The BATS Language Reference Manual

Behrooz Badii(Team Leader)   Aleksandr Borovinskiy
Tanya Shtemberg   Sui Sum Wong

October 28, 2003

Introduction:

The BATS language, a geometric figure drawing language, was designed to be simple to use. Through its simplicity, complex figures can be drawn using complex algorithms. The language also includes loops, conditional statements, and functions to modularize a programmer's language. These pieces of block structured programming are essential for effectively representing geometric algorithms in an understandable format. And since this is a drawing language the choice of declaring, assigning, and using lines, as will be seen, will make drawing vastly easier. BATS will use an interpreter that interprets BATS code into the target language Java.

Syntax (Grammar) notation:

This language reference manual uses a pseudo-ANTLR language to represent the grammar of BATS. Grammars are clearly divided from paragraphing and text to show the grammar exactly. Alternatives are separated by the '|' character. Nonterminals can be placed in an optional set of parentheses, but they are also always italicized. Literals or terminals are placed in single quotes (i.e. '1', 'e'). A set of literals, such as the alphabetic set, can be placed in parentheses, where the first quoted literal is followed by two periods, which is followed by the last literal of the set in single quotes. Optional expressions are placed in parentheses followed by the questions mark character '?'. Iteration is placed in parentheses followed by either the kleene star '*', which means that the parenthesized literals and/or syntactic categories can be matched zero or more times, or the plus character '+', which means that the parenthesized literals and/or syntactic categories must be matched at least once. Epsilon is represented as /*nothing*/. So in the grammar:

Yourname:
    Name ('A' . . 'Z')? (*Name*)?

Name:
    ('A' .. 'Z') ('a'..'z')*

Yourname is the starting symbol. The first name, or Name, is a syntactic subcategory that expands to one uppercase letter followed by any number of lowercase letters. After Name the middle initial is just an optional single uppercase letter. The last name is just Name reiterated to match a last name.

Lexical conventions:

The following are the tokens found in this language: identifiers, keywords, constants, operators, and separators.

Separators:

Blanks (the whitespace), tabs, new lines, carriage returns, and comments are ignored in token creation other than to serve the purpose of separating tokens. The input stream, which is a file containing BATS programs, is parsed so that each token is taken to include the longest string of characters which could possibly constitute a token. This means the parser will keep adding to the token until it hits a whitespace, a tab, a new line, a carriage return, or a comment.

Comments:

The question mark character # introduces a comment, which terminates with a new line. Another convention for comments is starting a comment with the characters #% and ending the comment with %#.

Identifiers (Names)

An identifier is a sequence of letters and digits where the first character must be alphabetic. The underscore ‘‘_’’ is counted as a letter. Upper and lower case letters are considered different. The following is the grammar for an identifier:

Identifier:

(‘A’..’Z’|’a’..’z’|’_’) (‘A’..’Z’|’a’..’z’|’0’..’9’|’_’)*

These are some examples of identifier names:

hello, hello1, _hi3, I_AM_A_VARIABLE

The following cannot be identifiers:

1joe, thr?ee, b|leach

Keywords

Keywords are reserved words that aid the programmer in creating an understandable program that the compiler will accept. These keywords are used for things such as looping, conditional statements, and type declarations. To make them easier to see in a BATS text, they all start with a capital letter. No identifier can have a name identical to the following keywords:

Double	Int	Boolean	Line	Point	From
To	Draw	Call	Function	Begin	End
Start	Terminate	While	For	If	Else
WhileEnd	ForEnd	IfEnd	True	False	Do
Color	Then				

Constants

There are three different types of constants in the BATS language: integer constants, double constants, and Boolean constants. These three are defined as follows:

Integers:

An integer consists of a sequence of ASCII character ‘1’ to ‘9’, and that sequence of ASCII characters represents its real number value. For example, the real number value equivalent of the sequence “123” is 123 or one hundred twenty three. The following is the grammar composing an integer constant:

Integer:
(*Digit*)+

Digit:
(‘0’ . . ’9’)

Doubles:

A double constant is a sequence of digits, followed by a mandatory ASCII decimal or dot ‘.’, which is then followed by another sequence of digits, which is then followed by an optional exponent. The exponent part of a double constant consists of an upper or lower case ‘e’, followed by an optional positive or negative sign, followed by another sequence of digits. The following is the grammar composing a double constant:

Double:
(*Digit*)+ ’.’ (*Digit*)+ (*Exponent*)?

