The Specification of the IMG Language

The Build-it, Break-it Problem Definition

January 11, 2014

1 Problem overview

In this contest, you are required to implement an interpreter for an image rendering language, IMG, which we specify below. At a high level, IMG is a domain-specific language for creating scenes that consist of 2D shapes. An IMG program creates shapes (such as lines, ellipses, rectangles, etc.) and can manipulate them by grouping, ungrouping, scaling, translating, etc. In IMG, these domain-specific actions are mixed with program execution (e.g., loops and conditional control flow).

Conceptually, the final result of interpreting a IMG program is a rendered scene. Concretely, the final result of running the IMG interpreter on a (legal) IMG program is a file in SVG format, as defined here:

http://www.w3.org/TR/SVG/

The IMG interpreter that you will implement consists of a program imgrun invoked from the command line shell as follows:

```
$ imgrun stickman.img 20 20
```

The program expects three arguments: an input file (stickman.img above) and two positive integers. The input file consists of a IMG program, whose semantics is described by the remainder of this document. The integers give the width and height of the viewBox to render in the final SVG, as defined here:

http://www.w3.org/TR/SVG/coords.html#ViewBoxAttribute

We give an example of running IMG to produce an SVG file. First, consider the following example IMG program:

Listing 1: stickman.img

```
def main ( w , h ) {

  var head;  var neck;
  var torso; var arms;
  var legl;  var legr;
```

```
  head  = drawEllipse(10,5,4,3);
  neck  = drawLine   (10,8,10,10);
  torso = drawLine   (10,10,10,12);
  arms  = drawLine   (5,10,15,10);
  legl  = drawLine   (6, 14,10,12);
  legr  = drawLine   (14,14,10,12);
}
```

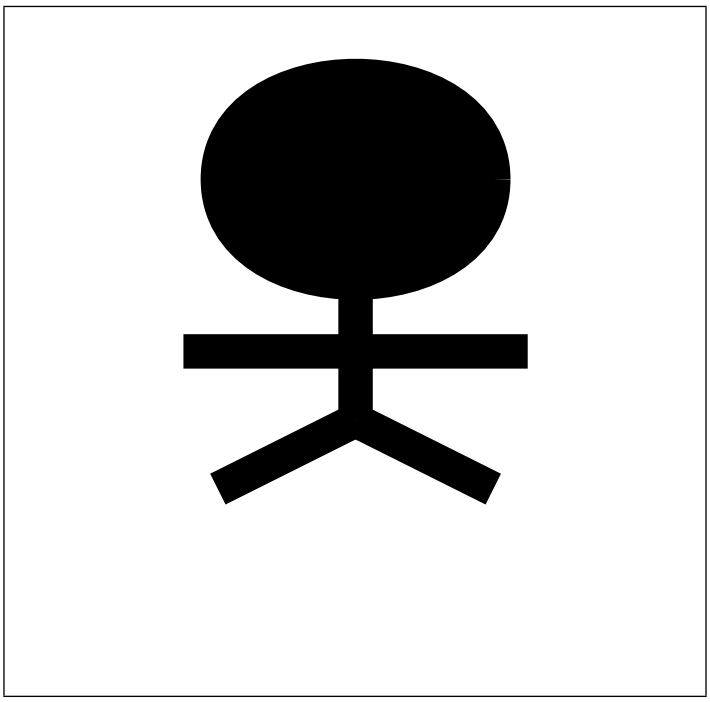
The program consists of one *procedure*, named `main`. The procedure is parameterized by two formal arguments (`w` and `h`) that will be bound to the width and height supplied on the command line (20 and 20, respectively, in the example usage shown above). The `main` procedure consists of some variable declarations, followed by a series of assignments to the declared variables. Each assignment statement shown above creates a *shape* (there is one ellipse and five lines) and binds the shape to a declared variable. Upon the completion of running `main`, the interpreter `imgrun` produces a *final scene* that consists of the collection of shapes that are assigned to the variables of `main` upon the completion of its execution. In the example shown, the final scene consists of the six shapes named by the six variables that are declared and defined (viz., `head`, `neck`, `torso`, `arms`, `legl` and `legr`).

