Boomslang Language Reference Manual

Nathan Cuevas	Robert Kim	Nikhil Min Kovelamudi	David Steiner
njc2150	rk3145	nmk2146	ds3816

February 2021

Contents

1 Introduction

Boomslang is a general-purpose imperative programming language inspired by Python that aims to create a language as easy to use as Python, but with the added type safety of Java. The language is strongly and statically typed, has no type inferencing, and supports object-oriented programming without requiring that all functions be part of objects.

Boomslang uses syntactically significant whitespace as opposed to curly braces and semicolons. In addition to adding static typing to a Python-style syntax, Boomslang aims to add quality of life features such as automatic constructor generation for data classes, a new syntax for loops, and a better operator overloading syntax. Boomslang does not offer any garbage collection nor does it allow for pointer arithmetic.

2 Lexical Conventions

Boomslang has the following kinds of tokens: keywords, type names, identifiers, operators, literals, comments, punctuation, and syntactically significant whitespace. Boomslang uses the ASCII character set and is case-sensitive.

2.1 Keywords

Boomslang treats the following keywords as reserved and will always return a special token matching the name rather than treating it as any other type of token. The reserved keywords are: `not or and loop while if elif else def class return returns self required optional static NULL int long float boolean char string void true false`

The following are keywords in Python but not in Boomslang: `except lambda with as finally nonlocal assert None yield break for from continue global pass raise del import in is try True False`

2.2 Type names

Boomslang is a statically typed language. A type is either a primitive type, which means it matches one of the below types, a class name, or it is an array type.

2.2.1 Primitive type tokens

The primitive datatype tokens are: `int long float boolean char string void`

2.2.2 Class names

Class names must start with a capital letter, followed by one or more letters of any case. Class names are encouraged to take the UpperCamelCase form but are not required to by the compiler. Class names cannot contain anything other than letters. Class names are identified by the regular expression `['A'-'Z']['a'-'z' 'A'-'Z']*`

2.2.3 Array types

Array types consist of a primitive type token followed by a [, followed by an integer literal, followed by a closing].

2.3 Identifiers

Identifiers are case-sensitive. They consist of a lowercase letter followed by one or more lowercase letters, uppercase letters, numbers, or underscores in any order. Whereas class names always begin with an uppercase letter, identifiers always begin with a lowercase letter. Class names and identifiers are separate tokens. Keywords cannot be used as identifiers, so if a keyword is seen it will always be tokenized as the keyword above and not the identifier. Identifiers are identified by the regular expression `['a'-'z']['a'-'z' 'A'-'Z' '0'-'9' '_']*`

2.4 Operators

Boomslang includes the following tokens for operators: `+ - * / % = += -= *= /= == != > < >= <=`

See mathematical operators and boolean operators below for the semantic meaning of each operator.

2.4.1 Object operators

Boomslang allows for users to define custom infix operators as syntactic sugar for classes. See this section for a complete code example. An object operator is defined to be one or more special characters. This is the precise regex: `['+' '-' '%' '&' '$' '@' '!' '#' '^' '*' '/' '~' '?' '>' '<']+`

Note that if something matches both an object operator and a primitive operator, then the primitive operator token above will be emitted by the scanner/lexer.

2.5 Literals

Boomslang supports integer literals, floating point literals, boolean literals, character literals, and string literals.

2.5.1 Integer literals

Integer literals consist of one or more numbers between 0 and 9 next to each other. We do not support any syntax for numbers using `e`. The regular expression for int literals is `['0'-'9']+`

2.5.2 Long literals

Long literals consist of an integer literal followed by the character 'L'. For instance, "500L" would be treated as a literal for a long.