Exponent:
(‘e’|’E’) (‘+’|’-‘)? [*Digit*]+

Notice the mandatory digit in front of the decimal. This makes the double constant found in the BATS language similar to the double values in a regular hand-held calculator. For example, the number .067 is represented as 0.067 in the BATS language, which is similar to the representation of that number in a calculator. The following are examples double constants:

145.167 145.167 E+26 0.12 e-13

The following are examples of input that can’t be a double constant:

145. .167 145. e+12

Booleans:

There are only two Boolean constants in the BATS language, and they are “True” and “False”, representing the logical values of true and false, respectively. The following is the grammar for a Boolean constant:

Boolean:
‘True’|’False’

What an Identifier Stands For

An identifier can stand for another identifier, a constant, or a more complex set of constants. The attributes of an identifier are the storage class and its type. The storage class of an identifier defines the scope of an identifier, that is to say, how long and where the identifier exists through the running of a program. Its type is the type of constant or complex set of constants it stands for.

Currently, there are two storage classes an identifier can choose from to have. The first storage class is global. The identifier exists inside and outside all functions until the program has terminated. A global identifier is declared outside all functions, in the global arena. The second storage class is functional. Here, the identifier exists inside only a specific function, and disappears when the function returns to its caller.

The type of an identifier can be of the following simple constants:

- Integer:
  Integers (*Int*) are represented by an actual integer value in the target language of BATS, Java. Integers can have any value from –2147483648 to 2147483647, inclusive.

- Double:
  Doubles (*Double)* are represented a double-precision floating point variables found in Java. Doubles can have magnitude in the range of approximately $10^{\pm 38}$ or 0. The precision of a double is 17 decimal digits.

- Boolean:
  Booleans (*Boolean*) are represented as the string “true” or “false” in Java. Booleans are primarily used for conditional statements.

The type of a identifier can be a complex set of constants, which are:

- Point
  Points (*Point*) are a set of two double identifiers. These are essentially used for drawing and geometric figures. The following is the grammar of a point:

  Point:
  ‘(‘ *Double* ‘)’ ‘,’ ‘(‘ *Double* ‘)’

- Line

Lines (*Line)* are a set of two or more points. These are the building blocks of BATS and they are important enough to be given a separate section further down in this reference manual.

- Array

  Arrays (*Array*) are a set of one type of identifier. One can have an array of points, doubles, integers, Booleans, points, lines, or even arrays. An identifier can also be an array with an index following it. For example, if we have array a, a[0] would be the index of the first array. Array grammar is described in its own section.

- Functions calls

  Functions (*Function)* calls return an object of a given type. They can return integers, doubles, Booleans, points, or lines. Function definitions are described in its own section.

An *Identifier* must be one of these, and nothing else.

Conversions between identifiers

This section deals with the automatic conversions of identifiers in places where one can be used for another:

Integer to Double

Wherever a double can be used, an integer can also be used. The conversion from an integer to a double does not create any information loss.

Double to Integer

When doubles are converted to integers, there is information loss. Also, doubles cannot be used in the place of integers. For example, array indexes are based on integer values. It makes sense to look for the value at an array index of 2, but it does not make sense to look for a value at an array index of 2.56.

Variables

Variables are expressions that stand for another expression. Expressions are defined below. Some operators yield variables or expect them, and that detail will be described for each operator.

Expressions

The precedence of expression operators will be in descending order through their explanation in this section. Operators in subsections have equal precedence. Note: 1st level precedence is the highest precedence, 2nd level precedence is second highest, and so on.

1st level precedence

Expressions in the highest precedence are grouped right to left.

Identifiers

Identifiers are in this level. An identifier is an expression if it has been properly declared and it has a variable of the proper type assigned to it. The identifier in an expression gives that type.

Array indices

Obtaining information from an array index or indices (in case of a multi-dimensional array) has first level precedence. This is discussed in more detail in the Arrays section

Constants

Constants (Integers, Doubles, Booleans) are in this level.

Parenthetical expression

(*expression*)
An expression with parentheses around it, (*expression*), is at this level. The value of the parenthesized expression is equal to that of the expression inside the parentheses.