The command line invocation `imgrun stickman.img 20 20` results in the following SVG description being produced on *standard output* (viz., `stdout` in C):

Listing 2: Output of running "`imgrun stickman.img 20 20`"

```
<?xml version="1.0"?><svg viewBox="0 0 20 20"
xmlns="http://www.w3.org/2000/svg" version="1.1">
<g stroke="black" fill="black">
<line x1="14" y1="14" x2="10" y2="12" />
<line x1="6" y1="14" x2="10" y2="12" />
<line x1="5" y1="10" x2="15" y2="10" />
<line x1="10" y1="10" x2="10" y2="12" />
<line x1="10" y1="8" x2="10" y2="10" />
<ellipse cx="10" cy="5" rx="4" ry="3" />
</g></svg>
```

The SVG output is rendered as follows:



To simplify various aspects of this system, all scenes defined by `IMG` programs are monochrome (viz., black foreground, transparent background). Likewise, all the shapes defined by `IMG` programs consist of black stroke and fill. A consequence of these simplifications is that the order of shapes does not affect the final perceived image, since with all black shapes, one cannot distinguish different orderings of occlusion. We define a more precise notion of image equivalence in Section B.

Program	P	::=	$p\ P \mid p$
Procedure	p	::=	$\texttt{def}\ f(\bar{x})\ \{S\}$
Statement	S	::=	$\texttt{var}\ x\,; \mid e_1 = e_2\,; \mid \texttt{if}(e)\ S \mid \texttt{while}(e)\ S \mid \texttt{return}\ e\,; \mid S_1\ S_2 \mid \{S\} \mid e\,;$
Expression	e	::=	$(e) \mid n \mid s \mid \texttt{none} \mid \texttt{true} \mid \texttt{false} \mid x \mid [e] \mid e_1.e_2 \mid e_1 \oplus e_2 \mid f(\bar{e})$
String const	s		
Integer const	n		
Binary operator	$\oplus$	::=	$++ \mid + \mid - \mid \times \mid / \mid \% \mid \gg \mid \ll \mid == \mid != \mid > \mid <$
Library procedures	g	::=	$\texttt{drawLine} \mid \texttt{drawEllipse} \mid \texttt{drawBox} \mid \texttt{drawText} \mid \texttt{destroyShape}$ $\mid \texttt{drawLineConnectingShapes} \mid \texttt{drawTextOnShape}$ $\mid \texttt{getShapeXCoordinate} \mid \texttt{getShapeYCoordinate} \mid \texttt{clearScene}$ $\mid \texttt{sin} \mid \texttt{cos} \mid \texttt{tan} \mid \texttt{arcsin} \mid \texttt{arccos} \mid \texttt{arctan}$
Type	τ	::=	$\texttt{int} \mid \texttt{string} \mid \texttt{void} \mid \texttt{bool} \mid \texttt{table} \mid \texttt{shape}$

Figure 1: IMG Language: Abstract syntax

2 The IMG Language: Mandatory features

This section introduces the abstract syntax, typing, and run-time semantics of IMG programs executed by the imgrun interpreter. The IMG language is divided into *mandatory features*, described here, and *optional features*, described in Section 3. For reference throughout, the *abstract syntax* for IMG is shown in Figure 1, and the *exit error codes* for imgrun is shown in Figure 2. The remainder of this section gives details about the semantics of each mandatory language construct, including both legal execution and the conditions under which the imgrun interpreter exits with an error code.

Overview of program structure. Though shown here with various typography, the concrete syntax for IMG is given by ordinary ASCII characters; we refer the reader to Appendix A for specific details. Here, we use the notation given in Figure 1. An IMG program P consists of a (non-empty) collection of procedures. A procedure p consists of a sequence of formal parameters (written $\bar{x}$) and a statement body (written S). Every program has a special procedure named main that serves as the entry point for the program's execution, as introduced in the previous section (more on main later). It is mandatory to support programs with a single procedure (viz., main); an optional feature (Section 3) consists of supporting multiple, mutually-recursive procedures. Statements perform variable declaration, variable assignment, and control flow via standard constructs like if and while. Control flow conditions and variable assignments are parameterized by expressions, each written as e. Among its expressions, IMG provides library procedures g to create, group and transform shapes.

2.1 Statements

This section explains the syntax of statements in IMG, along with how they are executed by the IMG interpreter.

Code	Error condition
10	When the input `IMG` program is not syntactically valid. Refer to Appendix A for details.
20	When a run-time type error or procedure-arity error occurs.
30	When the program uses a shape destroyed earlier by `destroyShape`.
40	When the program creates a table with negative size
50	When the program uses a variable before or without a corresponding declaration.
60	When the program declares a variable that is already declared.

Figure 2: `imgrun`: Error codes on exit, and their conditions.

