Boomslang

Nathan Cuevas
Robert Kim
Nikhil Kovelamudi
David Steiner
Boomslang in a Nutshell

- Python-inspired syntax
- **Static type checking** for safety and readability
- **Enhanced** object oriented features
  - Auto-generated constructors with required and optional parameters
  - Operator overloading syntax
  - Automatic to_string methods
- Automatic coercion between appropriate types (e.g. int and float)

A Boomslang is a large, venomous snake found in Africa

Boom = tree
Slang = snake
Motivation for **Boomslang**

- We wanted a language that was **fun and breezy to write in**
- Safe, readable, and opinionated
- We wanted to reduce boilerplate so that object-oriented programming wasn’t such a chore
- We wanted a solid set of fully-baked features
  - Arrays
  - Nulls
  - Primitive data types
  - Classes and generics
  - ...Many more
Boomslang in Depth

- Types
  - Primitives are int, long, float, char, string, bool, void
  - Class
  - Array (can be array of arrays)
  - Null

- A program is a sequence of one of three things
  - Statement
  - Function declaration
  - Class declaration
Boomslang in More Depth

- Strongly and statically typed - no type inferencing or duck typing
- Mutual recursion is allowed. Objects can reference other objects or themselves. Functions can call other functions or themselves. Classes and functions do not need to be defined before they are used.
- Compile and runtime exceptions
- Strings are first class: This means you can write things like “foo” == “bar” (false), “string” + “bar” (“stringbar”)
Key Features
Syntax

- **If, Elif, and Else** operate similarly to Python
- **Function Declaration**
  - Return type declaration required for non-void return. Formal types required
- **Loops** are a hybrid of for and while loops
  - Loop *(do this every loop)* while *(boolean expression passes)*
  - Statement after “loop” keyword can be omitted for pure while loop
- Variables declared inside functions/classes are **local variables** and outside are **global variables**
- **No main() function**

```plaintext
int x = 0
if x == 0:
    println(x)
elif x == 1:
    println(x+3)
else:
    println(x-2)

def foo(int a) returns void:
    println(a + 3)

int i = 0
loop i+=3 while i < 100:
    println(i)

int x = 5

def inc_x() returns void:
    x += 1

println(x)
inc_x()
inc_x()
```

*if/else branches*  
*loops*  
*function declaration*  
*globals*
Arrays

- Arrays supported for each available type
- Arrays can be initialized with default values, using the default construct
- Boomslang supports multidimensional arrays and array reassignment
- `len()` can dynamically get runtime size of the arrays

```java
int[] arr = default int[5]
string[][] arr = default string[11][2]
long[] arr = default long[0]
```

**default construct**

```java
int[] arr = [1,2]
arr = [9,8,7,6,5,4]
arr = default int[1000]
```

**array reassignment**

```java
int[][] arr = default int[3][2]
boolean[][] arrBool = default boolean[3][2]
int i = 0
int j = 0
loop i+=1 while i < len(arr):
    j = 0
    loop j+=1 while j < len(arr[0]):
        println(arr[i][j])
        println(arrBool[i][j])
```

**multidimensional arrays and len()**
Functions (1)

- **Useful Built-In Functions**
  - polymorphic println() function
  - type conversion functions such as int_to_float() and float_to_string()
  - concat_strings() function that can be implicitly called with ‘+’

```plaintext
println("PLT is awesome!")
println(1)
println(3.14)
println(false)
```

```plaintext
int x = 7
int y = 6

def my_multiply(int a, int b) returns int:
    return a * b

println(x + " times " + y + " is " + my_multiply(x,y))
```

**polymorphic printing**

<table>
<thead>
<tr>
<th>1</th>
<th>3.1400</th>
<th>false</th>
</tr>
</thead>
</table>

**string concat using ‘+’**

```plaintext
PLT is awesome!
7 times 6 is 42
```
Functions (2)

- Function overloading
- all functions/methods support standard and mutual recursion

```python
def is_even(int n) returns boolean:
    if n == 0:
        return true
    else:
        return is_odd(n - 1)

def is_odd(int n) returns boolean:
    if n == 0:
        return false
    else:
        return is_even(n - 1)
```

```python
int[] arr = [3, 16, 27, 35]
int i = 0
loop i+=1 while i < len(arr):
    int elem = arr[i]
    println(elem + " is even? "+ is_even(elem))
    println(elem + " is odd? "+ is_odd(elem))
```

**function overloading** 51.6340

**mutual recursion**
Classes (1)

- **Class Constructors** are both familiar and unfamiliar
- Static variables are modeled in LLVM as global variables (MyObject.x is a global variable named “@MyObject.x”)
- Required and optional variables are instance variables. What is the difference between them?