Function call

A function call is an expression followed by parentheses containing a list, possibly empty, of expressions that are the arguments of the function. These arguments can be expressions that have to evaluate to the correct type of value in the function definition (discussed in further detail in the Functions section). Arguments are passed by value in function calls, since a copy is made of each actual parameter. Recursive function calls are possible. The following is the grammar for a function call:

Funccall:
    'Call' *expression* '(' *Arglist* ')'

Arglist:
    Expression | *Moreargs*

Moreargs:
‘,’ *expression* | /*nothing*/

Note: the *expression* after ‘Call’ has to be the name of a function.

2nd level precedence

Unary operators are in this level of precedence, and they are grouped right-to-left.

Number Negation

To negate a number, which an integer or double, the following grammar is needed:

Negate:
‘-‘ *expression*

This turns 25 into –25 and –45.7 into 45.7.

Boolean Negation

To negate a Boolean, the following grammar is needed:

NegateBool:
‘!’ *expression*

This turns False into True and True into False

Incrementing

To increment a variable, the variable must first be an integer. The type stays intact, and the value of the integer is one higher than what it previously was. The grammar is:

Increment:
variable’+’’+’

Decrementing

To decrement a variable, the variable must first be an integer. The type stays intact, and the value of the integer is one lower than what it previously was. The grammar is:

Decrement:
variable’-’’-’

3rd level precedence

Multiplicative operators, *,/, and %, are in this level of precedence, and they are grouped left to right.

Multiplication

Only two numbers can be multiplied. If two integers are multiplied, then the product is an integer. If two doubles are multiplied, then the product is a double. If an integer and a double are multiplied, the product is a double. No other combinations are possible. The grammar is:

Multiply:
expression '' expression*

Division

Only two numbers can be divided. If two integers are multiplied, then the product is an integer. If two doubles are multiplied, then the product is a double. If an integer and a double are multiplied, the product is a double. No other combinations are possible. The grammar is:

Divide:
expression '/' expression

Modular division

Only two integers can be used in modular division. It yields the remainder from the division of the first integer by the second. The grammar is:

ModDivide:
expression '%' expression

4th level precedence

Additive operators, + and -, are in this level of precedence, and they are grouped left to right.

Addition

Only two numbers can be added. If two integers are multiplied, then the sum is an integer. If two doubles are multiplied, then the sum is a double. If an integer and a double are multiplied, the sum is a double. No other combinations are possible. The grammar is:

Addition:

expression '+' expression

Subtraction

Only two numbers can be subtracted. If two integers are subtracted, then the subtraction result is an integer. If two doubles are subtracted, then the subtraction result is a double. If an integer and a double are subtracted, the subtraction result is a double. No other combinations are possible. The grammar is:

Subtraction:

expression '-' expression

5th level precedence

Relational operators are in this level. They are grouped right to left.

Less than
Greater than
Less than or equal
Greater than or equal

The operators <, >, <=, and >= evaluate to true if expression on the left operator is less than, greater than, less than or equal to, and greater than or equal to the expression on the right of the operator, respectively. The expression returns false otherwise. The following are the respective grammars for the operators:

Less:

e*xpression* '<' expression

Greater:

e*xpression* '>' expression

LessEqual:

e*xpression* '<''=' expression

GreaterEqual:

e*xpression* '>''=' expression

6th level precedence

The equality operators are in this level of precedence, and they are grouped right to left. So True == False != False has a result of False.

EqualsEquals
NotEquals

The equalsequals operator ‘==’ and notequals ‘!=’ return true when the value of the expression on the left is equal and not equal to the value of the expression on the right, respectively. It returns false otherwise. The grammars are:

EqualsEquals:
expression ‘=’’=’ expression

Notequals:
expression ‘!’’=’ expression

7th level precedence

And

The *And* operator is in this level of precedence. It is used to connect Boolean expressions together. It is grouped right to left. The following is the grammar

And:
expression ‘&’&’ expression

Both expressions must be Boolean. If both sub-expressions return true, the *And* expression returns true, and it returns false otherwise.

8th level precedence

Or

The *Or* operator is in this level of precedence. It is used to connect Boolean expressions together. It is grouped right to left. The following is the grammar

And:
expression ‘|’|’ expression

Both expressions must be Boolean. If both sub-expressions return false, the *Or* expression returns false, and it returns true otherwise.

9th level precedence

Assignment operators, which are found in this level, are grouped right to left. All of them require a variable as their left operand, and the type of that variable being the result of the expression on the right.