2.1.1 Variable declaration

The statement `var` x declares a variable x in a procedure. The variable's value is initialized to `none`. A variable can only be declared once. If x has already been declared, the interpreter exits with **error code 60** (see Figure 2). Every variable must be declared before it's used. If a variable name is used without a corresponding legal declaration, the interpreter exits with **error code 50** (see Figure 2). The dynamic extent of a variable declaration is the procedure body in which the declaration appears.

2.1.2 Assignment statements

Statement $e_1 = e_2$ is an assignment statement in `IMG`, where e_1 and e_2 are expressions. The interpreter first checks that e_1 is an *lvalue*, i.e. e_1 should either be a variable x (variable assignment), or a table index expression $e_3.e_4$ (table update).

Variable assignment. The interpreter evaluates e_2 to value v_2. If v_2 is a non-table value, the interpreter creates a copy of v_2 and assigns it to x. For example, if v_2 is a line shape, the interpreter creates a new line same as v_2 and assigns it to x. If v_2 is a table, then the interpreter points x to the same table, i.e. assignment of a table *does not* create a new table.

Table update. The interpreter evaluates e_3 to v_3 and checks that it's a table value. The interpreter then evaluates e_4 to value v_4, and the RHS of the assignment e_2 to v_2. Table updates differ from variable assignments in only one respect: even shape values (like table values) are made to point to the same shape, rather than creating a new one. So, the key v_4 in table v_3 is mapped to v_2 (or a copy of v_2). Any earlier binding of v_4 in v_3 is overridden by this new binding.

2.1.3 Conditional statements

`IMG` consists of conditional statements of the form `if`(e) S. To evaluate a conditional statement, the interpreter first evaluates e to v and checks that it is a `bool` value. If v is `true`, the interpreter executes S, else it continues to the next statement.

2.1.4 Loop statements

`IMG` has looping statements `while`(e) S. To evaluate these, the interpreter first evaluates e to v and checks that it is a `bool` value. If v is `true`, the interpreter executes S, and

then executes `while`(e) S again. If v is `false`, the interpreter continues to the next statement.

2.1.5 Return statements

Return statements `return` e are used to return a value from a procedure. The interpreter evaluates e to a value v, and returns it. The programmer may omit return statements; where missing, they are implicitly assumed to be returning the expression `none`.

2.1.6 Sequence statements

Sequence statements S_1 S_2 are evaluated by the interpreter in the order of S_1 followed by S_2.

2.1.7 Statement blocks

Statements blocks $\{S\}$ are executed by executing S. Blocks do not control or delimit variable scope in any way. Variables declared in the block are accessible after it ends.

2.1.8 Expression statements

An expression statement e is executed by evaluating the expression e and then ignoring its value.

2.2 Expressions

This section explains the syntax of expressions in the language, along with how they are evaluated by the `IMG` interpreter. For each expression, we specify the checks performed by the interpreter. It is implicit througout that anywhere a type-check fails, the interpreter exits immediately with **error code 20** (see Figure 2).

2.2.1 Constants and values

Constants in the language consist of integer constants n, string constants s, a `void`-typed constant `none`, and boolean constants `true` and `false`. Values in `IMG` also include table values (of type `table`), and implicitly assumed to be returning the value `none`.

Integer values. Integers in `IMG` are signed and consist of a fixed width of 32 bits. Overflow or underflow (e.g., by adding large numbers or subtracting large numbers, respectively) does *not* cause an error, but instead wraps around without error, as in the language C.

Table values. An `IMG` program can create and manipulate *tables* which are mappings from keys to values. A table may contain keys and values of *any* type. For example, a table may map integer 1 to string "*hello*" and string "*hi*" to boolean `true`. See Section 2.2.3 for the semantics of tables.

Shape values. Shapes in `IMG` can be lines, ellipses, boxes, or text. A line is described by its x and y coordinates (x_1, y_1) and (x_2, y_2). An ellipse is described by its center coordinate (x, y), and horizontal radius a and vertical radius b. A box is described by the coordinate of its top left corner (x, y), and width w and height h. A text shape is described by its starting coordinate (x, y), and a string s for the contained text. Shapes are created and manipulated in `IMG` using library functions described later.

2.2.2 Variables

A variable x evaluates to its currently-assigned value. Variables must be declared before being used, otherwise the interpreter exists with **error code 50** (see Figure 2).

2.2.3 Tables

Table creation. Expression $[e]$ creates a table. The interpreter evaluates e and checks that it evaluates to a non-negative integer. Otherwise, it exits with **error code 40** (see Figure 2). It then creates a new table with initial size being the value of e.

Table read. Expression $e_1.e_2$ reads a table entry. The interpreter first evaluates e_1 and checks that it evaluates to a table. The interpreter then evaluates e_2 to a value v_2, and returns the mapping of v_2 in the table. If v_2 is not mapped in the table, the interpreter exits with **error code 20** (see Figure 2).