2.5.3 Floating point literals

Floating point literals consist of one or more numbers, followed by a period, followed by one or more numbers. We do not support any syntax for numbers using `e`. We also do not support starting a floating point with a period, such as in `.5`. The regular expression for float literals is `['0'-'9']+('.'['0'-'9']+)?`

2.5.4 Boolean literals

Boolean literals are either the string "true" or the string "false". These are case-sensitive, so "True" or "FALSE" will not match the boolean literal token. Boolean literals have a higher precedence than identifiers, so "true" and "false" always get tokenized as a boolean literal rather than an identifier. Thus "true" and "false" are not valid identifiers.

2.5.5 Character literals

Character literals consist of a single quote, followed by a single character, followed by a single quote. The exact regex to match character literals is `'.'`

2.5.6 String literals

String literals consist of a double quote, followed by any character other than a double quote or newline, followed by a double quote. String literals **cannot** contain newlines or double quote characters in Boomslang.

2.6 Comments

Boomslang supports both single-line and multi-line comments. Single-line comments start with a `#` character and indicate that everything after the `#` on that line is a comment and can be discarded.

Alternatively, a comment can be started using `/#` and closed using `#/`. Comments starting with `/#` are allowed to span multiple lines. The language uses `/#` and `#/` rather than `/*` and `*/` (C/Java style) or `"""` (Python style) to be more consistent with the single-line comment character. Since `#` is used for single-line comments, we view `/#` and `#/` as more consistent aesthetically than what is currently used in C, Java, or Python.

2.7 Punctuation

Boomslang uses the following characters for punctuation: `( ) [ ] : . , _`

- Left and right parentheses can be used to group expressions and are also used to define and call functions.
- Left and right brackets are used for array literals and array access.
- Colons are used when defining classes and functions.
- Periods are used to invoke functions that are part of objects.
- Commas are used to separate parameters to function calls as well as array parameters.
- Underscores are used exclusively to indicate that a method in a class is an object operator.

2.8 Syntactically significant whitespace

Boomslang uses syntactically significant whitespace rather than curly braces or semi-colons. Boomslang differs from Python in that spaces are not syntactically significant and only tab characters can be used.

The `NEWLINE` character is used to indicate the end of each statement. One `INDENT` token is used every time a line has increased indentation level compared to the previous indentation level. A `DEDENT` token is released for each corresponding decrease in the indentation level.

Consider the following code block for a concrete example:

```
1  int x = 5
2
3  def foo(string y) returns void:
4      char c = 'c'
5      if x > 5:
6          if c == 'c':
7              println("foo")
8      elif x > 10:
9          println("x was greater than 10")
10
11 println(1)
```

An `INDENT` token would be emitted after the `NEWLINE` on line 3. Another would be emitted after the `NEWLINE` on lines 5, 6, and 8. No `INDENT` would be emitted after the `NEWLINE` on line 4, because line 5 is at the same level of indentations. Thus `INDENT` is not the same as how many tab characters were on a line. Two `DEDENT` tokens would be emitted after the `NEWLINE` on line 7, and one `DEDENT` would be emitted after the `NEWLINE` on line 10.

3 Syntax

3.1 Conventions used in this manual

A context-free grammar is used to specify valid syntax for Boomslang programs. In this manual, nonterminal symbols will appear as *italicized-strings-in-lowercase* (clicking the nonterminal will open the section of the document where productions of that nonterminal are defined). Terminals will appear as uppercase strings in a monospaced font. For enhanced legibility, hyphens will be used to separate words within a nonterminal or terminal. An example of a terminal could be `TERMINAL`. If a terminal is also a keyword in the language, then it will be colored in orange as in `KEYWORD`. If there is more than one production for the same nonterminal symbol, the different alternatives will be listed on separate lines.

An example of a production would be

$$\textit{foo} \rightarrow \textit{bar}\ \texttt{CLASS}\ \texttt{BAZ}$$

Which would mean the nonterminal *foo* consists of the nonterminal *bar* followed by the keyword terminal `CLASS` followed by the terminal `BAZ`.