2 constructors are automatically generated:

```python
class MyObject:
    static:
        int x = 5
        string foo = "bar"
    required:
        int z
        float f0000
    optional:
        boolean boo = true

def construct(int z, float f0000):
    self.z = z
    self.f0000 = f0000
    self.boo = true

def construct(int z, float f0000, boolean boo):
    self.z = z
    self.f0000 = f0000
    self.boo = boo

MyObject mo = MyObject(10, 1.0)
println(mo.z)
println(mo.f0000)
```
Classes also come with a **built-in automatic to_string method**

```python
class MyObject:
    static:
        int x = 5
        string foo = "bar"

    required:
        int z
        float f0000

    optional:
        boolean boo = true

MyObject mo = MyObject(10, 1.0)
println(mo.z)
println(mo.f0000)
```

```bash
>println(mo)

MyObject:
x:5
foo:bar
z:10
f0000:1.0000
boo:true
```
Classes (3)

- Classes allow for chaining any valid expressions, be they variables or functions.
- So `foo.var.func().var.var.func().func()` could be a valid expression.
- In addition to calling `foo.mymethod()`, we have a special syntax for `object operators`.

```plaintext
>mylist ++ 2
```
Classes (4)

- **Boomslang** also supports **generic classes**
- These classes cannot be used directly, but can be **instantiated** to succinctly make new classes using the **generic template**

LIVE DEMO
Exceptions

- Compile time checks for
  - type compatibility
  - class/function declarations
  - variable initialization

- Runtime checks for division by zero and null objects

```java
class MyClass:
    required:
        int x

MyClass foo = NULL
foo.x
```

```python```
int y = 2
int x = 5 / (y - 2)
```

```
Exception DivideByZeroException
```

```java```
myfunc(string x) returns int:
    return 5

myfunc(5)
```

```
Fatal error: exception Failure("No matching signature found for function call myfunc")
```
Test-driven development

- Unit tests for lexer and parser utilizing the run_tests.py script (1018 lines of code)
- boomc shell script to test each file individually
- Over 250+ tests in the final repository
- REPL to troubleshoot issues
- Our AST and SAST are both able to be pretty printed as graphviz .dot files. 
  ./boomslang.native -a and ./boomslang.native -s , respectively
Examples

```python
def test_simple_assignment_passes_1(self):
    program = b"int x = 5
    self.assertProgramPasses(program)

def test_object_variable_access(self):
    program = b""
    class MyObject:
        static:
            int x = 5
    MyObject myobject = MyObject()
    myobject.x
    self.assertProgramPasses(program)

def test_invalid_assignment_fails_1(self):
    program = b"int x = \n"
    self.assertProgramFails(program)

def test_invalid_array_access_fails(self):
    program = b""
    int x = 5
    x[5]
```

```python
class Family:
    optional:
        string[] relation = default string[4][2]

def updateFathersName(string newName):
    self.relation[0][0] = newName

Family doeFamily = Family(["Mike","Jill","Jim","Kate"], ["dad","mom","son","daughter"])
doeFamily.updateFathersName("John")
int i = 0
loop i += 1 while i < 4:
    println(doeFamily.relation[0][i] + " is " + doeFamily.relation[1][i])

```

**final tests**

```
jact/src# ./repl
int x = "incorrect"
Fatal error: exception Failure("Illegal assignment. LHS was type int but RHS type was string")
```

**use repl for troubleshooting**
Future Work

● Show the user the line number where error occurred
● Automatic garbage collection
● Support for static functions
● Ability to import from other modules
● Working REPL for codegen (current REPL only goes up to semant)
● List comprehensions
● Inheritance
● Improvements to coercions
● Remove NULL (less is more, and Maybe is better than NULL)
Thank You!

Questions?