Table update. Tables are updated via assignment statements. See Section 2.1.2.

2.2.4 Binary operations

`IMG` has binary operation expressions of the form $e_1 \oplus e_2$. `IMG` does *not* specify an operator precedence, meaning that expressions such as $10-7/8$ are ambiguous, since it evaluates to either 10 or 0 depending on how the operations are assigned precedence. Rather than define a standard precedence, `IMG` programs must use parenthesis to disambiguate. The interpreter first evaluates e_1 to a value v_1 and e_2 to a value v_2. The operators are then applied as specified in Figure 3.

2.2.5 Procedure calls

Expression $f(\bar{e})$ is used to invoke a procedure f with arguments $\bar{e}$, which we write as an abbreviation for zero or more expressions. The interpreter first evaluates all the argument expressions to values $\bar{v}$. The interpreter checks that number of values in $\bar{v}$ is same as the number of arguments expected by f. The interpreter then assigns the corresponding formal parameters of f to values from $\bar{v}$. For example, if a procedure with signature `foo(a, b, c)` is invoked as `foo(2, 3, 4)`, the interpreter will execute `a = 2; b = 3; c = 4;` . The interpreter then executes the body of f. If the invocation of f terminates in a `return` e_1 statement, the corresponding value is returned as the final value of $f(\bar{e})$ (see executing return statements below). If the invocation of f terminates in some statement other than `return` statement, then the procedure returns `none` to its caller.

2.2.6 Library procedures

`IMG` has a number of library procedures that a program can use to create and manipulate shapes and the scene. The semantics of these procedures is given below:

drawLine(e_1,e_2,e_3,e_4)**.** The interpreter evaluates e_1 to v_1, e_2 to v_2, e_3 to v_3, and e_4 to v_4. The interpreter checks that each of the values is an integer value. The result is a new line shape with $x_1 = v_1$, $y_1 = v_2$, $x_2 = v_3$, and $y_2 = v_4$. If `drawLine` is invoked with < 4 or > 4 arguments, the interpreter exits with a type error.

drawBox(e_1,e_2,e_3,e_4)**.** The interpreter evaluates e_1 to v_1, e_2 to v_2, e_3 to v_3, and e_4 to v_4. The interpreter checks that each of the values is an integer value. The result is a new box shape with top left coordinate $x = v_1$, $y = v_2$, and width as v_3 and height as

$\oplus$	v_1's type	v_2's type	Result type	Explanation
$++$	`string`	`string`	`string`	The concatenation of the arguments.
$+$	`int`	`int`	`int`	sum of v_1 and v_2
$+$	`shape`	`int`	`shape`	a new shape same as v_1, except y-coordinates moved up (positive y) by v_2
$-$	`int`	`int`	`int`	subtraction of v_2 from v_1
$-$	`shape`	`int`	`shape`	a new shape same as v_1, except y-coordinates moved down (negative y) by v_2
$\times$	`int`	`int`	`int`	multiplication of v_1 and v_2
$\times$	`shape`	`int`	`shape`	a new shape same as v_1, except x and y-coordinates scaled up (multiplied) by v_2
$\%$	`int`	`int`	`int`	remainder of dividing v_1 by v_2
$/$	`int`	`int`	`int`	division of v_1 by v_2
$/$	`shape`	`int`	`shape`	a new shape same as v_1, except x and y-coordinates scaled down (divided) by v_2
$\ll$	`int`	`int`	`int`	v_1 left shifted by v_2
$\ll$	`shape`	`int`	`shape`	a new shape same as v_1, except x-coordinates shifted left (negative x) by v_2
$\gg$	`int`	`int`	`int`	v_1 right shifted by v_2
$\gg$	`shape`	`int`	`shape`	a new shape same as v_1, except x-coordinates shifted right (positive x) by v_2
$==$	`int`	`int`	`bool`	`true` if v_1 same as v_2, `false` otherwise
$==$	`string`	`string`	`bool`	`true` if v_1 same as v_2, `false` otherwise
$==$	`bool`	`bool`	`bool`	`true` if v_1 same as v_2, `false` otherwise
$==$	`shape`	`shape`	`bool`	`true` if v_1 is same shape (with same parameters) as v_2, `false` otherwise
$==$	`void`	`void`	`bool`	`true`
$==$	`void`	non-`void`	`bool`	`false`
$==$	non-`void`	`void`	`bool`	`false`
$!=$	_	_	`bool`	negation of $v_1 == v_2$
$<$	`int`	`int`	`bool`	`true` if $v_1 < v_2$, `false` otherwise
$>$	`int`	`int`	`bool`	`true` if $v_1 > v_2$, `false` otherwise