3.2 Types

3.2.1 Representation of primitives in memory

- `int` refers to a 32-bit integer and corresponds to `i32` in LLVM.
- `long` refers to a 64-bit integer and corresponds to `i64` in LLVM.
- `float` refers to a 32-bit floating point value and corresponds to `float` in LLVM.
- `char` refers to an ASCII character and corresponds to the `i8` type in LLVM.
- `string` refers to an array whose elements are all `char`. `string`'s are immutable in Boomslang.
- `boolean` refers to a value that is either "true" or "false" and corresponds to the `i1` type in LLVM.

3.2.2 Classes

The primitive types above can be combined to yield more powerful types. Boomslang allows users to define objects which are "manipulatable regions of storage" which consist of a struct that has fields made up of primitive types or other objects. Classes in Boomslang do not support inheritance.

Formally, we say a class definition consists of only the following rule:

classdecl → CLASS TYPE : NEWLINE
INDENT STATIC : NEWLINE INDENT *assigns* NEWLINE
DEDENT REQUIRED : NEWLINE INDENT *vdecls* NEWLINE
DEDENT OPTIONAL : NEWLINE INDENT *assigns* NEWLINE
DEDENT *optional-fdecls* NEWLINE

The fact that the entries appear on different lines here does not indicate that there is more than one production here. This was purely a cosmetic choice since the real production, which should all go on a single line, is too long to fit on this page.

Objects can be instantiated using the following rule.

object-instantiation → TYPE(*params*)
→ TYPE()

That is to say, an object can be instantiated with or without parameters in its constructor.

Fields within objects can be accessed directly (i.e. without needing to go through a function) using the following syntax:

object-variable-access → IDENTIFIER.IDENTIFIER
→ SELF.IDENTIFIER

SELF is a keyword that allows the programmer to access an object's field from within the object itself.

3.2.3 Arrays

Arrays are available as an aggregate data type. Arrays can consist of any type, e.g. primitives, objects, or other arrays. Boomslang does not allow for nested arrays, so something like `int[][] x` would not be legal.

Arrays can be defined via the following array literal syntax:

array-literal → [*params*]
→ []

The *params* inside the brackets must all have the same type. Thus [1, "1", 2.2] would not be a valid *array-literal*.

The entry at a given index within an array is accessed via the following syntax:

array-access → IDENTIFIER[*expr*]

Note that the *expr* inside the brackets for array accesses must evaluate to be an integer, although the parser isn't able to check that.

3.3 Functions

3.3.1 Function declarations

Functions in Boomslang do not need to be declared within a class. Functions can either specify their return type using the `returns` keyword followed by a TYPE, or if this is absent the function will behave as though it returned `void`.

fdecl → DEF IDENTIFIER(*type-params*) RETURNS TYPE : NEWLINE INDENT *stmts* DEDENT
→ DEF IDENTIFIER(*type-params*) : NEWLINE INDENT *stmts* DEDENT
→ DEF IDENTIFIER() RETURNS TYPE : NEWLINE INDENT *stmts* DEDENT
→ DEF IDENTIFIER() : NEWLINE INDENT *stmts* DEDENT
→ DEF UNDERSCORE OBJ-OPERATOR(TYPE IDENTIFIER) RETURNS TYPE : NEWLINE INDENT *stmts* DEDENT

An optional block of function declarations can be defined using the following rules:

$$\begin{aligned} \textit{optional-fdecls} &\rightarrow \textit{optional-fdecls}\ \ \textit{fdecl} \\ &\rightarrow \epsilon \end{aligned}$$

3.3.2 Function calls

Functions can either be called directly or be called as methods on an object. Functions can be called with or without parameters.

$$\begin{aligned} \textit{func-call} &\rightarrow \texttt{IDENTIFIER.IDENTIFIER}(\textit{params}) \\ &\rightarrow \texttt{SELF.IDENTIFIER}(\textit{params}) \\ &\rightarrow \texttt{IDENTIFIER}(\textit{params}) \\ &\rightarrow \texttt{IDENTIFIER.IDENTIFIER}() \\ &\rightarrow \texttt{SELF.IDENTIFIER}() \\ &\rightarrow \texttt{IDENTIFIER}() \end{aligned}$$

Note that the parentheses are *always* mandatory, even if the function takes no arguments.

