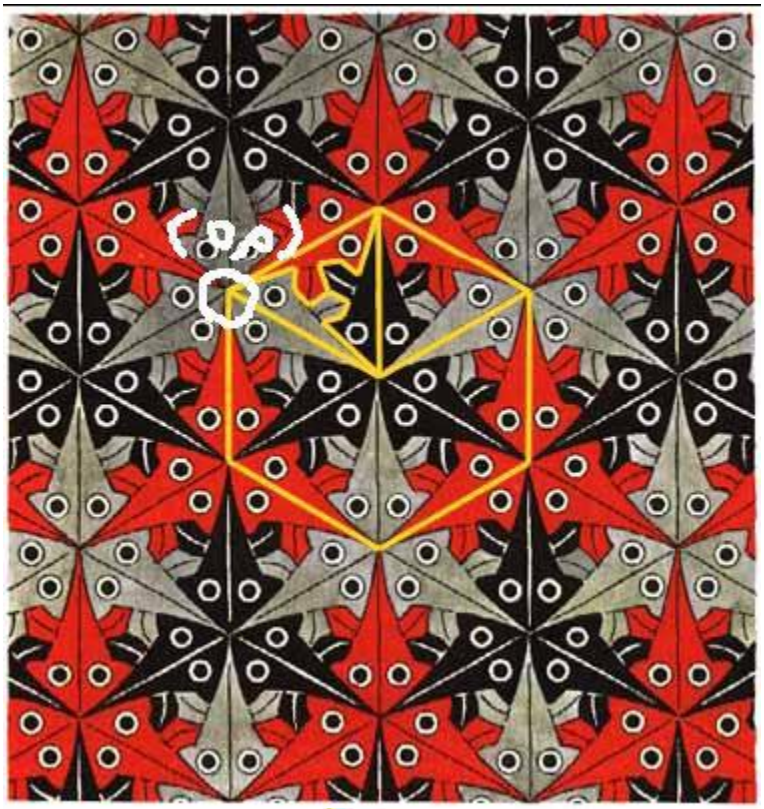
```
defunc fall3 (imgA startTime endTime) imgB
{
	coordA = <xof(imgA),yof(imgA)-3>;
	imgB = move (imgA coordA startTime endTime);
}

imgA[0] = "./small_circle.jpg";
imgB = "./triangle.jpg";
```

```
imgC = "./rect.jpg";
imgA[1] = "./mid_circle.jpg";
imgA[2] = "./big_circle.jpg";

numA = 0;
while(numA < 3)
{
       imgA[numA] = imgA[numA] <numA,numA> @ numA * 30;
       numA = numA + 1;
}

imgC = imgA $ imgB;
imgC = imgC<-3,4>;
imgD = fall3 (imgC[0] 1 3);

print(imgD);
```

define a function which moves image down to 3 point as fall3. imgA is defined as a list of circles, while imgB is triangle and imgC is rectangle. Rotate each element of imgA with 0, 30, and 60. Then combine imgA and imgB as imgC. After creating imgC, imgC set to (-3,4) coordination and call the fall3 function with imgC[0], 1, 3. return value is assigned to imgD and it will be shown by print() function.

11.2 Example 2 – Escher picture

```
imgA = "./lizardWhite.jpg";
imgB = "./lizardBlack.jpg";
imgC = (imgA $ (imgB @-90)) # (imgB @90 $ imgA @180);
imgD = (imgC @180 # imgC @180) $ (imgC # imgC);
print(imgD)
```



img1

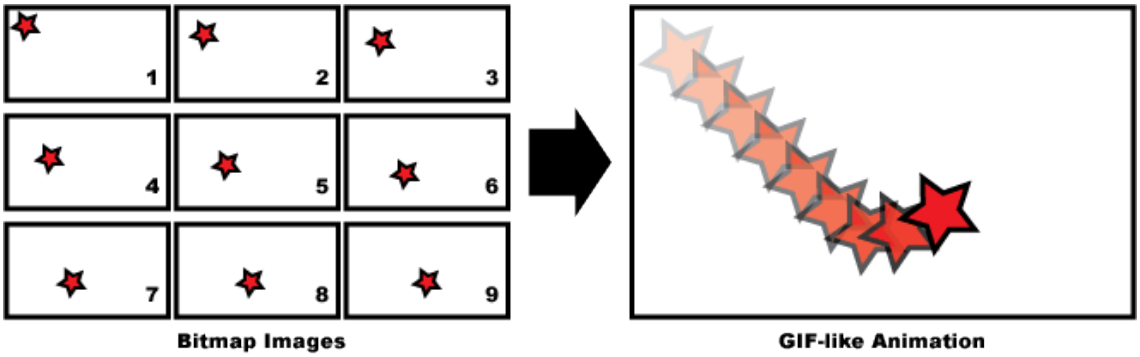
img2

(Assume that each triangle with edge 1)

```
img1 = "./fishL.jpg"
img2 = "./fishR.jpg"
imglist[1] = img1 <0,0>
imglist[2] = img2 <squr(3)/2, -1/2>
imglist[3] = img2 <0,0> @ 120
imglist[4] = img1 < squr(3)/2, -1/2> @ 120
imglist[5] = img1<0, -1> @ -120
imglist[6] = img2 <….
defunc showall ( imagelist[] ) imgX
{
       intA = 1;
       imgX=null;
       while (intA<sizeof(imglist[]))
       {
              imgX = combine (imgX, img[intA]);
       }
}
img = showall (imglist[]);
```

```
imglist[1] = img <0,0>
……
```

11.3 Example 3 – animation



If you want to display this Bitmap Images like an animation, you can easily use the animation function, which is one of the built-in functions in IPL:

```
imgA[0] = "./ani00.jpg";
imgA[1] = "./ani01.jpg";
imgA[2] = "./ani02.jpg";
imgA[3] = "./ani03.jpg";
imgA[4] = "./ani04.jpg";
imgA[5] = "./ani05.jpg";
imgA[6] = "./ani06.jpg";
imgA[7] = "./ani07.jpg";
imgA[8] = "./ani08.jpg";
imgB = animate imgA 0.3;
```

APPENDIX 1
Syntax Summary

1. Expressions

expression:
primary
lvalue asgnop primary

primary:
identifier
constant
string
coordination
primary (*primary-list*$_{opt}$)
primary [*primary*]
primary coordination
primary binop primary

lvalue:
identifier
primary [*primary*]

coordination:
< *primary* , *primary* >

The primary-expression operators

[ ] { }

have highest priority and group left–to right

Unary operator

!

has priority below the primary operators but higher than any binary operator, and groups right-to-left. Binary operators and the conditional operator all group left-to-right, and have priority decreasing as indicated:

binop:

/	divide	%	mod	*	multiply
+	plus	-	minus		
^	Scale	@	Rotate		
$	Horizontal	#	Vertical		
<	>	<=	>=		
==	!=				
|	&				

Assignment operator has the same priority, and groups right-to-left.

asgnop:
=

The comma operator has the lowest priority, and groups left-to-right.

statement:
expression
{ *statement-list* }

`if` (*primary*) *statement*
`while` (*primary*) *statement*
`defunc` *func_identifier* (*param-list*$_{opt}$) *return-identifier statement*
`break;`
`continue;`

statement-list:
statement
statement statement-list

param-list:
identifier
identifier parameter-list

return-identifier:
identifier
VOID