Figure 3: Semantics of binary operators

v_4. If `drawBox` is invoked with < 4 or > 4 arguments, the interpreter exits with a type error.

drawEllipse(e_1, e_2, e_3, e_4)**.** The interpreter evaluates e_1 to v_1, e_2 to v_2, e_3 to v_3, and e_4 to v_4. The interpreter checks that each of the values is an integer value. The result is a new ellipse shape with center coordinate $x = v_1$, $y = v_2$, and horizontal radius as v_3 and vertical radius as v_4. If `drawEllipse` is invoked with < 4 or > 4 arguments, the interpreter exits with a type error.

drawText(e_1, e_2, e_3)**.** The interpreter evaluates e_1 to v_1, e_2 to v_2, and e_3 to v_3. The interpreter checks that v_1 and v_2 are integer values and v_3 is a string value. The result is a new text shape with coordinate $x = v_1$, $y = v_2$, and text as v_3. If `drawText` is invoked with < 3 or > 3 arguments, the interpreter exits with a type error.

drawLineConnectingShapes(e_1, e_2)**.** The interpreter evaluates e_1 to v_1 and e_2 to v_2. The interpreter checks that both the values are shape values. The result is a new line connecting the center of v_1 to the center of v_2. The center of a line, box, and ellipse is as usual. Center of a text shape is defined as its x and y coordinates. If `drawLineConnectingShapes` is invoked with < 2 or > 2 arguments, the interpreter exits with a type error.

drawTextOnShape(e_1, e_2)**.** The interpreter evaluates e_1 to v_1 and e_2 to v_2. The interpreter checks that v_1 is a shape value and v_2 is a string value. The result is a new text shape starting at the center of v_1 and containing the text v_2. If `drawTextOnShape` is invoked with < 2 or > 2 arguments, the interpreter exits with a type error.

getShapeXCoordinate(e)**, getShapeYCoordinate**(e)**.** The interpreter evaluates e to v, and checks that v is a shape value. The result is the x and y coordinate respectively of the center of shape v. If these procedures are invoked with < 1 or > 1 arguments, the interpreter exits with a type error.

sin, cos, tan, arcsin, arccos, arctan. These trignometric functions take one argument each. The interpreter evaluates the argument and checks that it is an integer value. The return value from these functions is the corresponding trignometric function applied to the argument interpreted as degrees. If these functions are called with > 1 or < 1 argument, the interpreter exits with type error.

destroyShape(e)**.** Destroying a shape removes it from the final scene and makes it unusable. The interpreter evaluates e to v, and checks that v is a shape value. Destroying a shape makes it unusable. For example, consider an `IMG` code snippet below:

```
var a; var b;
a = drawLine(10, 20, 30, 40);
destroyShape(a);
b = a;
```

At this point, it's invalid to read `a` at line 4, and so, at line 4, the interpreter exits with **error code 30** (see Figure 2). Another example of an invalid use of destroyed shape is as follows:

```
var a; var b; var c;
```

```
a = drawLine(10, 20, 30, 40);
b = [1];
b.0 = a;
destroyShape(a);
c = b.0;
```

Reading `b.0` at the last line is invalid since when shape `a` is destroyed, that also makes `b.0` invalid (recall when shapes are put in tables, they are not copied). However, following example works fine:

```
var a; var b; var c;
a = drawLine(10, 20, 30, 40);
b = a;
destroyShape(a);
c = b;
```

Destroying `a` does not destroy `b` (recall that assigning a shape to a variable copies it into a new shape).

clearScene()**.** Clearing a scene destroys all the shapes held in all variables.

3 The IMG Language: Optional features

This section gives four optional features that enhance the mandatory features, making IMG more interesting to implement and to program.

3.1 Optional Feature: Dynamic type checks

This optional feature consists of extending IMG with an additional binary operator `instanceOf`, which can be used to dynamically test the types of values:

$$\oplus ::= \cdots \mid e_1 \texttt{ instanceOf } e_2$$

The `instanceOf` operator works as follows: First, it evaluates both operands to values and checks that the second operand is a string. Otherwise, the interpreter exits with **error code 20** (see Figure 2). The operator checks whether the string value of the second operand classifies the value type of the first operand, according to the *value classification hierarchy*, illustrated in Figure 4. If the value of the second operand is a string that correctly classifies the type of the first operand, the operator returns true; otherwise, it returns false.