3.4 Expressions

3.4.1 Literals

A literal by itself can be considered an expression. NULL is also considered an expression.

Thus all of the following productions are valid expressions:

$$\begin{aligned} \textit{expr} &\rightarrow \texttt{INT-LITERAL} \\ &\rightarrow \texttt{LONG-LITERAL} \\ &\rightarrow \texttt{FLOAT-LITERAL} \\ &\rightarrow \texttt{CHAR-LITERAL} \\ &\rightarrow \texttt{STRING-LITERAL} \\ &\rightarrow \texttt{BOOLEAN-LITERAL} \\ &\rightarrow \texttt{NULL} \end{aligned}$$

3.4.2 Identifiers, functions, and classes

Identifiers by themselves are considered valid expressions. This applies whether the identifier is used on its own or it refers to a field inside an object. Instantiations of objects are also valid expressions.

Thus all of the following productions are valid expressions:

$$\begin{aligned} \textit{expr} &\rightarrow \texttt{IDENTIFIER} \\ &\rightarrow \textit{object-instantiation} \\ &\rightarrow \textit{object-variable-access} \\ &\rightarrow \textit{func-call} \end{aligned}$$

3.4.3 Array expressions

Array expressions can be either an *array-literal* or an *array-access*. Thus the following productions are valid expressions:

$$\begin{aligned} \textit{expr} &\rightarrow \textit{array-literal} \\ &\rightarrow \textit{array-access} \end{aligned}$$

3.4.4 Parentheses

Expressions can be wrapped in parentheses. The type and value of an expression are the same as that of the original expression. Parentheses allow for grouping of expressions to override the default precedence for mathematical operators.

$$\textit{expr} \rightarrow (\textit{expr})$$

3.4.5 Mathematical operators

$$\begin{aligned} \textit{expr} &\rightarrow \textit{expr}\ \texttt{+}\ \textit{expr} \\ &\rightarrow \textit{expr}\ \texttt{-}\ \textit{expr} \\ &\rightarrow \textit{expr}\ \texttt{*}\ \textit{expr} \\ &\rightarrow \textit{expr}\ \texttt{/}\ \textit{expr} \\ &\rightarrow \textit{expr}\ \texttt{\%}\ \textit{expr} \\ &\rightarrow \textit{expr}\ \texttt{OBJ-OPERATOR}\ \textit{expr} \\ &\rightarrow \texttt{-}\textit{expr} \end{aligned}$$

All the *expr*'s used above for mathematical operators must evaluate to numeric types. See below for interoperability between different numeric types. + refers and − (when appearing between two expressions) refer to addition and subtraction, respectively, and group from left to right. They have equal precedence among themselves and are lower precedence than ∗, /, and %.

∗, /, and % refer to multiplication, division, and modulo, respectively, and group from left to right. They have equal precedence among themselves and are higher precedence than + and −.

`OBJ-OPERATOR` refers to a user defined infix operator, which appears as a method prefixed by an underscore on the class. See this section for more information and a code sample.

Unary minus, as in $-expr$, refers to the negative sign and has a higher precedence than the other mathematical operators.