Equals

In the Equals expression, the value of the right expression replaces that of the object referred to by the variable in the grammar below. Both the variable type and the expression result on the right must be of the same type (Int, Line, Point, Double, Array, Boolean). The following is the grammar:

Equals:
  v*ariable* '<''-' *expression*

EqualPlus
EqualMinus
EqualMultiply
EqualDivision
EqualModDivision

In the above expressions, the value of the variable plus, minus, multiplied by, divided by, and modularly divided by the expression on the right replaced that of the objected referred to by the variable in the grammar below, respectively. The variable and expression can either both be of type integer, of type double, or the variable can be of type double while the expression is of type integer. No other combinations are allowed. The following are the grammars:

EqualPlus:
  variable '+''<''-'expression
EqualMinus
  variable '-''<''-' expression
EqualMultiply
  variable '''<''-' expression*
EqualDivision
  variable '/''<''-' expression
EqualModDivision
  variable '%''<''-' expression

Declarations

Declarations, specifically declaration lists, are used in the global scope (outside of functions) to declare global variables, in function definitions, and in the first part of a function to declare temporary function variables. However, the BATS programming language has two kind of declaration lists, one specific for global variables, and another for local variables. The following is the grammar for the global declaration list and the local declaration list.

DeclarationsGlobal:
  'Global' *type-specify identifier identifier-list* ';' (*Declarations*)*?*

type-specify:
  'Int'|'Boolean'|'Double'|'Line'|'Point'|
  'Array' ('['*Integer*']')+ *typeforarray*

typeforarray:
‘Int’|’Boolean’|’Double’|’Line’|’Point’

identifier-list:
‘,’ identifer (identifier-list)?

The following is the grammar used for a declaration list as found in a function body.

Declarations:
type-specify identifier identifier-list ‘;’ (*Declarations*)*?*

type-specify:
‘Int’|’Boolean’|’Double’|’Line’|’Point’|
’Array’ (‘[‘*Integer*’]’)+ *typeforarray*

typeforarray:
‘Int’|’Boolean’|’Double’|’Line’|’Point’

identifier-list:
‘,’ identifer (identifier-list)?

How Declared identifiers work

Identifiers that are declared become variables. Each variable that has been declared will yield its type and value or object. So if we have Int x. The type of variable x is Int, which is an integer. Using an assignment operator, we can assign an integer to the variable x. After that assignment, any usage of x will yield that assigned integer.

Statements

Statements are executed in sequence unless a conditional, loop, or function is used. Most statements are expressions with the following grammar:

e*xpression* ‘;’

A statement can be a list of statements, or *statement-list* to be executed:

s*tatement-list*:
statement (*statement-list*)?

Conditional Statements

The grammar of a conditional statement is:

IfBlock:

‘If’ *expression Then statement* (‘*else’ statement)? ‘IfEnd*‘

If the evaluation of the *expression* in parentheses is true, then the first *statement* is evaluated. If it is false, then if the optional else clause exists, it is evaluated. After this, the ‘IfEnd’ keyword is matched and the program leaves the if block.

Loops

There are two loops used in BATS: the while loop and for loop.
The while loop’s grammar is the following:

WhileBlock:
‘While’ *expression* ‘Do’ *statement* ‘WhileEnd’

While the Boolean value of the *expression* is true, then the *statement* is executed. When the *expression* value is false, the WhileEnd keyword is found and the program leaves the while block.
The for loop has the following grammar:

ForBlock:
‘For’ (*expression*)? ‘;’ (*expression*) ‘;’ (*expression*)? ‘Do’ *statement* ‘*ForEnd*’

The first and last *expressions* are optional. The first *expression* is typically initialization of a variable, like x = 0. The second *expression* is a test before iteration, similar to the *expression* found in *WhileBlock*. The last *expression* is usually used to increment or decrement of the *variable* initialized in the first *expression*. If the first and last *expressions* are not present, then the for loop just becomes a while loop.

Lack of a break statement

To create cleaner and more of a block structured language, a break statement found in Java and C is not found in the BATS language.