Examples. In the following listing, the programmer tests the type of a line created by `drawLine`. Each test shown below yields `true` except the last two, which yield `false`; in each case, the test respects the type hierarchy of IMG, as shown in Figure 4.

```
var ln; ln = drawLine(0,1,2,3) ;
var b1; b1 = ln instanceOf "value" ;
var b2; b2 = ln instanceOf "shape" ;
var b3; b3 = ln instanceOf "line" ;
var b4; b4 = ln instanceOf "int" ;
var b5; b5 = ln instanceOf "banana" ;
```

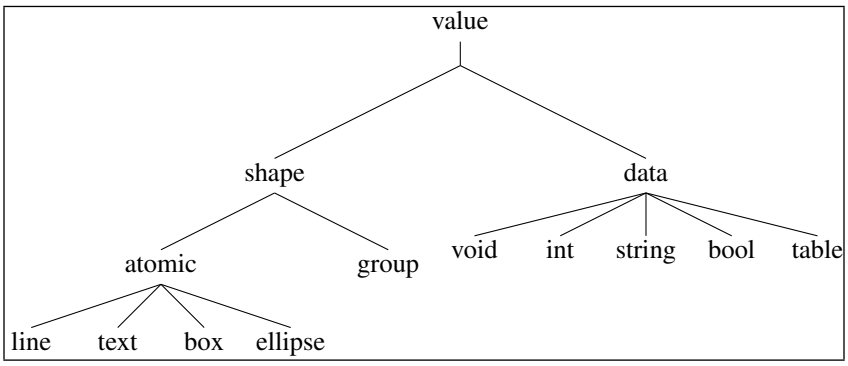


Figure 4: Value classification hierarchy: Organizes `IMG` values into distinct classes.

Above, the variables `b1`, `b2`, and `b3` each contain the value `true`, while `b4` and `b5` each contain `false`.

All `IMG` values are legal test subjects. The following tests use the boolean value `true` as a test subject, resulting in outcomes of `true`, `true`, `true` and `false`:

```
var b6; b6 = b1 instanceOf "value" ;
var b7; b7 = b1 instanceOf "data" ;
var b8; b8 = b1 instanceOf "bool" ;
var b9; b9 = b1 instanceOf "shape" ;
```

Case sensitivity. Notice that the value classifications in Figure 4 as well as the strings used above consist of all lowercase letters. When given other strings for testing, `instanceOf` does not signal an error, but instead merely yields the value `false`.

3.2 Optional Feature: Recursive procedures

This optional feature consists of extending `IMG` with support for multiple, mutually-recursive, programmer-defined procedures. As with `main`, each programmer-defined procedure p consists of a name f and a statement S, using syntax of the form:

$$\texttt{def}\ f(\bar{x})\ \{\ S\ \}$$

An `IMG` procedure f is *programmer-defined* if its name is not `main`. It is an error to define multiple procedures with the same name; when this occurs, the interpreter should exit with **error code 60** (see Figure 2). We refer the reader to Figure 1 and Section A for additional discussion of `IMG` syntax.

Examples. As in other programming languages with "methods" (like Java) or "functions" (like C), programmer-defined procedures in `IMG` allow programmers to abstract and parameterize a collection of statements. Doing so generally promotes greater code reuse. For instance, consider the following listing (cf.., Listing 1):

Listing 3: stickman-proc.img

```
def stickman (x, y, sx, sy) {
  var sm;
  sm = [10];
  sm."head"  = drawEllipse(x+0,y-(sy*5),sx*2,sy*3);
  sm."neck"  = drawLine   (x+0,y-(sy*2),x+0, y);
  sm."torso" = drawLine   (x+0,y       ,x+0, y+(sy*2));
  sm."arms"  = drawLine   (x-(sx*3),y+0,x+(sx*3),y+0);
  sm."legl"  = drawLine   (x-(sx*2),y+(sy*4), x+0,y+(sy*2));
  sm."legr"  = drawLine   (x+(sx*2),y+(sy*4), x+0,y+(sy*2));
  return sm;
}

def main ( w , h ) {
  var sm;
  sm = stickman(w/2, h/2, w/8, h/16);
  var head;  head  = sm."head" ;
  var neck;  neck  = sm."neck" ;
  var torso; torso = sm."torso" ;
  var arms;  arms  = sm."arms" ;
  var legl;  legl  = sm."legl" ;
  var legr;  legr  = sm."legr" ;
}
```

The procedure `stickman` is parameterized by the four parameters `x`, `y`, `sx` and `sy`, which give the position and size of the stickman to construct. The procedure uses a table stored in variable `sm` to collect the shapes of the stickman figure, naming each shape by the corresponding body part of the figure. The procedure returns this table to its caller (`main`), which projects each body part, storing each into a variable that effectively places the shape into the final rendered scene.