3.4.6 Assignments

Assignments are also expressions. Assignment operators all have the same precedence, and their precedence is lower than all other operators. Assignment operators are right associative.

$$
\begin{aligned}
\textit{assign} &\rightarrow \texttt{TYPE IDENTIFIER =}\ \textit{expr} \\
&\rightarrow \texttt{IDENTIFIER =}\ \textit{expr} \\
&\rightarrow \textit{object-variable-access}\ \texttt{=}\ \textit{expr}
\end{aligned}
$$

Boomslang also includes the update and assign operators $+=$, $-=$, $*=$, and $/=$. These refer to setting the new value of the left-hand side equal to the old value of the left-hand side plus/minus/times/divided by the value of the right-hand side, respectively.

$$
\begin{aligned}
\textit{assign-update} &\rightarrow \texttt{IDENTIFIER +=}\ \textit{expr} \\
&\rightarrow \texttt{IDENTIFIER -=}\ \textit{expr} \\
&\rightarrow \texttt{IDENTIFIER *=}\ \textit{expr} \\
&\rightarrow \texttt{IDENTIFIER /=}\ \textit{expr} \\
&\rightarrow \textit{object-variable-access}\ \texttt{+=}\ \textit{expr} \\
&\rightarrow \textit{object-variable-access}\ \texttt{-=}\ \textit{expr} \\
&\rightarrow \textit{object-variable-access}\ \texttt{*=}\ \textit{expr} \\
&\rightarrow \textit{object-variable-access}\ \texttt{/=}\ \textit{expr}
\end{aligned}
$$

Thus the following are valid expressions:

$$
\begin{aligned}
\textit{expr} &\rightarrow \textit{assign} \\
&\rightarrow \textit{assign-update}
\end{aligned}
$$

Assignments can also be placed in a block, one after the other.

$$
\begin{aligned}
\textit{assigns} &\rightarrow \textit{assign} \\
&\rightarrow \textit{assigns}\ \texttt{NEWLINE}\ \textit{assign}
\end{aligned}
$$

3.4.7 Boolean operators

Boomslang supports comparison operators such as greater than, greater than equals, less than, less than equals, not equals, and equals. The language also supports `not`, `or`, and `and`.

All of the boolean operators are left associative. Their precedence, in ascending order of precedence, is `or`, `and`, `not`, followed by `==`, `!=`, `>`, `<`, `>=`, and `<=`, which all have the same level of precedence.

Boolean operators rank above assignments in terms of precedence but below mathematical operators.

In sum, the valid expressions involving boolean operators are:

$$
\begin{aligned}
\textit{expr} &\rightarrow \textit{expr}\ \texttt{==}\ \textit{expr} \\
&\rightarrow \textit{expr}\ \texttt{!=}\ \textit{expr} \\
&\rightarrow \textit{expr}\ \texttt{>}\ \textit{expr} \\
&\rightarrow \textit{expr}\ \texttt{<}\ \textit{expr} \\
&\rightarrow \textit{expr}\ \texttt{>=}\ \textit{expr} \\
&\rightarrow \textit{expr}\ \texttt{<=}\ \textit{expr} \\
&\rightarrow \texttt{NOT}\ \textit{expr} \\
&\rightarrow \textit{expr}\ \texttt{OR}\ \textit{expr} \\
&\rightarrow \textit{expr}\ \texttt{AND}\ \textit{expr}
\end{aligned}
$$

`==` and `!=` check value based equality for primitive types and check pointer equality for user defined types. That is to say, if `==` or `!=` are used for user defined objects, they are only deemed equal/not equal based on whether the objects refer to the same location in memory or not, respectively.

3.5 High-level program structure

3.5.1 The program

A program in Boomslang is made up of one or more statements, function declarations, class declarations, or newlines in any order, ending with the `EOF` token.

Formally, a program is:

$$\textit{program} \rightarrow \textit{program-without-eof}\ \texttt{EOF}$$

Where *program-without-eof* is:

$$\begin{aligned}\textit{program-without-eof} &\rightarrow \textit{program-without-eof}\ \ \textit{stmt}\\ &\rightarrow \textit{program-without-eof}\ \ \textit{fdecl}\\ &\rightarrow \textit{program-without-eof}\ \ \textit{classdecl}\\ &\rightarrow \textit{program-without-eof}\ \ \texttt{NEWLINE}\\ &\rightarrow \epsilon\end{aligned}$$

3.6 Statements

Boomslang makes a distinction between statements and expressions, even though both can have side effects and an expression by itself can be a statement. Each statement in Boomslang is terminated by a `NEWLINE`. No semicolons are used in Boomslang.