Return statement

A return statement is what a function uses to return to its caller. Each function must return in the BATS language, even if what is returned by the function is not assigned to anything. For example, a function can just return true and not assign the true value to anything. The grammar for return is the following:
Returnstmt:
‘Return’ *expression* ‘;’

Functions

A major part of the BATS language is the use of functions. To be used, they first must be defined in the following manner:

FunctionDecl:
  type-specify FunctionDeclarator FunctionBody

FunctionDeclarator:
  Identifier '(' (*Parameters*)? ')'

Parameters:
  type-specify Identifier Parameter-list

Parameter-list
  *',' Parameters | /*nothing*/*

FunctionBody
  '{' (Declarations)? Statement '}'

If, in *Parameters*, a Double is stated, but an integer is passed to that function, the integer is automatically converted to a double. The reverse does not occur. Functions cannot be nested; functions are only declared outside other functions in the global scope.

Arrays

Now we must realize how to get to a specific index of an array. The number of bracket sets must be of the same number of brackets when the Array was declared. The following is the grammar that is used to obtain information from an array.

ArrayUse:
  Identifier ('[' *Integer* ']')+

All one has to do to use an array is use *ArrayUse* after declaring an array and assigning the correct type of variables to the array. *Arrayuse* is an *expression*, which would evaluate the type and the value at the specified index. It has the same precedence as identifiers, that is, first level precedence.

Lines

Lines are special type of *Variables*, which can be compared to a single array of type *Point*. However, there is a special syntax for Lines. In fact, there are many ways for assigning a Line to a Line *Variable*, as is seen in the following grammar:

AssignLine:
  'From' *Point* ('to' *Point*)+
  | '(' *Point* (*Point*)+ ')'

Of course, the *Point* can be expanded to two *Double* variables or constants that have a comma between them. So that means these two Line assignment statements can each both expand to various lexical notations. The following are example of how Lines can be assigned to Line J (P1 and P2 are points, x1, x2, y1, and y2 are doubles or integers):

```
J <- From x1, y1 to x2, y2;
J <- ( x1, y1 to x2, y2);
J <- From P1 to P2 to x1, y1 to x2, y2;
J <- (P1 P2);
J <- (15.0, 11.6 17.5,7);
```

Scope

Global variables and functions, which are declared outside any other functions, have scope over the entire program; in other words, they have global scope. They can be used or called anywhere within the program. Any variables in a function have a local or lexical scope limited to the function. After there is a return from the function, the variables are thrown away.

Built-In Functions

This section discusses two important functions that are usable in BATS: Color and Draw.

Color

The Color function is used to specify the color of the next Line to be drawn (with the built-in function Draw). It takes in three double values (the Red, Green, and Blue value that constitute the RGB value) for the next lines until Color is called again. The default color for a line is black. The following is the grammar for Color

Color:
    'Color' *expression* ',' *expression* ',' *expression*

Calling the built-in function is an *expression*. Therefore, when it's a statement, a semicolon is added to the end to create something like the following example:

```
Color .05, .05, .05;
```

This makes the next lines gray.

Draw

This is the most important function in BATS, and it's built-in, ready to be used. Draw takes in a Line and produces it graphically on the screen in the viewer. The following is the grammar for Draw:

```
        Draw
            'Draw' Line
```

So you can have the following examples of drawing using Draw.

```
    Draw From x1, y1 to x2, y2;
    Draw ( x1, y1 to x2, y2);
    J <- From P1 to P2 to x1, y1 to x2, y2;         #if J is a Line
    Draw J;
    Draw (P1 P2);                                   #if p1 and p2 are points
    Draw (15.0, 11.6 17.5,7);
```

BATS, an example:

```
Start                                   #Start beings every BATS file
Global Line Xshape, Yshape, temp;
GlobalPoint x, y;
Boolean Begin                           #%This is the main file of a BATS program, the
                                        compiler seeks this out and starts running code from
                                        here%#
Int w, z;
w <- 100;
z <- 10;
x  <- 100,0;
y  <- 0,-100;
Xshape <- From –100,0 to x;             #making x and y axes
Yshape<-  From y to 0,100;
Draw Xshape;                            #drawing x and y axes
Draw Yshape;
    While w > 0 Do
        temp <- (z,0 0,w);
        Draw temp;                      #draw top right quadrant
        temp <- (z,0 0,-w);
        Draw temp;                      #draw bottom right quadrant
        temp <- (-z,0 0,w);
        Draw temp;                      #draw top left quadrant
        temp <- (-z,0 0,-w);
        Draw temp;                      #draw bottom left quadrant
        w -<- 10
        z +<- 10
    WhileEnd                            #note, indentation is not required
Return True;
End                                     #ends Begin function
Terminate                               #This ends every BATS file
```