Though not shown above, the final scene may consist of many stickmen by making multiple calls to `stickman`. We note that if the procedure `main` calls `stickman` multiple times, it must still project each of the returned shapes into local variables, as illustrated above; shape groups (Section 3.3) overcome this burden by introducing shapes that *contain* other shapes.

Ordering and recursion. The order of programmer-defined procedures does not affect a program's meaning. For instance, the listing above would have the same meaning if the order of `stickman` and `main` were exchanged.

Programmer-defined procedures (but not `main`) may be recursive. For instance, consider this listing:

```
def fib ( n ) {
  if ( n <  0 ) return none ;
  if ( n == 0 ) return 1 ;
  if ( n == 1 ) return 1 ;
  return fib ( n - 1 ) + fib ( n - 2 ) ;
}

def main ( w , h ) {
  var a ; a = drawLine ( fib(1), fib(2), fib(3), fib(4) ) ;
```

```
}
```

The procedure `fib` computes the n^{th} Fibonacci number through self-recursion. Mutual recursion is also permitted. For instance, the recursive procedure `fib` above can be duplicated into two mutually-recursive procedures `fibA` and `fibB`:

```
def fibA ( n ) {
  if ( n <  0 ) return 0-1 ;
  if ( n == 0 ) return 1 ;
  if ( n == 1 ) return 1 ;
  return fibB ( n - 1 ) + fibB ( n - 2 ) ;
}

def fibB ( n ) {
  if ( n <  0 ) return 0-1 ;
  if ( n == 0 ) return 1 ;
  if ( n == 1 ) return 1 ;
  return fibA ( n - 1 ) + fibA ( n - 2 ) ;
}

def main ( w , h ) {
  var a ; a = drawLine ( fibA(1), fibB(2), fibA(3), fibB(4) ) ;
}
```

3.3 Optional Feature: Group shapes

This optional feature consists of adding an additional class of shapes called *group shapes* (or simply *groups* for short). A group is a shape that consists of a collection of *component* shapes. Groups can be used, for instance, for passing collections of shapes between procedures (see Section 3.2) and for rendering large shape collections within the procedure `main` without having to either define long lists of variables.

This feature extends the `IMG` language with an additional library procedure called `drawGroup`:

$$g ::= \cdots \mid \texttt{drawGroup}$$

Unlike the other drawing procedures, `drawGroup` can be called with any number of argument expressions (zero or more), all of which must evaluate to shapes (not integers). If a non-shape is given to `drawGroup`, the interpreter exits with **error code 20** (see Figure 2). Otherwise, the procedure returns a shape value; specifically, it returns a value of class *group*, as specified by Figure 4. We call the shape arguments the *component* shapes of the group.

Component shapes supplied to `drawGroup` are not copied by `drawGroup`; rather, these arguments are possibly *shared* with variables and even other groups. The programmer may use `destroyShape` on a shared component shape, which removes it from all groups in which it is present.

This feature extends each binary operation over shapes given in Figure 3 to support groups, i.e., the operators $+$, $-$, $/$, $\times$, $<<$, $>>$, $==$ and $!=$. In each case, the operator performs the operation over each component shape within the group (transitively, in the

case of a nested group component). As with operators on atomic shapes, translation and scaling operators on groups result in new groups (rather than side-effect existing groups). Moreover, the procedure `destroyShape` destroys group shapes by destroying all of its component shapes (transitively).

This feature extends the statements of `IMG` so that the programmer can iterate over component shapes within a group:

$$S ::= \cdots \mid \texttt{foreach}\ x\ \texttt{in}\ e\ \texttt{do}\ S$$

The semantics of `foreach` is as follows. First, expression e evaluates to a group value. Otherwise, the interpreter exits with **error code 20** (see Figure 2). Next, for each component shape value v within the group, the interpreter runs statement S with variable x bound to v. Conceptually, the interpreter *flattens* groups of groups before iterating over them, meaning that x is never bound to a group shape, but instead only ever takes on the value of *atomic* shapes (viz., lines, boxes, text and ellipses).