Boomslang has four types of statements: expressions, return statements, if statements, and loops. Formally, these are specified as:

$$\begin{aligned}\textit{stmt} &\rightarrow \textit{expr}\ \texttt{NEWLINE}\\ &\rightarrow \texttt{RETURN}\ \ \textit{expr}\ \texttt{NEWLINE}\\ &\rightarrow \textit{if-stmt}\\ &\rightarrow \textit{loop}\end{aligned}$$

A group of statements can appear one after the other:

$$\begin{aligned}\textit{stmts} &\rightarrow \textit{stmt}\ \texttt{NEWLINE}\\ &\rightarrow \textit{stmts}\ \ \textit{stmt}\end{aligned}$$

3.6.1 If statements

Users can branch on conditionals using if statements. If statements can be used as solo if statements, if/else statements, or if/elif/else statements, with an arbitrary amount of elifs.

Formally, if statements are defined as:

if-stmt → IF *expr* : NEWLINE INDENT *stmts* DEDENT
→ IF *expr* : NEWLINE INDENT *stmts* DEDENT ELSE : NEWLINE INDENT *stmts* DEDENT
→ IF *expr* : NEWLINE INDENT *stmts* DEDENT *elif* ELSE : NEWLINE INDENT *stmts* DEDENT
→ IF *expr* : NEWLINE INDENT *stmts* DEDENT *elif*

Where *elif* is defined as:

elif → ELIF *expr* : NEWLINE INDENT *stmts* DEDENT
→ *elif* ELIF *expr* : NEWLINE INDENT *stmts* DEDENT

3.6.2 Loop statements

Boomslang only offers one kind of loop, a "loop while" construct that uses a novel syntax.

loop → LOOP *expr* WHILE *expr* : NEWLINE INDENT *stmts* DEDENT
→ LOOP WHILE *expr* : NEWLINE INDENT *stmts* DEDENT

3.7 Parameters and variable declarations

As a statically typed language, Boomslang requires function declarations to specify the types of parameters. This is done using the following grammar, where each typed parameter is separated by a comma:

type-params → TYPE IDENTIFIER
→ *type-params* , TYPE IDENTIFIER

Parameters written without their type are also available when making function calls and initializing array literals. Their syntax is:

params → *expr*
→ *params* , *expr*

Variables can also be declared without being part of an assignment. This can be used to define the fields within a class.

vdecl → TYPE IDENTIFIER

vdecls → *vdecl*
→ *vdecls* NEWLINE *vdecl*

3.8 Associativity and precedence table

The table is increasing order of precedence.

Associativity	Operator(s)
right	`= += -= *= /=`
left	`OR`
left	`AND`
left	`NOT`
left	`== != > < >= <=`
left	`+ -`
left	`* / %`
left	`OBJ-OPERATOR`
non-associative	`UNARY-MINUS`

4 Semantics

4.1 Declaration rules

Variables and functions must be defined before they are used. Semantically,

```
int x = 5
println(x)  # prints 5
```

is legal but

```
x = 5
```

is not. The initial declaration of a variable must include its type, but afterwards it can be modified/reassigned without specifying its type. However, when this is done, the new value being assigned to it must match its original type.

Thus,

```
int x = 5
x = 6
println(x)  # prints 6
```

is legal but

```
int x = 5
x = "foo"
println(x)
```

is not, because x was declared to be an `int` but the next line is trying to assign a `string` to it.

4.2 Scoping rules

Scoping is determined by the level of indentation. A variable's scope applies starting on its level of indentation and all subsequent lines that have an indentation level greater than or equal to the indentation level of the most recent declaration or assignment.

Consider the following program as an example:

```
int x = 5
println(x)  # prints 5
def foo(int x) returns int:
    x = x + 1
```

```
    return x

println(foo(5))  # prints 6

if true:
    int x = 20
    println(x)  # prints 20

println(x)  # prints 5
```

Inside of classes, there is no concept of a public or private variable. All fields within classes are public and are visible throughout the class.

4.3 Function parameters are passed by value

Boomslang passes by value. Objects are passed using the Java style, which is like "passing by value of the reference." The following code sample elucidates passing primitives and objects into functions.

```
class MyObject:
    required:
        int x
        int y
        int z

MyObject my_object = MyObject(1, 2, 3)

int primitive_param = 1

# Before we call my_function_1, everything is as expected.
println(my_object.x)  # prints 1
println(my_object.y)  # prints 2
println(my_object.z)  # prints 3
println(primitive_param)  # prints 1

def my_function_1(MyObject object_param, int primitive_param):
    object_param.x = 500
    object_param.y = 500
    object_param.z = 500
    primitive_param = 500

my_function_1(my_object, primitive_param)

# After we call my_function_1, the object was mutated but the primitive param was not.
println(my_object.x)  # prints 500
println(my_object.y)  # prints 500
println(my_object.z)  # prints 500
println(primitive_param)  # prints 1

def my_function_2(MyObject object_param):
    object_param = MyObject(20, 20, 20)

my_old_object = my_object
my_function_2(my_object)
println(my_old_object == my_object)  # prints "true"
```

4.4 Overloading

The same function name can be reused as long as each one has a unique sequence of type parameters. Thus the following is legal:

```
def my_function_1(MyObject object_param, int primitive_param) returns int:
    return primitive_param

def my_function_1(int primitive_param, MyObject object_param) returns int:
    return primitive_param
```

4.5 Mutability

Objects in Boomslang are mutable by default. Strings and primitives are immutable.

4.6 Exception handling

Boomslang does not support any exception handling features. Thus `raises`, `throws`, `try`, `catch`, and `except` are not reserved keywords in the language.

5 Conversions

Some operators in Boomslang can cause conversion of the value of the operands from one type to another.

5.1 Floats and Integers

When floats and ints are combined in Boomslang arithmetic, the integer is treated as a floating point number and the result is always a float. This includes arithmetic expressions that include longs. If a long and an int are combined in any arithmetic operator, the result is always of type long.

5.2 Characters and Strings

When a character is added to a string using the `+` operator for concatenation, i.e. 'c' `+` "haracter", the resulting data type is a string, in this case "character". This string type conversion also occurs when 2 characters are concatenated, e.g. 'o'+'k' `==` "ok". Use of the `+` operator between chars and/or strings always converts to a string.

5.3 Object operators

Boomslang objects may also contain user-defined infix operator definitions in the form of special function definitions that define how objects work with operators. The type used in the parameter within the function declaration defines what types can work on the right hand side of the operator. The return type of the expression is defined by the return type of the function.

The following sample code demonstrates how the custom infix operators work in practice:

```
class MyClass:
    required:
        int x
        int y

    def _+(MyClass b) returns MyClass: # addition function
        return MyClass(self.x+b.x, self.y+b.y)

    def _-(MyClass b) returns MyClass: # subtraction function
        return MyClass(self.x-b.x, self.y-b.y)
```

```
    def _%%%(MyClass b) returns int:
        return self.x * b.x

MyClass instA = MyClass(1, 3)
MyClass instB = MyClass(2, 2)
MyClass sum = instA + instB # compiler will convert this to instA._+(instB)
MyClass difference = instA - instB # compiler will convert this to instA._-(instB)
int num = instA %%% instB  # compiler will convert this to instA._%%%(instB)
println(sum.x) # prints '3'
println(sum.y) # prints '5'
println(difference.x) # prints '-1'
println(difference.y) # prints '1'
println(num) # prints '2'
```