Example. For instance, the following example gives a variant of the original stickman example (cf., Listing 1) where a single variable `sm1` stores all the atomic shapes used in the figure, and where a use of `foreach` copies those components into the group shape `sm2`:

Listing 4: Example of using `drawGroup` and `foreach`.

```
def main ( w , h ) {
  var sm1 ;
  sm1  = drawGroup (
           drawEllipse(10,5,4,3),
           drawLine   (10,8,10,10),
           drawLine   (10,10,10,12),
           drawLine   (5,10,15,10),
           drawLine   (6, 14,10,12),
           drawLine   (14,14,10,12)
         );
  var sm2;
  sm2 = drawGroup ( ) ;
  foreach x in sm1 do
    sm2 = drawGroup ( sm2 , x ) ;
  sm2 = ( sm2 + 10 ) >> 10 ;
}
```

3.4 Optional Feature: Dynamic code

This feature consists of adding an additional library procedure called `eval`:

$$g ::= \cdots \mid \texttt{eval}$$

The procedure accepts exactly one argument, which must evaluate to a string value. Other numbers or types of arguments cause the interpreter to exit with **error code 20** (see Figure 2).

The meaning of calling `eval` with a string value is as follows: First, the interpreter parses the string value (as concrete syntax) into either a statement *S* or an expression *e*. If the string fails to parse into one of these forms, the interpreter exits with **error code 10** (see Figure 2). Next, the interpreter evaluates statement *S* or expression *e*, eventually getting a final resulting value (viz., the return value of statement *S*, or the valuation of expression *e*, respectively). Last, the interpreter completes call to `eval` by returning this resulting value.

Some programming languages attempt to restrict when and how variables are accessed by dynamically-created, or dynamically-loaded code. This is not the case in `IMG`. In particular, the code string given to `eval` may use variables defined in the enclosing scope. For comparison, we note that this design decision is consistent with how `eval` works in JavaScript.[1] Here's a simple example of using `eval`:

Listing 5: Example of using `eval` to evaluate a string as a statement.

```
def main ( w , h ) {
  var s;
  eval("s"++"="++"drawLine(0,0,w,h);");
}
```

Notice that the variable names `s`, `w` and `h` appear in the concatenated strings.

A IMG: Concrete Syntax

All characters used in `IMG` programs are 7-bit ASCII. White space tokens (consisting of spaces, tabs and linefeed characters) are relevant for separating the non-whitespace tokens defined below, but are otherwise ignored. In particular, indentation is not semantically relevant. We give the concrete syntax of `IMG` using EBNF notation:

Listing 6: Concrete syntax of `IMG` in EBNF notation.

```
strConst ::= '"' [A-Za-z][A-Za-z0-9]* '"'
intConst ::= ( '-' )? [0-9]+
ident    ::= [A-Za-z][A-Za-z0-9]*
idents   ::= ident | ident ',' idents
identz   ::=  | idents

exp      ::= intConst | strConst | 'none' | 'true' | 'false' | ident |
          | '[' exp ']' | exp '.' exp
          | exp binop exp | ident '(' expz ')' | '(' exp ')'
binOp    ::= '++' | '+' | '-' | '*' | '/' | '%' | '>>' | '<<'
           | '==' | '!=' | '>' | '<'
exps     ::= exp | exp ',' exps
expz     ::=  | exps

stmt     ::= 'var' ident ';'
          | exp '=' exp ';'
          | 'if' '(' exp ')' stmt
          | 'while' '(' exp ')' stmt
```

[1] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/eval

```
           | 'return' exp ';'
           | stmt stmt
           | '{' stmt '}'
           | exp ';'

proc     ::= 'def' ident '(' identz ')' '{' stmt '}'
prog     ::= proc prog | proc
```

There are several freely-available tools that understand this format. For instance, the following online tool can generate so-called "railroad diagrams":

http://bottlecaps.de/rr/ui

When two parsings of statement sequencing are possible, the more shallow (less nested) one is preferred. This leads to behavior similar to `C` in that only the first unnested statement following an `if` or `while` applies.

B `imgrun`: Output Equivalence

An `IMG` program determines a unique output SVG image, modulo two factors that we control for: (1) the precedence of binary operations in the `IMG` program may be ambiguous, and (2) the ordering of shapes in the final SVG file may be arbitrary.

To control for the first factor, we ensure that all `IMG` test programs *parenthesize binary operations* to make their precedence unambiguous.

To control for the second factor, we compare each output SVG file against a standard one *after rasterizing each to a pixel-based representation*. Since all shapes in `IMG` have a uniform color (viz., black stroke and black fill), their rasterization is not influenced by their order in the SVG file. We say that two SVG files are *equivalent* if 99% of their pixels agree, when rendered at high resolution with a monochrome representation (where one pixel is one bit).